the event. The type of the adjustment is specified by *type*, and its associated value is *val*.

The **getAdjustable( )** method returns the object that generated the event. Its form is shown here:

Adjustable getAdjustable( )

The type of the adjustment event may be obtained by the **getAdjustmentType( )** method. It returns one of the constants defined by **AdjustmentEvent**. The general form is shown here:

int getAdjustmentType( )

The amount of the adjustment can be obtained from the **getValue( )** method, shown here:

int getValue( )

For example, when a scroll bar is manipulated, this method returns the value represented by that change.

The ComponentEvent Class

A **ComponentEvent** is generated when the size, position, or visibility of a component is changed. There are four types of component events. The **ComponentEvent** class defines integer constants that can be used to identify them. The constants and their meanings are shown here:

COMPONENT_HIDDEN	The component was hidden.
COMPONENT_MOVED	The component was moved.
COMPONENT_RESIZED	The component was resized.
COMPONENT_SHOWN	The component became visible.

ComponentEvent has this constructor:

ComponentEvent(Component *src*, int *type*)

Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type*.

ComponentEvent is the superclass either directly or indirectly of **ContainerEvent**, **FocusEvent**, **KeyEvent**, **MouseEvent**, and **WindowEvent**,

among others.

The **getComponent( )** method returns the component that generated the event. It is shown here:

Component getComponent( )

The ContainerEvent Class

A **ContainerEvent** is generated when a component is added to or removed from a container. There are two types of container events. The **ContainerEvent** class defines **int** constants that can be used to identify them: **COMPONENT_ADDED** and **COMPONENT_REMOVED**. They indicate that a component has been added to or removed from the container.

ContainerEvent is a subclass of **ComponentEvent** and has this constructor:

ContainerEvent(Component *src*, int *type*, Component *comp*)

Here, *src* is a reference to the container that generated this event. The type of the event is specified by *type*, and the component that has been added to or removed from the container is *comp*.

You can obtain a reference to the container that generated this event by using the **getContainer ( )** method, shown here:

Container getContainer( )

The **getChild( )** method returns a reference to the component that was added to or removed from the container. Its general form is shown here:

Component getChild( )

The FocusEvent Class

A **FocusEvent** is generated when a component gains or loses input focus. These events are identified by the integer constants **FOCUS_GAINED** and **FOCUS_LOST**.

FocusEvent is a subclass of **ComponentEvent** and has these constructors:

FocusEvent(Component *src*, int *type*)
FocusEvent(Component *src*, int *type*, boolean *temporaryFlag*)
FocusEvent(Component *src*, int *type*, boolean *temporaryFlag*, Component *other*)

FocusEvent(Component *src*, int *type*, boolean *temporaryFlag*, Component *other*,
FocusEvent.Cause *what*)

Here, *src* is a reference to the component that generated this event. The type of the event is specified by *type*. The argument *temporaryFlag* is set to **true** if the focus event is temporary. Otherwise, it is set to **false**. (A temporary focus event occurs as a result of another user interface operation. For example, assume that the focus is in a text field. If the user moves the mouse to adjust a scroll bar, the focus is temporarily lost.)

The other component involved in the focus change, called the *opposite component*, is passed in *other*. Therefore, if a **FOCUS_GAINED** event occurred, *other* will refer to the component that lost focus. Conversely, if a **FOCUS_LOST** event occurred, *other* will refer to the component that gains focus.

The fourth constructor was added by JDK 9. Its *what* parameter specifies why the event was generated. It is specified as a **FocusEvent.Cause** enumeration value that identifies the cause of the focus event. The **FocusEvent.Cause** enumeration was also added by JDK 9.

You can determine the other component by calling **getOppositeComponent( )**, shown here:

Component getOppositeComponent( )

The opposite component is returned.

The **isTemporary( )** method indicates if this focus change is temporary. Its form is shown here:

boolean isTemporary( )

The method returns **true** if the change is temporary. Otherwise, it returns **false**.

Beginning with JDK 9, you can obtain the cause of the event by calling **getCause( )**, shown here:

final FocusEvent.Cause getCause( )

The cause is returned in the form of a **FocusEvent.Cause** enumeration value.

The InputEvent Class

The abstract class **InputEvent** is a subclass of **ComponentEvent** and is the

superclass for component input events. Its subclasses are **KeyEvent** and **MouseEvent**.

InputEvent defines several integer constants that represent any modifiers, such as the control key being pressed, that might be associated with the event. Originally, the **InputEvent** class defined the following eight values to represent the modifiers, and these modfiers may still be found in older legacy code:

ALT_MASK	BUTTON2_MASK	META_MASK
ALT_GRAPH_MASK	BUTTON3_MASK	SHIFT_MASK
BUTTON1_MASK	CTRL_MASK	

However, because of possible conflicts between the modifiers used by keyboard events and mouse events, and other issues, the following extended modifier values were added:

ALT_DOWN_MASK	BUTTON2_DOWN_MASK	META_DOWN_MASK
ALT_GRAPH_DOWN_MASK	BUTTON3_DOWN_MASK	SHIFT_DOWN_MASK
BUTTON1_DOWN_MASK	CTRL_DOWN_MASK	

When writing new code, you should use the new, extended modifiers rather than the original modifiers. Furthermore, the original modifiers have have been deprecated by JDK 9.

To test if a modifier was pressed at the time an event is generated, use the **isAltDown( )**, **isAltGraphDown( )**, **isControlDown( )**, **isMetaDown( )**, and **isShiftDown( )** methods. The forms of these methods are shown here:

```
boolean isAltDown( )
boolean isAltGraphDown( )
boolean isControlDown( )
boolean isMetaDown( )
boolean isShiftDown( )
```

It is possible to obtain a value that contains all of the original modifier flags by calling the **getModifiers( )** method. It is shown here:

```
int getModifiers( )
```

Although you may still encounter **getModifiers( )** in legacy code, it is important to point out that because the original modifier flags have been deprecated by

JDK 9, this method has also been deprecated by JDK 9. Instead, you should obtain the extended modifiers by calling **getModifiersEx( )**, which is shown here:

int getModifiersEx( )

The ItemEvent Class

An **ItemEvent** is generated when a check box or a list item is clicked or when a checkable menu item is selected or deselected. (Check boxes and list boxes are described later in this book.) There are two types of item events, which are identified by the following integer constants:

DESELECTED	The user deselected an item.
SELECTED	The user selected an item.

In addition, **ItemEvent** defines the integer constant, **ITEM_STATE_CHANGED**, that signifies a change of state.

ItemEvent has this constructor:

ItemEvent(ItemSelectable *src*, int *type*, Object *entry*, int *state*)

Here, *src* is a reference to the component that generated this event. For example, this might be a list or choice element. The type of the event is specified by *type*. The specific item that generated the item event is passed in *entry*. The current state of that item is in *state*.

The **getItem( )** method can be used to obtain a reference to the item that changed. Its signature is shown here:

Object getItem( )

The **getItemSelectable( )** method can be used to obtain a reference to the **ItemSelectable** object that generated an event. Its general form is shown here:

ItemSelectable getItemSelectable( )

Lists and choices are examples of user interface elements that implement the **ItemSelectable** interface.

The **getStateChange( )** method returns the state change (that is, **SELECTED** or **DESELECTED**) for the event. It is shown here:

int getStateChange( )

The KeyEvent Class

A **KeyEvent** is generated when keyboard input occurs. There are three types of key events, which are identified by these integer constants: **KEY_PRESSED**, **KEY_RELEASED**, and **KEY_TYPED**. The first two events are generated when any key is pressed or released. The last event occurs only when a character is generated. Remember, not all keypresses result in characters. For example, pressing SHIFT does not generate a character.

There are many other integer constants that are defined by **KeyEvent**. For example, **VK_0** through **VK_9** and **VK_A** through **VK_Z** define the ASCII equivalents of the numbers and letters. Here are some others:

VK_ALT	VK_DOWN	VK_LEFT	VK_RIGHT
VK_CANCEL	VK_ENTER	VK_PAGE_DOWN	VK_SHIFT
VK_CONTROL	VK_ESCAPE	VK_PAGE_UP	VK_UP

The **VK** constants specify *virtual key codes* and are independent of any modifiers, such as control, shift, or alt.

KeyEvent is a subclass of **InputEvent**. Here is one of its constructors:

KeyEvent(Component *src*, int *type*, long *when*, int *modifiers*, int *code*, char *ch*)

Here, *src* is a reference to the component that generated the event. The type of the event is specified by *type*. The system time at which the key was pressed is passed in *when*. The *modifiers* argument indicates which modifiers were pressed when this key event occurred. The virtual key code, such as **VK_UP**, **VK_A**, and so forth, is passed in *code*. The character equivalent (if one exists) is passed in *ch*. If no valid character exists, then *ch* contains **CHAR_UNDEFINED**. For **KEY_TYPED** events, *code* will contain **VK_UNDEFINED**.

The **KeyEvent** class defines several methods, but probably the most commonly used ones are **getKeyChar( )**, which returns the character that was entered, and **getKeyCode( )**, which returns the key code. Their general forms are shown here:

char getKeyChar( )
int getKeyCode( )

If no valid character is available, then **getKeyChar( )** returns **CHAR_UNDEFINED**. When a **KEY_TYPED** event occurs, **getKeyCode( )** returns **VK_UNDEFINED**.

The MouseEvent Class

There are eight types of mouse events. The **MouseEvent** class defines the following integer constants that can be used to identify them:

MOUSE_CLICKED	The user clicked the mouse.
MOUSE_DRAGGED	The user dragged the mouse.
MOUSE_ENTERED	The mouse entered a component.
MOUSE_EXITED	The mouse exited from a component.
MOUSE_MOVED	The mouse moved.
MOUSE_PRESSED	The mouse was pressed.
MOUSE_RELEASED	The mouse was released.
MOUSE_WHEEL	The mouse wheel was moved.

MouseEvent is a subclass of **InputEvent**. Here is one of its constructors:

MouseEvent(Component *src*, int *type*, long *when*, int *modifiers*,
int *x*, int *y*, int *clicks*, boolean *triggersPopup*)

Here, *src* is a reference to the component that generated the event. The type of the event is specified by *type*. The system time at which the mouse event occurred is passed in *when*. The *modifiers* argument indicates which modifiers were pressed when a mouse event occurred. The coordinates of the mouse are passed in *x* and *y*. The click count is passed in *clicks*. The *triggersPopup* flag indicates if this event causes a pop-up menu to appear on this platform.

Two commonly used methods in this class are **getX( )** and **getY( )**. These return the X and Y coordinates of the mouse within the component when the event occurred. Their forms are shown here:

```
int getX( )
int getY( )
```

Alternatively, you can use the **getPoint( )** method to obtain the coordinates of the mouse. It is shown here:

Point getPoint( )

It returns a **Point** object that contains the X,Y coordinates in its integer members: **x** and **y**.

The **translatePoint( )** method changes the location of the event. Its form is shown here:

void translatePoint(int *x*, int *y*)

Here, the arguments *x* and *y* are added to the coordinates of the event.

The **getClickCount( )** method obtains the number of mouse clicks for this event. Its signature is shown here:

int getClickCount( )

The **isPopupTrigger( )** method tests if this event causes a pop-up menu to appear on this platform. Its form is shown here:

boolean isPopupTrigger( )

Also available is the **getButton( )** method, shown here:

int getButton( )

It returns a value that represents the button that caused the event. For most cases, the return value will be one of these constants defined by **MouseEvent**:

NOBUTTON	BUTTON1	BUTTON2	BUTTON3

The **NOBUTTON** value indicates that no button was pressed or released.

Also available are three methods that obtain the coordinates of the mouse relative to the screen rather than the component. They are shown here:

Point getLocationOnScreen( )

int getXOnScreen( )

int getYOnScreen( )

The **getLocationOnScreen( )** method returns a **Point** object that contains both the X and Y coordinate. The other two methods return the indicated coordinate.

The MouseWheelEvent Class

The **MouseWheelEvent** class encapsulates a mouse wheel event. It is a subclass of **MouseEvent**. Not all mice have wheels. If a mouse has a wheel, it is typically located between the left and right buttons. Mouse wheels are used for scrolling. **MouseWheelEvent** defines these two integer constants:

WHEEL_BLOCK_SCROLL	A page-up or page-down scroll event occurred.
WHEEL_UNIT_SCROLL	A line-up or line-down scroll event occurred.

Here is one of the constructors defined by **MouseWheelEvent**:

MouseWheelEvent(Component *src*, int *type*, long *when*, int *modifiers*,
int *x*, int *y*, int *clicks*, boolean *triggersPopup*,
int *scrollHow*, int *amount*, int *count*)

Here, *src* is a reference to the object that generated the event. The type of the event is specified by *type*. The system time at which the mouse event occurred is passed in *when*. The *modifiers* argument indicates which modifiers were pressed when the event occurred. The coordinates of the mouse are passed in *x* and *y*. The number of clicks is passed in *clicks*. The *triggersPopup* flag indicates if this event causes a pop-up menu to appear on this platform. The *scrollHow* value must be either **WHEEL_UNIT_SCROLL** or **WHEEL_BLOCK_ SCROLL**. The number of units to scroll is passed in *amount*. The *count* parameter indicates the number of rotational units that the wheel moved.

MouseWheelEvent defines methods that give you access to the wheel event. To obtain the number of rotational units, call **getWheelRotation( )**, shown here:

int getWheelRotation( )

It returns the number of rotational units. If the value is positive, the wheel moved counterclockwise. If the value is negative, the wheel moved clockwise. Also available is a method called **getPreciseWheelRotation( )**, which supports high-resolution wheels. It works like **getWheelRotation( )**, but returns a **double**.

To obtain the type of scroll, call **getScrollType( )**, shown next:

int getScrollType( )

It returns either **WHEEL_UNIT_SCROLL** or **WHEEL_BLOCK_SCROLL**.

If the scroll type is **WHEEL_UNIT_SCROLL**, you can obtain the number of units to scroll by calling **getScrollAmount( )**. It is shown here:

int getScrollAmount( )

The TextEvent Class

Instances of this class describe text events. These are generated by text fields and text areas when characters are entered by a user or program. **TextEvent** defines the integer constant **TEXT_VALUE_CHANGED**.

The one constructor for this class is shown here:

TextEvent(Object *src*, int *type*)

Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type*.

The **TextEvent** object does not include the characters currently in the text component that generated the event. Instead, your program must use other methods associated with the text component to retrieve that information. This operation differs from other event objects discussed in this section. Think of a text event notification as a signal to a listener that it should retrieve information from a specific text component.

The WindowEvent Class

There are ten types of window events. The **WindowEvent** class defines integer constants that can be used to identify them. The constants and their meanings are shown here:

WINDOW_ACTIVATED	The window was activated.
WINDOW_CLOSED	The window has been closed.
WINDOW_CLOSING	The user requested that the window be closed.
WINDOW_DEACTIVATED	The window was deactivated.
WINDOW_DEICONIFIED	The window was deiconified.
WINDOW_GAINED_FOCUS	The window gained input focus.
WINDOW_ICONIFIED	The window was iconified.
WINDOW_LOST_FOCUS	The window lost input focus.
WINDOW_OPENED	The window was opened.
WINDOW_STATE_CHANGED	The state of the window changed.

WindowEvent is a subclass of **ComponentEvent**. It defines several constructors. The first is

WindowEvent(Window *src*, int *type*)

Here, *src* is a reference to the object that generated this event. The type of the event is *type*.

The next three constructors offer more detailed control:

WindowEvent(Window *src*, int *type*, Window *other*)
WindowEvent(Window *src*, int *type*, int *fromState*, int *toState*)
WindowEvent(Window *src*, int *type*, Window *other*, int *fromState*, int *toState*)

Here, *other* specifies the opposite window when a focus or activation event occurs. The *fromState* specifies the prior state of the window, and *toState* specifies the new state that the window will have when a window state change occurs.

A commonly used method in this class is **getWindow( )**. It returns the **Window** object that generated the event. Its general form is shown here:

Window getWindow( )

WindowEvent also defines methods that return the opposite window (when a focus or activation event has occurred), the previous window state, and the current window state. These methods are shown here:

Window getOppositeWindow( )
int getOldState( )
int getNewState( )

Sources of Events

Table 24-2 lists some of the user interface components that can generate the events described in the previous section. In addition to these graphical user interface elements, any class derived from **Component**, such as **Frame**, can generate events. For example, you can receive key and mouse events from an instance of **Frame**. In this chapter, we will be handling only mouse and keyboard events, but subsequent chapters will be handling events from a variety of sources.

Event Source	Description
Button	Generates action events when the button is pressed.
Check box	Generates item events when the check box is selected or deselected.
Choice	Generates item events when the choice is changed.
List	Generates action events when an item is double-clicked; generates item events when an item is selected or deselected.
Menu item	Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.
Scroll bar	Generates adjustment events when the scroll bar is manipulated.
Text components	Generates text events when the user enters a character.
Window	Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Table 24-2 Event Source Examples

Event Listener Interfaces

As explained, the delegation event model has two parts: sources and listeners. As it relates to this chapter, listeners are created by implementing one or more of the interfaces defined by the **java.awt.event** package. When an event occurs, the event source invokes the appropriate method defined by the listener and provides an event object as its argument. Table 24-3 lists several commonly used listener interfaces and provides a brief description of the methods that they define. The following sections examine the specific methods that are contained in each interface.

Interface	Description
ActionListener	Defines one method to receive action events.
AdjustmentListener	Defines one method to receive adjustment events.
ComponentListener	Defines four methods to recognize when a component is hidden, moved, resized, or shown.
ContainerListener	Defines two methods to recognize when a component is added to or removed from a container.
FocusListener	Defines two methods to recognize when a component gains or loses keyboard focus.
ItemListener	Defines one method to recognize when the state of an item changes.
KeyListener	Defines three methods to recognize when a key is pressed, released, or typed.
MouseListener	Defines five methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released.
MouseMotionListener	Defines two methods to recognize when the mouse is dragged or moved.
MouseWheelListener	Defines one method to recognize when the mouse wheel is moved.
TextListener	Defines one method to recognize when a text value changes.
WindowFocusListener	Defines two methods to recognize when a window gains or loses input focus.
WindowListener	Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Table 24-3 Commonly Used Event Listener Interfaces

The ActionListener Interface

This interface defines the **actionPerformed( )** method that is invoked when an action event occurs. Its general form is shown here:

void actionPerformed(ActionEvent *ae*)

The AdjustmentListener Interface

This interface defines the **adjustmentValueChanged( )** method that is invoked when an adjustment event occurs. Its general form is shown here:

void adjustmentValueChanged(AdjustmentEvent *ae*)

The ComponentListener Interface

This interface defines four methods that are invoked when a component is

resized, moved, shown, or hidden. Their general forms are shown here:

void componentResized(ComponentEvent *ce*)
void componentMoved(ComponentEvent *ce*)
void componentShown(ComponentEvent *ce*)
void componentHidden(ComponentEvent *ce*)

The ContainerListener Interface

This interface contains two methods. When a component is added to a container, **componentAdded( )** is invoked. When a component is removed from a container, **componentRemoved( )** is invoked. Their general forms are shown here:

void componentAdded(ContainerEvent *ce*)
void componentRemoved(ContainerEvent *ce*)

The FocusListener Interface

This interface defines two methods. When a component obtains keyboard focus, **focusGained( )** is invoked. When a component loses keyboard focus, **focusLost( )** is called. Their general forms are shown here:

void focusGained(FocusEvent *fe*)
void focusLost(FocusEvent *fe*)

The ItemListener Interface

This interface defines the **itemStateChanged( )** method that is invoked when the state of an item changes. Its general form is shown here:

void itemStateChanged(ItemEvent *ie*)

The KeyListener Interface

This interface defines three methods. The **keyPressed( )** and **keyReleased( )** methods are invoked when a key is pressed and released, respectively. The **keyTyped( )** method is invoked when a character has been entered.

For example, if a user presses and releases the A key, three events are generated in sequence: key pressed, typed, and released. If a user presses and

releases the HOME key, two key events are generated in sequence: key pressed and released.

The general forms of these methods are shown here:

void keyPressed(KeyEvent *ke*)
void keyReleased(KeyEvent *ke*)
void keyTyped(KeyEvent *ke*)

The MouseListener Interface

This interface defines five methods. If the mouse is pressed and released at the same point, **mouseClicked( )** is invoked. When the mouse enters a component, the **mouseEntered( )** method is called. When it leaves, **mouseExited( )** is called. The **mousePressed( )** and **mouseReleased( )** methods are invoked when the mouse is pressed and released, respectively.

The general forms of these methods are shown here:

void mouseClicked(MouseEvent *me*)
void mouseEntered(MouseEvent *me*)
void mouseExited(MouseEvent *me*)
void mousePressed(MouseEvent *me*)
void mouseReleased(MouseEvent *me*)

The MouseMotionListener Interface

This interface defines two methods. The **mouseDragged( )** method is called multiple times as the mouse is dragged. The **mouseMoved( )** method is called multiple times as the mouse is moved. Their general forms are shown here:

void mouseDragged(MouseEvent *me*)
void mouseMoved(MouseEvent *me*)

The MouseWheelListener Interface

This interface defines the **mouseWheelMoved( )** method that is invoked when the mouse wheel is moved. Its general form is shown here:

void mouseWheelMoved(MouseWheelEvent *mwe*)

The TextListener Interface

This interface defines the **textValueChanged( )** method that is invoked when a change occurs in a text area or text field. Its general form is shown here:

```
void textValueChanged(TextEvent te)
```

The WindowFocusListener Interface

This interface defines two methods: **windowGainedFocus( )** and **windowLostFocus( )**. These are called when a window gains or loses input focus. Their general forms are shown here:

```
void windowGainedFocus(WindowEvent we)
void windowLostFocus(WindowEvent we)
```

The WindowListener Interface

This interface defines seven methods. The **windowActivated( )** and **windowDeactivated( )** methods are invoked when a window is activated or deactivated, respectively. If a window is iconified, the **windowIconified( )** method is called. When a window is deiconified, the **windowDeiconified( )** method is called. When a window is opened or closed, the **windowOpened( )** or **windowClosed( )** methods are called, respectively. The **windowClosing( )** method is called when a window is being closed. The general forms of these methods are

```
void windowActivated(WindowEvent we)
void windowClosed(WindowEvent we)
void windowClosing(WindowEvent we)
void windowDeactivated(WindowEvent we)
void windowDeiconified(WindowEvent we)
void windowIconified(WindowEvent we)
void windowOpened(WindowEvent we)
```

Using the Delegation Event Model

Now that you have learned the theory behind the delegation event model and have had an overview of its various components, it is time to see it in practice.

Using the delegation event model is actually quite easy. Just follow these two steps:

1. Implement the appropriate interface in the listener so that it can receive the type of event desired.
2. Implement code to register and unregister (if necessary) the listener as a recipient for the event notifications.

Remember that a source may generate several types of events. Each event must be registered separately. Also, an object may register to receive several types of events, but it must implement all of the interfaces that are required to receive these events. In all cases, an event handler must return quickly. As explained earlier, an event handler must not retain control for an extended period of time.

To see how the delegation model works in practice, we will look at examples that handle two common event generators: the mouse and keyboard.

Some Key AWT GUI Concepts

To demonstrate the fundamentals of event handling, we will use several simple, GUI-based programs. As stated earlier, most events to which your program will respond will be generated by user interaction with GUI programs. Although the GUI programs shown in this chapter are very simple, it is still necessary to explain a few key concepts because GUI-based programs differ from the console-based programs found in many other parts of this book.

Before we begin, it is important to point out that all modern versions of Java support two GUI frameworks: the AWT and Swing. The AWT was Java's first GUI framework, and for very limited GUI programs, it is the easiest to use. Swing, which is built on the foundation of the AWT, was Java's second GUI framework and is its most popular and widely used. (A third Java GUI called JavaFX was provided with several recent versions of Java. However, beginning JDK 11, it is no longer part of the JDK.) Both the AWT and Swing are discussed later in this book. However, to demonstrate the fundamentals of event handling, simple AWT-based GUI programs are an appropriate choice and are used here.

There are four key AWT features used by the following programs. First, all create a top-level window by extending the **Frame** class. **Frame** defines what one would think of as a "normal" window. For example, it has minimize, maximize, and close boxes. It can be resized, covered, and redisplayed. Second, all override the **paint( )** method to display output in the window. This method is called by the run-time system to display output in the window. For example, it is

called when a window is first shown and after a window has been hidden and then uncovered. Third, when your program needs output displayed, it does not call **paint( )** directly. Instead, you call **repaint( )**. In essence, **repaint( )** tells the AWT to call **paint( )**. You will see how the process works in the examples that follow. Finally, when the top-level **Frame** window for an application is closed—for example, by clicking its close box—the program must explicitly exit, often through a call to **System.exit( )**. Clicking the close box, by itself, does not cause the program to terminate. Therefore, it is necessary for an AWT-based GUI program to handle a window-close event.

Handling Mouse Events

To handle mouse events, you must implement the **MouseListener** and the **MouseMotionListener** interfaces. (You may also want to implement **MouseWheelListener**, but we won't be doing so, here.) The following program demonstrates the process. It displays the current coordinates of the mouse in the program's window. Each time a button is pressed, the phrase "Button Down" is displayed at the location of the mouse pointer. Each time the button is released, the phrase "Button Released" is shown. If a button is clicked, a message stating this fact is displayed at the current mouse location.

As the mouse enters or exits the window, a message is displayed that indicates what happened. When dragging the mouse, a * is shown, which tracks with the mouse pointer as it is dragged. Notice that the two variables, **mouseX** and **mouseY**, store the location of the mouse when a mouse pressed, released, or dragged event occurs. These coordinates are then used by **paint( )** to display output at the point of these occurrences.

```
// Demonstrate several mouse event handlers.
import java.awt.*;
import java.awt.event.*;

public class MouseEventsDemo extends Frame
  implements MouseListener, MouseMotionListener {

  String msg = "";
  int mouseX = 0, mouseY = 0; // coordinates of mouse

  public MouseEventsDemo() {
     addMouseListener(this);
     addMouseMotionListener(this);
     addWindowListener(new MyWindowAdapter());
  }

// Handle mouse clicked.
public void mouseClicked(MouseEvent me) {
  msg = msg + " -- click received";
  repaint();
}

// Handle mouse entered.
public void mouseEntered(MouseEvent me) {
  mouseX = 100;
  mouseY = 100;
  msg = "Mouse entered.";
  repaint();
}
```

```
// Handle mouse exited.
public void mouseExited(MouseEvent me) {
  mouseX = 100;
  mouseY = 100;
  msg = "Mouse exited.";
  repaint();
}

// Handle button pressed.
public void mousePressed(MouseEvent me) {
  // save coordinates
  mouseX = me.getX();
  mouseY = me.getY();
  msg = "Button down";
  repaint();
}

// Handle button released.
public void mouseReleased(MouseEvent me) {
  // save coordinates
  mouseX = me.getX();
  mouseY = me.getY();
  msg = "Button Released";
  repaint();
}

// Handle mouse dragged.
public void mouseDragged(MouseEvent me) {
  // save coordinates
  mouseX = me.getX();
  mouseY = me.getY();
  msg = "*" + " mouse at " + mouseX + ", " + mouseY;
  repaint();
}
```

```
// Handle mouse moved.
public void mouseMoved(MouseEvent me) {
  msg = "Moving mouse at " + me.getX() + ", " + me.getY();
  repaint();
}

  // Display msg in the window at current X,Y location.
  public void paint(Graphics g) {
    g.drawString(msg, mouseX, mouseY);
  }

  public static void main(String[] args) {
    MouseEventsDemo appwin = new MouseEventsDemo();

    appwin.setSize(new Dimension(300, 300));
    appwin.setTitle("MouseEventsDemo");
    appwin.setVisible(true);
  }
}

// When the close box in the frame is clicked,
// close the window and exit the program.
class MyWindowAdapter extends WindowAdapter {
  public void windowClosing(WindowEvent we) {
    System.exit(0);
  }
}
```

Sample output from this program is shown here:

Let’s look closely at this example. First, notice that **MouseEventsDemo** extends **Frame**. Thus, it forms the top-level window for the application. Next, notice that it implements both the **MouseListener** and **MouseMotionListener** interfaces. These two interfaces contain methods that receive and process various types of mouse events. Notice that **MouseEventsDemo** is both the source and the listener for these events. This works because **Frame** supplies the **addMouseListener( )** and **addMouseMotionListener( )** methods. Being both the source and the listener for events is not uncommon for simple GUI programs.

Inside the **MouseEventsDemo** constructor, the program registers itself as a listener for mouse events. This is done by calling **addMouseListener( )** and **addMouseMotionListener( )**. They are shown here:

void addMouseListener(MouseListener *ml*)
void addMouseMotionListener(MouseMotionListener *mml*)

Here, *ml* is a reference to the object receiving mouse events, and *mml* is a reference to the object receiving mouse motion events. In this program, the same object is used for both. **MouseEventsDemo** then implements all of the methods defined by the **MouseListener** and **MouseMotionListener** interfaces. These are the event handlers for the various mouse events. Each method handles its event and then returns.

Notice that the **MouseEventsDemo** constructor also adds a **WindowListener**. This is needed to enable the program to respond to a window close event when the user clicks the close box. This listener uses an *adapter*

class to implement the **WindowListener** interface. Adapter classes supply empty implementations of a listener interface, enabling you to override only the method or methods in which you are interested. They are described in detail later in this chapter, but one is used here to greatly simplify the code needed to close the program. In this case, the **windowClosing( )** method is overridden. This method is called by the AWT when the window is closed. Here, it calls **System.exit( )** to end the program.

Now notice the mouse event handlers. Each time a mouse event occurs, **msg** is assigned a string that describes what happened and then **repaint( )** is called. In this case, **repaint( )** ultimately causes the AWT to call **paint( )** to display output. (This process is examined in greater detail in Chapter 25.) Notice that **paint( )** has a parameter of type **Graphics**. This class describes the *graphics context* of the program. It is required for output. The program uses the **drawString( )** method provided by **Graphics** to actually display a string in the window at the specified X, Y location. The form used in the program is shown here:

void drawString(String *message*, int *x*, int *y*)

Here, *message* is the string to be output beginning at *x*, *y*. In a Java window, the upper-left corner is location 0,0. As mentioned, **mouseX** and **mouseY** keep track of the location of the mouse. These values are passed to **drawString( )** as the location at which output is displayed.

Finally, the program is started by creating a **MouseEventsDemo** instance and then setting the size of the window, its title, and making the window visible. These features are described in greater detail in Chapter 25.

Handling Keyboard Events

To handle keyboard events, you use the same general architecture as that shown in the mouse event example in the preceding section. The difference, of course, is that you will be implementing the **KeyListener** interface.

Before looking at an example, it is useful to review how key events are generated. When a key is pressed, a **KEY_PRESSED** event is generated. This results in a call to the **keyPressed( )** event handler. When the key is released, a **KEY_RELEASED** event is generated and the **keyReleased( )** handler is executed. If a character is generated by the keystroke, then a **KEY_TYPED** event is sent and the **keyTyped( )** handler is invoked. Thus, each time the user presses a key, at least two and often three events are generated. If all you care about are actual characters, then you can ignore the information passed by the

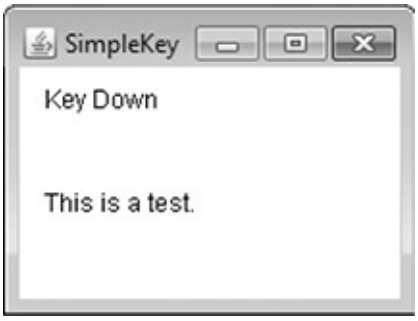
key press and release events. However, if your program needs to handle special keys, such as the arrow or function keys, then it must watch for them through the **keyPressed( )** handler.

The following program demonstrates keyboard input. It echoes keystrokes to the window and shows the pressed/released status of each key.

```
// Demonstrate the key event handlers.
import java.awt.*;
import java.awt.event.*;

public class SimpleKey extends Frame
  implements KeyListener {

  String msg = "";
  String keyState = "";

  public SimpleKey() {
    addKeyListener(this);
    addWindowListener(new MyWindowAdapter());
  }

  // Handle a key press.
  public void keyPressed(KeyEvent ke) {
    keyState = "Key Down";
    repaint();
  }

  // Handle a key release.
  public void keyReleased(KeyEvent ke) {
    keyState = "Key Up";
    repaint();
  }
```

```
  // Handle key typed.
  public void keyTyped(KeyEvent ke) {
    msg += ke.getKeyChar();
    repaint();
  }

  // Display keystrokes.
  public void paint(Graphics g) {
    g.drawString(msg, 20, 100);
    g.drawString(keyState, 20, 50);
  }

  public static void main(String[] args) {
    SimpleKey appwin = new SimpleKey();

    appwin.setSize(new Dimension(200, 150));
    appwin.setTitle("SimpleKey");
    appwin.setVisible(true);
  }
}

// When the close box in the frame is clicked,
// close the window and exit the program.
class MyWindowAdapter extends WindowAdapter {
  public void windowClosing(WindowEvent we) {
    System.exit(0);
  }
}
```

Sample output is shown here:



If you want to handle the special keys, such as the arrow or function keys, you need to respond to them within the **keyPressed( )** handler. They are not

available through **keyTyped( )**. To identify the keys, you use their virtual key codes. For example, the next program outputs the name of a few of the special keys:

```
// Demonstrate some virtual key codes.
import java.awt.*;
import java.awt.event.*;

public class KeyEventsDemo extends Frame
  implements KeyListener {

  String msg = "";
  String keyState = "";

  public KeyEventsDemo() {
    addKeyListener(this);
    addWindowListener(new MyWindowAdapter());
  }

  // Handle a key press.
  public void keyPressed(KeyEvent ke) {
    keyState = "Key Down";

    int key = ke.getKeyCode();
    switch(key) {
      case KeyEvent.VK_F1:
        msg += "<F1>";
        break;
      case KeyEvent.VK_F2:
        msg += "<F2>";
        break;
      case KeyEvent.VK_F3:
        msg += "<F3>";
        break;
      case KeyEvent.VK_PAGE_DOWN:
        msg += "<PgDn>";
        break;
```

```
      case KeyEvent.VK_PAGE_UP:
        msg += "<PgUp>";
        break;
      case KeyEvent.VK_LEFT:
        msg += "<Left Arrow>";
        break;
      case KeyEvent.VK_RIGHT:
        msg += "<Right Arrow>";
        break;
    }

    repaint();
  }

  // Handle a key release.
  public void keyReleased(KeyEvent ke) {
    keyState = "Key Up";
    repaint();
  }

  // Handle key typed.
  public void keyTyped(KeyEvent ke) {
    msg += ke.getKeyChar();
    repaint();
  }

  // Display keystrokes.
  public void paint(Graphics g) {
    g.drawString(msg, 20, 100);
    g.drawString(keyState, 20, 50);
  }
```

```
  public static void main(String[] args) {
    KeyEventsDemo appwin = new KeyEventsDemo();

    appwin.setSize(new Dimension(200, 150));
    appwin.setTitle("KeyEventsDemo");
    appwin.setVisible(true);
  }
}

// When the close box in the frame is clicked,
// close the window and exit the program.
class MyWindowAdapter extends WindowAdapter {
  public void windowClosing(WindowEvent we) {
    System.exit(0);
  }
}
```

Sample output is shown here:

The procedures shown in the preceding keyboard and mouse event examples can be generalized to any type of event handling, including those events generated by controls. In later chapters, you will see many examples that handle other types of events, but they will all follow the same basic structure as the programs just described.

Adapter Classes

Java provides a special feature, called an *adapter class*, that can simplify the creation of event handlers in certain situations. An adapter class provides an empty implementation of all methods in an event listener interface. Adapter classes are useful when you want to receive and process only some of the events

that are handled by a particular event listener interface. You can define a new class to act as an event listener by extending one of the adapter classes and implementing only those events in which you are interested.

For example, the **MouseMotionAdapter** class has two methods, **mouseDragged( )** and **mouseMoved( )**, which are the methods defined by the **MouseMotionListener** interface. If you were interested in only mouse drag events, then you could simply extend **MouseMotionAdapter** and override **mouseDragged( )**. The empty implementation of **mouseMoved( )** would handle the mouse motion events for you.

Table 24-4 lists several commonly used adapter classes in **java.awt.event** and notes the interface that each implements.

Adapter Class	Listener Interface
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener, MouseMotionListener, and MouseWheelListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener, WindowFocusListener, and WindowStateListener

Table 24-4 Commonly Used Listener Interfaces Implemented by Adapter Classes

You have already seen one adapter class in action in the preceding examples: **WindowAdapter**. Recall that the **WindowListener** interface defines seven methods, but only one, **windowClosing( )**, was needed by the programs. The use of the adapter prevented the need to provide empty implementations of the other unused methods, thus avoiding clutter in the examples. As you would expect, the other adapter classes can be employed in a similar fashion.

The following program provides another example of an adapter. It uses **MouseAdapter** to respond to mouse click and mouse drag events. As shown in Table 24-4, **MouseAdapter** implements all of the mouse listener interfaces. Thus, you can use it to handle all types of mouse events. Of course, you need override only those methods that are used by your program. In the following example, **MyMouseAdapter** extends **MouseAdapter** and overrides the

mouseClicked( ) and **mouseDragged( )** methods. All other mouse events are silently ignored. Notice that the **MyMouseAdapter** constructor is passed a reference to the **AdapterDemo** instance. This reference is saved and then used to assign a string to **msg** and to invoke **repaint( )** on the object that receives the event notification. As before, a **WindowAdapter** is used to handle a window closing event.

```
// Demonstrate adapter classes.
import java.awt.*;
import java.awt.event.*;

public class AdapterDemo extends Frame {
  String msg = "";

  public AdapterDemo() {
     addMouseListener(new MyMouseAdapter(this));
     addMouseMotionListener(new MyMouseAdapter(this));
     addWindowListener(new MyWindowAdapter());
  }

  // Display the mouse information.
  public void paint(Graphics g) {
    g.drawString(msg, 20, 80);
  }

  public static void main(String[] args) {
    AdapterDemo appwin = new AdapterDemo();

    appwin.setSize(new Dimension(200, 150));
    appwin.setTitle("AdapterDemo");
    appwin.setVisible(true);
  }
}
```

```
// Handle only mouse click and drag events.
class MyMouseAdapter extends MouseAdapter {
  AdapterDemo adapterDemo;
  public MyMouseAdapter(AdapterDemo adapterDemo) {
    this.adapterDemo = adapterDemo;
  }

  // Handle mouse clicked.
  public void mouseClicked(MouseEvent me) {
    adapterDemo.msg = "Mouse clicked";
    adapterDemo.repaint();
  }

  // Handle mouse dragged.
  public void mouseDragged(MouseEvent me) {
    adapterDemo.msg = "Mouse dragged";
    adapterDemo.repaint();
  }
}

// When the close box in the frame is clicked,
// close the window and exit the program.
class MyWindowAdapter extends WindowAdapter {
  public void windowClosing(WindowEvent we) {
    System.exit(0);
  }
}
```

As you can see by looking at the program, not having to implement all of the methods defined by the **MouseMotionListener**, **MouseListener**, and **MouseWheelListener** interfaces saves you a considerable amount of effort and prevents your code from becoming cluttered with empty methods. As an exercise, you might want to try rewriting one of the keyboard input examples shown earlier so that it uses a **KeyAdapter**.

Inner Classes

In Chapter 7, the basics of inner classes were explained. Here, you will see why they are important. Recall that an *inner class* is a class defined within another class, or even within an expression. This section illustrates how inner classes can

be used to simplify the code when using event adapter classes.

To understand the benefit provided by inner classes, consider the program shown in the following listing. It *does not* use an inner class. Its goal is to display the string "Mouse Pressed" when the mouse is pressed. Similar to the approach used by the preceding example, a reference to the **MousePressedDemo** instance is passed to the **MyMouseAdapter** constructor and saved. This reference is used to assign a string to **msg** and invoke **repaint( )** on the object that received the event.

```
// This program does NOT use an inner class.
import java.awt.*;
import java.awt.event.*;

public class MousePressedDemo extends Frame {
  String msg = "";

  public MousePressedDemo() {
    addMouseListener(new MyMouseAdapter(this));
    addWindowListener(new MyWindowAdapter());
  }

  public void paint(Graphics g) {
    g.drawString(msg, 20, 100);
  }

  public static void main(String[] args) {
    MousePressedDemo appwin = new MousePressedDemo();

    appwin.setSize(new Dimension(200, 150));
    appwin.setTitle("MousePressedDemo");
    appwin.setVisible(true);
  }
}

class MyMouseAdapter extends MouseAdapter {
  MousePressedDemo mousePressedDemo;

  public MyMouseAdapter(MousePressedDemo mousePressedDemo) {
    this.mousePressedDemo = mousePressedDemo;
  }

  // Handle a mouse pressed.
  public void mousePressed(MouseEvent me) {
    mousePressedDemo.msg = "Mouse Pressed.";
    mousePressedDemo.repaint();
  }
}

// When the close box in the frame is clicked,
// close the window and exit the program.
class MyWindowAdapter extends WindowAdapter {
  public void windowClosing(WindowEvent we) {
    System.exit(0);
  }
}
```

The following listing shows how the preceding program can be improved by using an inner class. Here, **InnerClassDemo** is the top-level class and **MyMouseAdapter** is an inner class. Because **MyMouseAdapter** is defined within the scope of **InnerClassDemo**, it has access to all of the variables and methods within the scope of that class. Therefore, the **mousePressed( )** method can call the **repaint( )** method directly. It no longer needs to do this via a stored reference. The same applies to assigning a value to **msg**. Thus, it is no longer necessary to pass **MyMouseAdapter( )** a reference to the invoking object. Also notice that **MyWindowAdapter** has been made into an inner class.

```
// Inner class demo.
import java.awt.*;
import java.awt.event.*;

public class InnerClassDemo extends Frame {
  String msg = "";

  public InnerClassDemo() {
    addMouseListener(new MyMouseAdapter());
    addWindowListener(new MyWindowAdapter());
  }

  // Inner class to handle mouse pressed events.
  class MyMouseAdapter extends MouseAdapter {
    public void mousePressed(MouseEvent me) {
      msg = "Mouse Pressed.";
      repaint();
    }
  }

  // Inner class to handle window close events.
  class MyWindowAdapter extends WindowAdapter {
    public void windowClosing(WindowEvent we) {
      System.exit(0);
    }
  }

  public void paint(Graphics g) {
    g.drawString(msg, 20, 80);
  }

  public static void main(String[] args) {
    InnerClassDemo appwin = new InnerClassDemo();

    appwin.setSize(new Dimension(200, 150));
    appwin.setTitle("InnerClassDemo");
    appwin.setVisible(true);
  }
}
```

Anonymous Inner Classes

An *anonymous* inner class is one that is not assigned a name. This section

illustrates how an anonymous inner class can facilitate the writing of event handlers. Consider the program shown in the following listing. As before, its goal is to display the string "Mouse Pressed" when the mouse is pressed.

```
// Anonymous inner class demo.
import java.awt.*;
import java.awt.event.*;

public class AnonymousInnerClassDemo extends Frame {
  String msg = "";

  public AnonymousInnerClassDemo() {

    // Anonymous inner class to handle mouse pressed events.
    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent me) {
        msg = "Mouse Pressed.";
        repaint();
      }
    });


    // Anonymous inner class to handle window close events.
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }

  public void paint(Graphics g) {
    g.drawString(msg, 20, 80);
  }

  public static void main(String[] args) {
    AnonymousInnerClassDemo appwin =
                  new AnonymousInnerClassDemo();

    appwin.setSize(new Dimension(200, 150));
    appwin.setTitle("AnonymousInnerClassDemo");
    appwin.setVisible(true);
  }
}
```

There is one top-level class in this program: **AnonymousInnerClassDemo**. Its constructor calls the **addMouseListener( )** method. Its argument is an expression that defines and instantiates an anonymous inner class. Let's analyze this expression carefully.

The syntax **new MouseAdapter(){...}** indicates to the compiler that the code between the braces defines an anonymous inner class. Furthermore, that class extends **MouseAdapter**. This new class is not named, but it is automatically instantiated when this expression is executed. This syntax can be generalized and is the same when creating other anonymous classes, such as when an anonymous **WindowAdapter** is created by the program.

Because this anonymous inner class is defined within the scope of **AnonymousInnerClassDemo**, it has access to all of the variables and methods within the scope of that class. Therefore, it can call the **repaint( )** method and access **msg** directly.

As just illustrated, both named and anonymous inner classes solve some annoying problems in a simple yet effective way. They also allow you to create more efficient code.

CHAPTER

Introducing the AWT: Working with Windows, Graphics, and Text

The Abstract Window Toolkit (AWT) was Java's first GUI framework, and it has been part of Java since version 1.0. It contains numerous classes and methods that allow you to create windows and simple controls. The AWT was introduced in Chapter 24, where it was used in several short examples that demonstrated event handling. This chapter begins a more detailed examination. Here, you will learn how to manage windows, work with fonts, output text, and utilize graphics. Chapter 26 describes various AWT controls, layout managers, and menus. It also explains further aspects of Java's event handling mechanism. Chapter 27 introduces the AWT's imaging subsystem.

It is important to state at the outset that you will seldom create GUIs based solely on the AWT because more powerful GUI frameworks (such as Swing, described later in this book) have been developed for Java. Despite this fact, the AWT remains an important part of Java. To understand why, consider the following.

At the time of this writing, the framework that is most widely used is Swing. Because Swing provides a richer, more flexible GUI framework than does the AWT, it is easy to jump to the conclusion that the AWT is no longer relevant—that it has been fully superseded by Swing. This assumption is, however, false. Instead, an understanding of the AWT is still important because the AWT underpins Swing, with many AWT classes being used either directly or indirectly by Swing. As a result, a solid knowledge of the AWT is still required to use Swing effectively. Also, for some types of small programs that make only minimal use of a GUI, using the AWT may still be appropriate. Therefore, even though the AWT constitutes Java's oldest GUI framework, a basic working knowledge of its fundamentals is still important today.

One last point before beginning: The AWT is quite large and a full description would easily fill an entire book. Therefore, it is not possible to describe in detail every AWT class, method, or instance variable. However, this and the following chapters explain the basic techniques needed to use the AWT. From there, you will be able to explore other parts of the AWT on your own. You will also be

ready to move on to Swing.

NOTE If you have not yet read Chapter 24, please do so now. It provides an overview of event handling, which is used by many of the examples in this chapter.

AWT Classes

The AWT classes are contained in the **java.awt** package. It is one of Java's largest packages. Fortunately, because it is logically organized in a top-down, hierarchical fashion, it is easier to understand and use than you might at first believe. Beginning with JDK 9, **java.awt** is part of the **java.desktop** module. Table 25-1 lists some of the many AWT classes.

Class	Description
AWTEvent	Encapsulates AWT events.
AWTEventMulticaster	Dispatches events to multiple listeners.
BorderLayout	The border layout manager. Border layouts use five components: North, South, East, West, and Center.
Button	Creates a push button control.
Canvas	A blank, semantics-free window.
CardLayout	The card layout manager. Card layouts emulate index cards. Only the one on top is showing.
Checkbox	Creates a check box control.
CheckboxGroup	Creates a group of check box controls.
CheckboxMenuItem	Creates an on/off menu item.
Choice	Creates a pop-up list.
Color	Manages colors in a portable, platform-independent fashion.
Component	An abstract superclass for various AWT components.
Container	A subclass of **Component** that can hold other components.
Cursor	Encapsulates a bitmapped cursor.
Dialog	Creates a dialog window.
Dimension	Specifies the dimensions of an object. The width is stored in **width**, and the height is stored in **height**.
EventQueue	Queues events.
FileDialog	Creates a window from which a file can be selected.

FlowLayout	The flow layout manager. Flow layout positions components left to right, top to bottom.
Font	Encapsulates a type font.
FontMetrics	Encapsulates various information related to a font. This information helps you display text in a window.
Frame	Creates a standard window that has a title bar, resize corners, and a menu bar.
Graphics	Encapsulates the graphics context. This context is used by the various output methods to display output in a window.
GraphicsDevice	Describes a graphics device such as a screen or printer.
GraphicsEnvironment	Describes the collection of available **Font** and **GraphicsDevice** objects.
GridBagConstraints	Defines various constraints relating to the **GridBagLayout** class.
GridBagLayout	The grid bag layout manager. Grid bag layout displays components subject to the constraints specified by **GridBagConstraints**.
GridLayout	The grid layout manager. Grid layout displays components in a two-dimensional grid.
Image	Encapsulates graphical images.
Insets	Encapsulates the borders of a container.
Label	Creates a label that displays a string.
List	Creates a list from which the user can choose. Similar to the standard Windows list box.
MediaTracker	Manages media objects.
Menu	Creates a pull-down menu.
MenuBar	Creates a menu bar.

MenuComponent	An abstract class implemented by various menu classes.
MenuItem	Creates a menu item.
MenuShortcut	Encapsulates a keyboard shortcut for a menu item.
Panel	The simplest concrete subclass of **Container**.
Point	Encapsulates a Cartesian coordinate pair, stored in **x** and **y**.
Polygon	Encapsulates a polygon.
PopupMenu	Encapsulates a pop-up menu.
PrintJob	An abstract class that represents a print job.
Rectangle	Encapsulates a rectangle.
Robot	Supports automated testing of AWT-based applications.
Scrollbar	Creates a scroll bar control.
ScrollPane	A container that provides horizontal and/or vertical scroll bars for another component.
SystemColor	Contains the colors of GUI widgets such as windows, scroll bars, text, and others.
TextArea	Creates a multiline edit control.
TextComponent	A superclass for **TextArea** and **TextField**.
TextField	Creates a single-line edit control.
Toolkit	Abstract class implemented by the AWT.
Window	Creates a window with no frame, no menu bar, and no title.

Table 25-1 A Sampling of AWT Classes

Although the basic structure of the AWT has been the same since Java 1.0, some of the original methods were deprecated and replaced by new ones. For backward-compatibility, Java still supports all the original 1.0 methods. However, because these methods are not for use with new code, this book does not describe them.

Window Fundamentals

The AWT defines windows according to a class hierarchy that adds functionality and specificity with each level. Arguably the two most important window-related classes are **Frame** and **Panel**. **Frame** encapsulates a top-level window and it is typically used to create what would be thought of as a standard application window. **Panel** provides a container to which other components can be added.

(**Panel** is also a superclass for **Applet**, which has been deprecated since JDK 9.) Much of the functionality of **Frame** and **Panel** is derived from their parent classes. Thus, a description of the class hierarchies relating to these two classes is fundamental to their understanding. Figure 25-1 shows the class hierarchy for **Panel** and **Frame**. Let's look at each of these classes now.

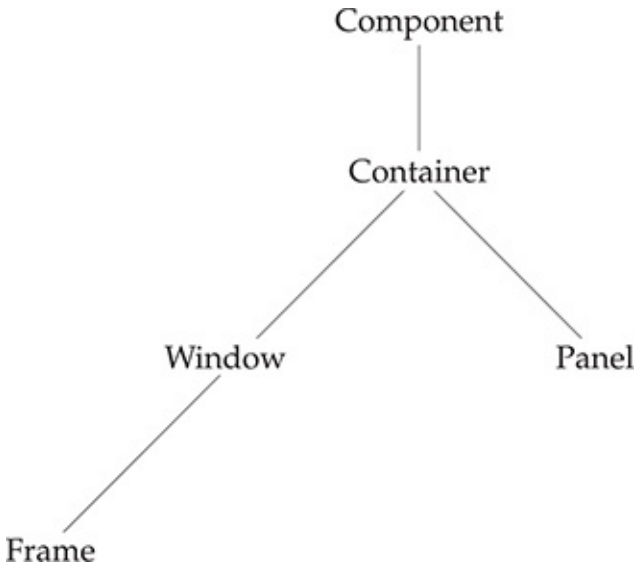


Figure 25-1 The class hierarchy for **Panel** and **Frame**

Component

At the top of the AWT hierarchy is the **Component** class. **Component** is an abstract class that encapsulates all of the attributes of a visual component. Except for menus, all user interface elements that are displayed on the screen and that interact with the user are subclasses of **Component**. It defines over a hundred public methods that are responsible for managing events, such as mouse and keyboard input, positioning and sizing the window, and repainting. A **Component** object is responsible for remembering the current foreground and background colors and the currently selected text font.

Container

The **Container** class is a subclass of **Component**. It has additional methods that allow other **Component** objects to be nested within it. Other **Container** objects can be stored inside of a **Container** (since they are themselves instances of **Component**). This makes for a multileveled containment system. A container is

responsible for laying out (that is, positioning) any components that it contains. It does this through the use of various layout managers, which you will learn about in Chapter 26.

Panel

The **Panel** class is a concrete subclass of **Container**. A **Panel** may be thought of as a recursively nestable, concrete screen component. Other components can be added to a **Panel** object by its **add( )** method (inherited from **Container**). Once these components have been added, you can position and resize them manually using the **setLocation( )**, **setSize( )**, **setPreferredSize( )**, or **setBounds( )** methods defined by **Component**.

Window

The **Window** class creates a top-level window. A *top-level window* is not contained within any other object; it sits directly on the desktop. Generally, you won't create **Window** objects directly. Instead, you will use a subclass of **Window** called **Frame**, described next.

Frame

Frame encapsulates what is commonly thought of as a "window." It is a subclass of **Window** and has a title bar, menu bar, borders, and resizing corners. The precise look of a **Frame** will differ among environments.

Canvas

Although it is not part of the hierarchy for **Panel** or **Frame**, there is one other class that you will find valuable: **Canvas**. Derived from **Component**, **Canvas** encapsulates a blank window upon which you can draw. You will see an example of **Canvas** later in this book.

Working with Frame Windows

Typically, the type of AWT-based application window you will most often create is derived from **Frame**. As mentioned, it creates a standard-style, top-level window that has all of the features normally associated with an application

window, such as a close box and title.

Here are two of **Frame**'s constructors:

Frame( ) throws HeadlessException
Frame(String *title*) throws HeadlessException

The first form creates a standard window that does not contain a title. The second form creates a window with the title specified by *title*. Notice that you cannot specify the dimensions of the window. Instead, you must set the size of the window after it has been created. A **HeadlessException** is thrown if an attempt is made to create a **Frame** instance in an environment that does not support user interaction.

There are several key methods you will use when working with **Frame** windows. They are examined here.

Setting the Window's Dimensions

The **setSize( )** method is used to set the dimensions of the window. It is shown here:

void setSize(int *newWidth*, int *newHeight*)
void setSize(Dimension *newSize*)

The new size of the window is specified by *newWidth* and *newHeight*, or by the **width** and **height** fields of the **Dimension** object passed in *newSize*. The dimensions are specified in terms of pixels.

The **getSize( )** method is used to obtain the current size of a window. One of its forms is shown here:

Dimension getSize( )

This method returns the current size of the window contained within the **width** and **height** fields of a **Dimension** object.

Hiding and Showing a Window

After a frame window has been created, it will not be visible until you call **setVisible( )**. Its signature is shown here:

void setVisible(boolean *visibleFlag*)

The component is visible if the argument to this method is **true**. Otherwise, it is hidden.

Setting a Window's Title

You can change the title in a frame window using **setTitle( )**, which has this general form:

void setTitle(String *newTitle*)

Here, *newTitle* is the new title for the window.

Closing a Frame Window

When using a frame window, your program must remove that window from the screen when it is closed. If it is not the top-level window of your application, this is done by calling **setVisible(false)**. For the main application window, you can simply terminate the program by calling **System.exit( )** as the examples in Chapter 24 did. To intercept a window-close event, you must implement the **windowClosing( )** method of the **WindowListener** interface. (See Chapter 24 for details on the **WindowListener** interface.)

The paint( ) Method

As you saw in Chapter 24, output to a window typically occurs when the **paint( )** method is called by the run-time system. This method is defined by **Component** and overridden by **Container** and **Window**. Thus, it is available to instances of **Frame**.

The **paint( )** method is called each time an AWT-based application's output must be redrawn. This situation can occur for several reasons. For example, the program's window may be overwritten by another window and then uncovered. Or the window may be minimized and then restored. **paint( )** is also called when the window is first displayed. Whatever the cause, whenever the window must redraw its output, **paint( )** is called. This implies that your program must have some way to retain its output so that it can be redisplayed each time **paint( )** executes.

The **paint( )** method is shown here:

void paint(Graphics *context*)

The **paint( )** method has one parameter of type **Graphics**. This parameter will contain the graphics context, which describes the graphics environment in which the program is running. This context is used whenever output to the window is required.

Displaying a String

To output a string to a **Frame**, use **drawString( )**, which is a member of the **Graphics** class. It was introduced in Chapter 24, and we will be making extensive use of it here and in the next chapter. This is the form we will use:

void drawString(String *message*, int *x*, int *y*)

Here, *message* is the string to be output beginning at *x*,*y*. In a Java window, the upper-left corner is location 0,0. The **drawString( )** method will not recognize newline characters. If you want to start a line of text on another line, you must do so manually, specifying the precise X,Y location where you want the line to begin. (As you will see in the next chapter, there are techniques that make this process easy.)

Setting the Foreground and Background Colors

You can set both the foreground and background colors used by a **Frame**. To set the background color, use **setBackground( )**. To set the foreground color (the color in which text is shown, for example), use **setForeground( )**. These methods are defined by **Component**, and they have the following general forms:

void setBackground(Color *newColor*)
void setForeground(Color *newColor*)

Here, *newColor* specifies the new color. The class **Color** defines the constants shown here that can be used to specify colors. (Uppercase versions of these constants are also provided.)

Color.black	Color.magenta
Color.blue	Color.orange
Color.cyan	Color.pink
Color.darkGray	Color.red
Color.gray	Color.white
Color.green	Color.yellow
Color.lightGray	

For example, the following sets the background color to green and the foreground color to red:

```
setBackground(Color.green);
setForeground(Color.red);
```

You can also create custom colors. A good place to initially set the foreground and background colors is in the constructor for the frame. Of course, you can change these colors as often as necessary during the execution of your program.

You can obtain the current settings for the background and foreground colors by calling **getBackground( )** and **getForeground( )**, respectively. They are also defined by **Component** and are shown here:

Color getBackground( )
Color getForeground( )

Requesting Repainting

As a general rule, an application writes to its window only when its **paint( )** method is called by the AWT. This raises an interesting question: How can the program itself cause its window to be updated to display new output? For example, if a program displays a moving banner, what mechanism does it use to update the window each time the banner scrolls? Remember, one of the fundamental architectural constraints imposed on a GUI program is that it must quickly return control to the run-time system. It cannot create a loop inside **paint( )** that repeatedly scrolls the banner, for example. This would prevent control from passing back to the AWT. Given this constraint, it may seem that output to your window will be difficult at best. Fortunately, this is not the case. Whenever your program needs to update the information displayed in its window, it simply calls **repaint( )**.

The **repaint( )** method is defined by **Component**. As it relates to **Frame**, this method causes the AWT run-time system to execute a call to the **update( )** method (also defined by **Component**). However, the default implementation of **update( )** calls **paint( )**. Thus, to output to a window, simply store the output and then call **repaint( )**. The AWT will then execute a call to **paint( )**, which can display the stored information. For example, if part of your program needs to output a string, it can store this string in a **String** variable and then call **repaint( )**. Inside **paint( )**, you will output the string using **drawString( )**.

The **repaint( )** method has four forms. Let's look at each one in turn. The simplest version of **repaint( )** is shown here:

```
void repaint( )
```

This version causes the entire window to be repainted. The following version specifies a region that will be repainted:

```
void repaint(int left, int top, int width, int height)
```

Here, the coordinates of the upper-left corner of the region are specified by *left* and *top*, and the width and height of the region are passed in *width* and *height.* These dimensions are specified in pixels. You save time by specifying a region to repaint. Window updates are costly in terms of time. If you need to update only a small portion of the window, it is more efficient to repaint only that region.

Calling **repaint( )** is essentially a request that a window be repainted sometime soon. However, if your system is slow or busy, **update( )** might not be called immediately. Multiple requests for repainting that occur within a short time can be collapsed by the AWT in a manner such that **update( )** is only called sporadically. This can be a problem in many situations, including animation, in which a consistent update time is necessary. One solution to this problem is to use the following forms of **repaint( )**:

```
void repaint(long maxDelay)
void repaint(long maxDelay, int x, int y, int width, int height)
```

Here, *maxDelay* specifies the maximum number of milliseconds that can elapse before **update( )** is called.

NOTE It is possible for a method other than **paint( )** or **update( )** to output to a window. To do so, it must obtain a graphics context by calling **getGraphics( )** (defined by **Component**) and then use this context to output to the window. However, for most applications, it is better and easier to route window output through **paint( )** and to call **repaint( )** when the contents of the window change.

Creating a Frame-Based Application

While it is possible to simply create a window by creating an instance of **Frame**, you will seldom do so, because you would not be able to do much with it. For example, you would not be able to receive or process events that occur within it or easily output information to it. Therefore, to create a **Frame**-based application, you will normally create a subclass of **Frame**. Among other reasons, doing so lets you override **paint( )** and provide event handling.

As the event handling examples in Chapter 24 illustrated, creating a new **Frame**-based application is actually quite easy. In general, you create a subclass of **Frame**, override **paint( )** to supply your output to the window, and implement the necessary event listeners. In all cases, you will need to implement the **windowClosing( )** method of the **WindowListener** interface. In a top-level frame, you will typically call **System.exit( )** to terminate the program. To simply remove a secondary frame from the screen, you can call **setVisible(false)** when the window is closed.

Once you have defined a **Frame** subclass, you can create an instance of that class. This causes a frame window to come into existence, but it will not be initially visible. You make it visible by calling **setVisible(true)**. When created, the window is given a default height and width. You can set the size of the window explicitly by calling the **setSize( )** method. For a top-level frame, you will want to define its title.

Introducing Graphics

The AWT includes several methods that support graphics. All graphics are drawn relative to a window. This can be the main window of an application or a child window. (These methods are also supported by Swing-based windows.) The origin of each window is at the top-left corner and is 0,0. Coordinates are specified in pixels. All output to a window takes place through a *graphics context*.

A graphics context is encapsulated by the **Graphics** class. Here are two ways in which a graphics context can be obtained:

- It is passed to a method, such as **paint( )** or **update( )**, as an argument.
- It is returned by the **getGraphics( )** method of **Component**.

Among other things, the **Graphics** class defines a number of methods that draw various types of objects, such as lines, rectangles, and arcs. In several cases,

objects can be drawn edge-only or filled. Objects are drawn and filled in the currently selected color, which is black by default. When a graphics object is drawn that exceeds the dimensions of the window, output is automatically clipped. A sampling of the drawing methods supported by **Graphics** is presented here.

NOTE A number of years ago, the graphics capabilities of Java were expanded by the inclusion of several new classes. One of these is **Graphics2D**, which extends **Graphics**. **Graphics2D** supports several enhancements to the basic capabilities provided by **Graphics**. To gain access to this extended functionality, you must cast the graphics context obtained from a method such as **paint( )**, to **Graphics2D**. Although the basic graphics functions supported by **Graphics** are adequate for the purposes of this book, **Graphics2D** is a class that you may want to explore fully on your own.

Drawing Lines

Lines are drawn by means of the **drawLine( )** method, shown here:

void drawLine(int *startX*, int *startY*, int *endX*, int *endY*)

drawLine( ) displays a line in the current drawing color that begins at *startX*, *startY* and ends at *endX*, *endY*.

Drawing Rectangles

The **drawRect( )** and **fillRect( )** methods display an outlined and filled rectangle, respectively. They are shown here:

void drawRect(int *left*, int *top*, int *width*, int *height*)
void fillRect(int *left*, int *top*, int *width*, int *height*)

The upper-left corner of the rectangle is at *left*, *top*. The dimensions of the rectangle are specified by *width* and *height*.

To draw a rounded rectangle, use **drawRoundRect( )** or **fillRoundRect( )**, both shown here:

void drawRoundRect(int *left*, int *top*, int *width*, int *height*,
int *xDiam*, int *yDiam*)
void fillRoundRect(int *left*, int *top*, int *width*, int *height*,
int *xDiam*, int *yDiam*)

A rounded rectangle has rounded corners. The upper-left corner of the rectangle is at *left, top*. The dimensions of the rectangle are specified by *width* and *height*.

The diameter of the rounding arc along the X axis is specified by *xDiam*. The diameter of the rounding arc along the Y axis is specified by *yDiam*.

Drawing Ellipses and Circles

To draw an ellipse, use **drawOval( )**. To fill an ellipse, use **fillOval( )**. These methods are shown here:

void drawOval(int *left*, int *top*, int *width*, int *height*)
void fillOval(int *left*, int *top*, int *width*, int *height*)

The ellipse is drawn within a bounding rectangle whose upper-left corner is specified by *left, top* and whose width and height are specified by *width* and *height*. To draw a circle, specify a square as the bounding rectangle.

Drawing Arcs

Arcs can be drawn with **drawArc( )** and **fillArc( )**, shown here:

void drawArc(int *left*, int *top*, int *width*, int *height*, int *startAngle*,
int *sweepAngle*)
void fillArc(int *left*, int *top*, int *width*, int *height*, int *startAngle*,
int *sweepAngle*)

The arc is bounded by the rectangle whose upper-left corner is specified by *left, top* and whose width and height are specified by *width* and *height*. The arc is drawn from *startAngle* through the angular distance specified by *sweepAngle*. Angles are specified in degrees. Zero degrees is on the horizontal, at the three o'clock position. The arc is drawn counterclockwise if *sweepAngle* is positive, and clockwise if *sweepAngle* is negative. Therefore, to draw an arc from twelve o'clock to six o'clock, the start angle would be 90 and the sweep angle 180.

Drawing Polygons

It is possible to draw arbitrarily shaped figures using **drawPolygon( )** and **fillPolygon( )**, shown here:

void drawPolygon(int *x*[ ], int *y*[ ], int *numPoints*)
void fillPolygon(int *x*[ ], int *y*[ ], int *numPoints*)

The polygon's endpoints are specified by the coordinate pairs contained within

the *x* and *y* arrays. The number of points defined by these arrays is specified by *numPoints*. There are alternative forms of these methods in which the polygon is specified by a **Polygon** object.

Demonstrating the Drawing Methods

The following program demonstrates the drawing methods just described.

```
// Draw graphics elements.
import java.awt.event.*;
import java.awt.*;

public class GraphicsDemo extends Frame {

  public GraphicsDemo() {
    // Anonymous inner class to handle window close events.
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }

  public void paint(Graphics g) {

    // Draw lines.
    g.drawLine(20, 40, 100, 90);
    g.drawLine(20, 90, 100, 40);
    g.drawLine(40, 45, 250, 80);

    // Draw rectangles.
    g.drawRect(20, 150, 60, 50);
    g.fillRect(110, 150, 60, 50);
    g.drawRoundRect(200, 150, 60, 50, 15, 15);
    g.fillRoundRect(290, 150, 60, 50, 30, 40);

    // Draw elipses and circles.
    g.drawOval(20, 250, 50, 50);
    g.fillOval(100, 250, 75, 50);
    g.drawOval(200, 260, 100, 40);

    // Draw arcs.
    g.drawArc(20, 350, 70, 70, 0, 180);
    g.fillArc(70, 350, 70, 70, 0, 75);

    // Draw a polygon.
    int xpoints[] = {20, 200, 20, 200, 20};
    int ypoints[] = {450, 450, 650, 650, 450};
    int num = 5;

    g.drawPolygon(xpoints, ypoints, num);
  }

  public static void main(String[] args) {
    GraphicsDemo appwin = new GraphicsDemo();

    appwin.setSize(new Dimension(370, 700));
    appwin.setTitle("GraphicsDemo");
    appwin.setVisible(true);
  }
}
```

Sample output is shown in Figure 25-1.

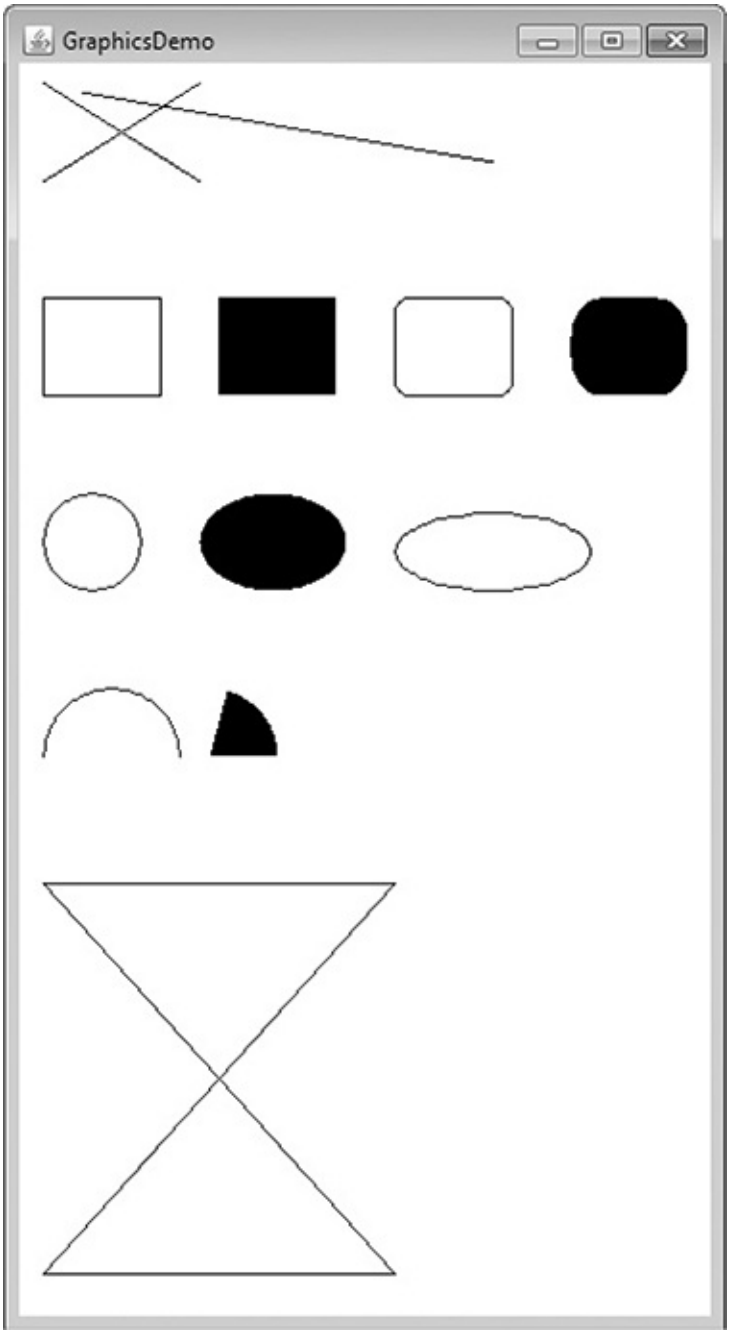


Figure 25-1 Sample output from the **GraphicsDemo** program

Sizing Graphics

Often, you will want to size a graphics object to fit the current size of the frame

in which it is drawn. To do so, first obtain the current dimensions of the frame by calling **getSize( )**. It returns the dimensions as integer values stored in the **width** and **height** fields of a **Dimension** instance. However, the dimensions returned by **getSize( )** reflect the overall size of the frame, including border and title bar. To obtain the dimensions of the paintable area, you will need to reduce the size obtained from **getSize( )** by the dimensions of the border/title bar. The values that describe the size of the border/title region are called *insets*. The inset values are obtained by calling **getInsets( )**, shown here:

```
Insets getInsets( )
```

It returns an **Insets** object that encapsulates the insets dimensions as four **int** values called **left**, **right**, **top**, and **bottom**. Therefore, the coordinate of the top-left corner of the paintable area is **left**, **top**. The coordinate of the bottom-right corner is **width−right**, **height−bottom**. Once you have both the size and insets, you can scale your graphical output accordingly.

To demonstrate this technique, here is a program whose frame starts with dimensions 200 × 200 pixels. It grows by 25 in both width and height each time the mouse is clicked until the size is larger than 500 × 500. At that point, the next click will return it to 200 × 200, and the process starts over. Within the paintable area, an X is drawn so that it fills the region.

```
// Resizing output to fit the current size of a window.
import java.awt.*;
import java.awt.event.*;

public class ResizeMe extends Frame {
  final int inc = 25;
  int max = 500;
  int min = 200;
  Dimension d;

  public ResizeMe() {
    // Anonymous inner class to handle mouse release events.
    addMouseListener(new MouseAdapter() {
      public void mouseReleased(MouseEvent me) {
        int w = (d.width + inc) > max?min :(d.width + inc);
        int h = (d.height + inc) > max?min :(d.height + inc);
        setSize(new Dimension(w, h));
      }
    });

    // Anonymous inner class to handle window close events.
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }

  public void paint(Graphics g) {
    Insets i = getInsets();
    d = getSize();

    g.drawLine(i.left, i.top, d.width-i.right, d.height-i.bottom);
    g.drawLine(i.left, d.height-i.bottom, d.width-i.right, i.top);
  }

  public static void main(String[] args) {
    ResizeMe appwin = new ResizeMe();

    appwin.setSize(new Dimension(200, 200));
    appwin.setTitle("ResizeMe");
    appwin.setVisible(true);
  }
}
```

Working with Color

Java supports color in a portable, device-independent fashion. The AWT color system allows you to specify any color you want. It then finds the best match for that color, given the limits of the display hardware currently executing your program. Thus, your code does not need to be concerned with the differences in the way color is supported by various hardware devices. Color is encapsulated by the **Color** class.

As you saw earlier, **Color** defines several constants (for example, **Color.black**) to specify a number of common colors. You can also create your own colors, using one of the **Color** constructors. Three commonly used forms are shown here:

Color(int *red*, int *green*, int *blue*)
Color(int *rgbValue*)
Color(float *red*, float *green*, float *blue*)

The first constructor takes three integers that specify the color as a mix of red, green, and blue. These values must be between 0 and 255, as in this example:

```
new Color(255, 100, 100); // light red
```

The second color constructor takes a single integer that contains the mix of red, green, and blue packed into an integer. The integer is organized with red in bits 16 to 23, green in bits 8 to 15, and blue in bits 0 to 7. Here is an example of this constructor:

```
int newRed = (0xff000000 | (0xc0 << 16) | (0x00 << 8) | 0x00);
Color darkRed = new Color(newRed);
```

The third constructor, **Color(float, float, float)**, takes three **float** values (between 0.0 and 1.0) that specify the relative mix of red, green, and blue.

Once you have created a color, you can use it to set the foreground and/or background color by using the **setForeground( )** and **setBackground( )** methods as mentioned earlier. You can also select it as the current drawing color.

Color Methods

The **Color** class defines several methods that help manipulate colors. Several are

examined here.

Using Hue, Saturation, and Brightness

The *hue-saturation-brightness (HSB)* color model is an alternative to red-green-blue (RGB) for specifying particular colors. Figuratively, *hue* is a wheel of color. The hue can be specified with a number between 0.0 and 1.0, which is used to obtain an angle into the color wheel. (The principal colors are approximately red, orange, yellow, green, blue, indigo, and violet.) *Saturation* is another scale ranging from 0.0 to 1.0, representing light pastels to intense hues. *Brightness* values also range from 0.0 to 1.0, where 1 is bright white and 0 is black. **Color** supplies two methods that let you convert between RGB and HSB. They are shown here:

static int HSBtoRGB(float *hue*, float *saturation*, float *brightness*)
static float[ ] RGBtoHSB(int *red*, int *green*, int *blue*, float *values*[ ])

HSBtoRGB( ) returns a packed RGB value compatible with the **Color(int)** constructor. **RGBtoHSB( )** returns a **float** array of HSB values corresponding to RGB integers. If *values* is not **null**, then this array is given the HSB values and returned. Otherwise, a new array is created and the HSB values are returned in it. In either case, the array contains the hue at index 0, saturation at index 1, and brightness at index 2.

getRed( ), getGreen( ), getBlue( )

You can obtain the red, green, and blue components of a color independently using **getRed( )**, **getGreen( )**, and **getBlue( )**, shown here:

int getRed( )
int getGreen( )
int getBlue( )

Each of these methods returns the RGB color component found in the invoking **Color** object in the lower 8 bits of an integer.

getRGB( )

To obtain a packed, RGB representation of a color, use **getRGB( )**, shown here:

int getRGB( )

The return value is organized as described earlier.

Setting the Current Graphics Color

By default, graphics objects are drawn in the current foreground color. You can change this color by calling the **Graphics** method **setColor( )** :

void setColor(Color *newColor*)

Here, *newColor* specifies the new drawing color.

You can obtain the current color by calling **getColor( )**, shown here:

Color getColor( )

A Color Demonstration Program

The following program constructs several colors and draws various objects using these colors:

```
// Demonstrate color.
import java.awt.*;
import java.awt.event.*;

public class ColorDemo extends Frame {

  public ColorDemo() {
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }

  // Draw in different colors.
  public void paint(Graphics g) {
    Color c1 = new Color(255, 100, 100);
    Color c2 = new Color(100, 255, 100);
    Color c3 = new Color(100, 100, 255);

    g.setColor(c1);
    g.drawLine(20, 40, 100, 100);
    g.drawLine(20, 100, 100, 20);

    g.setColor(c2);
    g.drawLine(40, 45, 250, 180);
    g.drawLine(75, 90, 400, 400);
```

```
    g.setColor(c3);
    g.drawLine(20, 150, 400, 40);
    g.drawLine(25, 290, 80, 19);

    g.setColor(Color.red);
    g.drawOval(20, 40, 50, 50);
    g.fillOval(70, 90, 140, 100);

    g.setColor(Color.blue);
    g.drawOval(190, 40, 90, 60);
    g.drawRect(40, 40, 55, 50);

    g.setColor(Color.cyan);
    g.fillRect(100, 40, 60, 70);
    g.drawRoundRect(190, 40, 60, 60, 15, 15);
  }

  public static void main(String[] args) {
    ColorDemo appwin = new ColorDemo();

    appwin.setSize(new Dimension(340, 260));
    appwin.setTitle("ColorDemo");
    appwin.setVisible(true);
  }
}
```

Setting the Paint Mode

The *paint mode* determines how objects are drawn in a window. By default, new output to a window overwrites any preexisting contents. However, it is possible to have new objects XORed onto the window by using **setXORMode( )**, as follows:

void setXORMode(Color *xorColor*)

Here, *xorColor* specifies the color that will be XORed to the window when an object is drawn. The advantage of XOR mode is that the new object is always guaranteed to be visible no matter what color the object is drawn over.

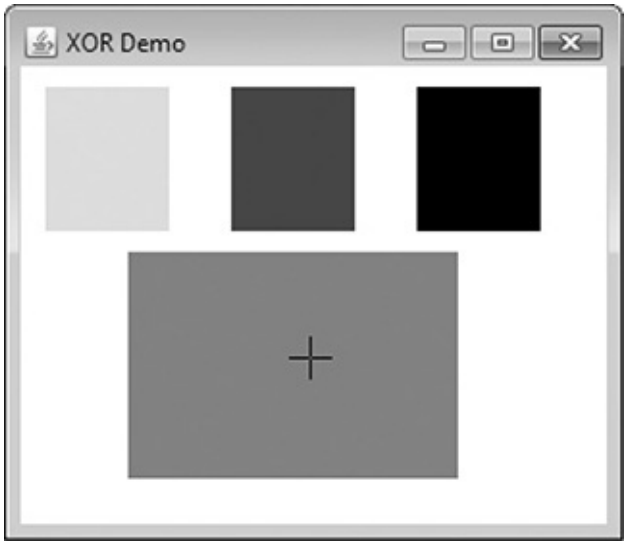
To return to overwrite mode, call **setPaintMode( )**, shown here:

void setPaintMode( )

In general, you will want to use overwrite mode for normal output, and XOR mode for special purposes. For example, the following program displays cross hairs that track the mouse pointer. The cross hairs are XORed onto the window and are always visible, no matter what the underlying color is.

```
// Demonstrate XOR mode.
import java.awt.*;
import java.awt.event.*;

public class XOR extends Frame {
  int chsX=100, chsY=100;

  public XOR() {
    addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseMoved(MouseEvent me) {
        int x = me.getX();
        int y = me.getY();
        chsX = x-10;
        chsY = y-10;
        repaint();
      }
    });

    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }

  // Demonstrate XOR mode.
  public void paint(Graphics g) {
    g.setColor(Color.green);
    g.fillRect(20, 40, 60, 70);

    g.setColor(Color.blue);
    g.fillRect(110, 40, 60, 70);

    g.setColor(Color.black);
    g.fillRect(200, 40, 60, 70);

    g.setColor(Color.red);
    g.fillRect(60, 120, 160, 110);

    // XOR cross hairs
    g.setXORMode(Color.black);
    g.drawLine(chsX-10, chsY, chsX+10, chsY);
    g.drawLine(chsX, chsY-10, chsX, chsY+10);
    g.setPaintMode();
  }

  public static void main(String[] args) {
    XOR appwin = new XOR();

    appwin.setSize(new Dimension(300, 260));
    appwin.setTitle("XOR Demo");
    appwin.setVisible(true);
  }

}
```

Sample output from this program is shown here:



Working with Fonts

The AWT supports multiple type fonts. Years ago, fonts emerged from the domain of traditional typesetting to become an important part of computer-generated documents and displays. The AWT provides flexibility by abstracting font-manipulation operations and allowing for dynamic selection of fonts.

Fonts have a family name, a logical font name, and a face name. The *family name* is the general name of the font, such as Courier. The *logical name* specifies a name, such as Monospaced, that is linked to an actual font at runtime. The *face name* specifies a specific font, such as Courier Italic.

Fonts are encapsulated by the **Font** class. Several of the methods defined by **Font** are listed in Table 25-2.

Method	Description
static Font decode(String *str*)	Returns a font given its name.
boolean equals(Object *FontObj*)	Returns **true** if the invoking object contains the same font as that specified by *FontObj*. Otherwise, it returns **false**.
String getFamily( )	Returns the name of the font family to which the invoking font belongs.
static Font getFont(String *property*)	Returns the font associated with the system property specified by *property*. **null** is returned if *property* does not exist.
static Font getFont(String *property*, Font *defaultFont*)	Returns the font associated with the system property specified by *property*. The font specified by *defaultFont* is returned if *property* does not exist.
String getFontName()	Returns the face name of the invoking font.
String getName( )	Returns the logical name of the invoking font.
int getSize( )	Returns the size, in points, of the invoking font.
int getStyle( )	Returns the style values of the invoking font.
int hashCode( )	Returns the hash code associated with the invoking object.
boolean isBold( )	Returns **true** if the font includes the **BOLD** style value. Otherwise, **false** is returned.
boolean isItalic( )	Returns **true** if the font includes the **ITALIC** style value. Otherwise, **false** is returned.
boolean isPlain( )	Returns **true** if the font includes the **PLAIN** style value. Otherwise, **false** is returned.
String toString( )	Returns the string equivalent of the invoking font.

Table 25-2 A Sampling of Methods Defined by **Font**

The **Font** class defines these protected variables:

Variable	Meaning
String name	Name of the font
float pointSize	Size of the font in points
int size	Size of the font in points
int style	Font style

Several static fields are also defined.

Determining the Available Fonts

When working with fonts, often you need to know which fonts are available on your machine. To obtain this information, you can use the **getAvailableFontFamilyNames( )** method defined by the **GraphicsEnvironment** class. It is shown here:

```
String[ ] getAvailableFontFamilyNames( )
```

This method returns an array of strings that contains the names of the available font families.

In addition, the **getAllFonts( )** method is defined by the **GraphicsEnvironment** class. It is shown here:

```
Font[ ] getAllFonts( )
```

This method returns an array of **Font** objects for all of the available fonts.

Since these methods are members of **GraphicsEnvironment**, you need a **GraphicsEnvironment** reference to call them. You can obtain this reference by using the **getLocalGraphicsEnvironment( )** static method, which is defined by **GraphicsEnvironment**. It is shown here:

```
static GraphicsEnvironment getLocalGraphicsEnvironment( )
```

Here is a program that shows how to obtain the names of the available font families:

```
// Display Fonts.
import java.awt.event.*;
import java.awt.*;

public class ShowFonts extends Frame {
  String msg = "First five fonts: ";
  GraphicsEnvironment ge;

  public ShowFonts() {
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });

    // Get the graphics environment.
    ge = GraphicsEnvironment.getLocalGraphicsEnvironment();

    // Obtain a list of the fonts.
    String[] fontList = ge.getAvailableFontFamilyNames();

    // Create a string of the first 5 fonts.
    for(int i=0; (i < 5) && (i < fontList.length); i++)
      msg += fontList[i] + " ";
  }

  // Display the fonts.
  public void paint(Graphics g) {
    g.drawString(msg, 10, 60);
  }
  public static void main(String[] args) {
    ShowFonts appwin = new ShowFonts();

    appwin.setSize(new Dimension(500, 100));
    appwin.setTitle("ShowFonts");
    appwin.setVisible(true);
  }
}
```

Sample output from this program is shown next. However, when you run this program, you may see a different set of fonts than the one shown in this illustration.

Creating and Selecting a Font

To create a new font, construct a **Font** object that describes that font. One **Font** constructor has this general form:

Font(String *fontName*, int *fontStyle*, int *pointSize*)

Here, *fontName* specifies the name of the desired font. The name can be specified using either the family or face name. All Java environments will support the following fonts: Dialog, DialogInput, SansSerif, Serif, and Monospaced. Dialog is the font used by your system's dialog boxes. Dialog is also the default if you don't explicitly set a font. You can also use any other fonts supported by your particular environment, but be careful—these other fonts may not be universally available.

The style of the font is specified by *fontStyle*. It may consist of one or more of these three constants: **Font.PLAIN**, **Font.BOLD**, and **Font.ITALIC**. To combine styles, OR them together. For example, **Font.BOLD | Font.ITALIC** specifies a bold, italics style.

The size, in points, of the font is specified by *pointSize*.

To use a font that you have created, you must select it using **setFont( )**, which is defined by **Component**. It has this general form:

void setFont(Font *fontObj*)

Here, *fontObj* is the object that contains the desired font.

The following program outputs a sample of each standard font. Each time you click the mouse within its window, a new font is selected and its name is displayed.

```
// Display fonts.
import java.awt.*;
import java.awt.event.*;

public class SampleFonts extends Frame {
  int next = 0;
  Font f;
  String msg;

  public SampleFonts() {
    f = new Font("Dialog", Font.PLAIN, 12);
    msg = "Dialog";
    setFont(f);

    addMouseListener(new MyMouseAdapter(this));

    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }

  public void paint(Graphics g) {
    g.drawString(msg, 10, 60);
  }

  public static void main(String[] args) {
    SampleFonts appwin = new SampleFonts();

    appwin.setSize(new Dimension(200, 100));
    appwin.setTitle("SampleFonts");
    appwin.setVisible(true);
  }
}
```

```
class MyMouseAdapter extends MouseAdapter {
  SampleFonts sampleFonts;

  public MyMouseAdapter(SampleFonts sampleFonts) {
    this.sampleFonts = sampleFonts;
  }

  public void mousePressed(MouseEvent me) {
    // Switch fonts with each mouse click.
    sampleFonts.next++;

    switch(sampleFonts.next) {
      case 0:
        sampleFonts.f = new Font("Dialog", Font.PLAIN, 12);
        sampleFonts.msg = "Dialog";
        break;
      case 1:
        sampleFonts.f = new Font("DialogInput", Font.PLAIN, 12);
        sampleFonts.msg = "DialogInput";
        break;
      case 2:
        sampleFonts.f = new Font("SansSerif", Font.PLAIN, 12);
        sampleFonts.msg = "SansSerif";
        break;
      case 3:
        sampleFonts.f = new Font("Serif", Font.PLAIN, 12);
        sampleFonts.msg = "Serif";
        break;
      case 4:
        sampleFonts.f = new Font("Monospaced", Font.PLAIN, 12);
        sampleFonts.msg = "Monospaced";
        break;
    }

    if(sampleFonts.next == 4) sampleFonts.next = -1;
    sampleFonts.setFont(sampleFonts.f);
    sampleFonts.repaint();
  }
}
```

Sample output from this program is shown here:

Obtaining Font Information

Suppose you want to obtain information about the currently selected font. To do this, you must first get the current font by calling **getFont( )**. This method is defined by the **Graphics** class, as shown here:

```
Font getFont( )
```

Once you have obtained the currently selected font, you can retrieve information about it using various methods defined by **Font**. For example, this program displays the name, family, size, and style of the currently selected font:

```
// Display font info.
import java.awt.event.*;
import java.awt.*;

public class FontInfo extends Frame {

  public FontInfo() {
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }

  public void paint(Graphics g) {
    Font f = g.getFont();
    String fontName = f.getName();
    String fontFamily = f.getFamily();
    int fontSize = f.getSize();
    int fontStyle = f.getStyle();
    String msg = "Family: " + fontName;

    msg += ", Font: " + fontFamily;
    msg += ", Size: " + fontSize + ", Style: ";

    if((fontStyle & Font.BOLD) == Font.BOLD)
      msg += "Bold ";
    if((fontStyle & Font.ITALIC) == Font.ITALIC)
      msg += "Italic ";
    if((fontStyle & Font.PLAIN) == Font.PLAIN)
      msg += "Plain ";

    g.drawString(msg, 10, 60);
  }

  public static void main(String[] args) {
    FontInfo appwin = new FontInfo();

    appwin.setSize(new Dimension(300, 100));
    appwin.setTitle("FontInfo");
    appwin.setVisible(true);
  }
}
```

Managing Text Output Using FontMetrics

As just explained, Java supports a number of fonts. For most fonts, characters are not all the same dimension—most fonts are proportional. Also, the height of each character, the length of *descenders* (the hanging parts of letters, such as *y*), and the amount of space between horizontal lines vary from font to font. Further, the point size of a font can be changed. That these (and other) attributes are variable would not be of too much consequence except that Java demands that you, the programmer, manually manage virtually all text output.

Given that the size of each font may differ and that fonts may be changed while your program is executing, there must be some way to determine the dimensions and various other attributes of the currently selected font. For example, to write one line of text after another implies that you have some way of knowing how tall the font is and how many pixels are needed between lines. To fill this need, the AWT includes the **FontMetrics** class, which encapsulates various information about a font. Let's begin by defining the common terminology used when describing fonts:

Height	The top-to-bottom size of a line of text
Baseline	The line that the bottoms of characters are aligned to (not counting descent)
Ascent	The distance from the baseline to the top of a character
Descent	The distance from the baseline to the bottom of a character
Leading	The distance between the bottom of one line of text and the top of the next

As you know, we have used the **drawString( )** method in many of the previous examples. It paints a string in the current font and color, beginning at a specified location. However, this location is at the left edge of the baseline of the characters, not at the upper-left corner as is usual with other drawing methods. It is a common error to draw a string at the same coordinate that you would draw a box. For example, if you were to draw a rectangle at the top, left location, you would see a full rectangle. If you were to draw the string "Typesetting" at this location, you would only see the tails (or descenders) of the *y*, *p*, and *g*. As you will see, by using font metrics, you can determine the proper placement of each string that you display.

FontMetrics defines several methods that help you manage text output. A number of commonly used ones are listed in Table 25-3. These methods help you properly display text in a window.

Method	Description
int bytesWidth(byte *b*[ ], int *start*, int *numBytes*)	Returns the width of *numBytes* characters held in array *b*, beginning at *start*.
int charsWidth(char *c*[ ], int *start*, int *numChars*)	Returns the width of *numChars* characters held in array *c*, beginning at *start*.
int charWidth(char *c*)	Returns the width of *c*.
int charWidth(int *c*)	Returns the width of *c*.
int getAscent( )	Returns the ascent of the font.
int getDescent( )	Returns the descent of the font.
Font getFont( )	Returns the font.
int getHeight( )	Returns the height of a line of text. This value can be used to output multiple lines of text in a window.
int getLeading( )	Returns the space between lines of text.
int getMaxAdvance( )	Returns the width of the widest character. –1 is returned if this value is not available.
int getMaxAscent( )	Returns the maximum ascent.
int getMaxDescent( )	Returns the maximum descent.
int[ ] getWidths( )	Returns the widths of the first 256 characters.
int stringWidth(String *str*)	Returns the width of the string specified by *str*.
String toString( )	Returns the string equivalent of the invoking object.

Table 25-3 A Sampling of Methods Defined by **FontMetrics**

Perhaps the most common use of **FontMetrics** is to determine the spacing between lines of text. The second most common use is to determine the length of a string that is being displayed. Here, you will see how to accomplish these tasks.

In general, to display multiple lines of text, your program must manually keep track of the current output position. Each time a newline is desired, the Y coordinate must be advanced to the beginning of the next line. Each time a string is displayed, the X coordinate must be set to the point at which the string ends. This allows the next string to be written so that it begins at the end of the preceding one.

To determine the spacing between lines, you can use the value returned by **getLeading( )**. To determine the total height of the font, add the value returned by **getAscent( )** to the value returned by **getDescent( )**. You can then use these

values to position each line of text you output. However, in many cases, you will not need to use these individual values. Often, all that you will need to know is the total height of a line, which is the sum of the leading space and the font's ascent and descent values. The easiest way to obtain this value is to call **getHeight( )**. In many cases, you can simply increment the Y coordinate by this value each time you want to advance to the next line when outputting text.

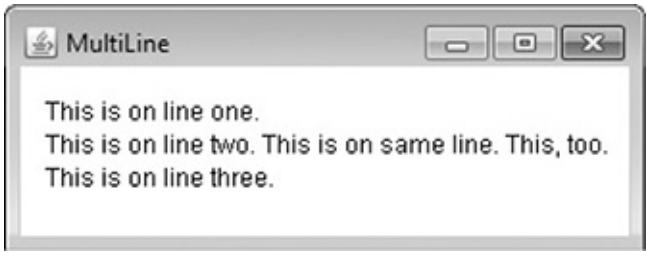
To start output at the end of previous output on the same line, you must know the length, in pixels, of each string that you display. To obtain this value, call **stringWidth( )**. You can use this value to advance the X coordinate each time you display a line.

The following program shows how to output multiple lines of text in a window. It also displays multiple sentences on the same line. Notice the variables **curX** and **curY**. They keep track of the current text output position.

```
// Demonstrate multiline output.
import java.awt.event.*;
import java.awt.*;

public class MultiLine extends Frame {
  int curX=20, curY=40; // current position

  public MultiLine() {
    Font f = new Font("SansSerif", Font.PLAIN, 12);
    setFont(f);

    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }

  public void paint(Graphics g) {
    FontMetrics fm = g.getFontMetrics();

    nextLine("This is on line one.", g);
    nextLine("This is on line two.", g);
    sameLine(" This is on same line.", g);
    sameLine(" This, too.", g);
    nextLine("This is on line three.", g);

    curX = 20; curY = 40; // reset the coordinates for each repaint
  }
```

```
  // Advance to next line.
  void nextLine(String s, Graphics g) {
    FontMetrics fm = g.getFontMetrics();

    curY += fm.getHeight(); // advance to next line
    curX = 20;
    g.drawString(s, curX, curY);
    curX += fm.stringWidth(s); // advance to end of line
  }

  // Display on same line.
  void sameLine(String s, Graphics g) {
    FontMetrics fm = g.getFontMetrics();

    g.drawString(s, curX, curY);
    curX += fm.stringWidth(s); // advance to end of line
  }

  public static void main(String[] args) {
    MultiLine appwin = new MultiLine();

    appwin.setSize(new Dimension(300, 120));
    appwin.setTitle("MultiLine");
    appwin.setVisible(true);
  }
}
```

Sample output from this program is shown here:



CHAPTER

Using AWT Controls, Layout Managers, and Menus

This chapter continues our overview of the Abstract Window Toolkit (AWT). It begins with a look at several of the AWT's controls and layout managers. It then discusses menus and the menu bar. The chapter also includes a discussion of the dialog box.

Controls are components that allow a user to interact with your application in various ways—for example, a commonly used control is the push button. A *layout manager* automatically positions components within a container. Thus, the appearance of a window is determined by a combination of the controls that it contains and the layout manager used to position them.

In addition to the controls, a frame window can also include a standard-style *menu bar*. Each entry in a menu bar activates a drop-down menu of options from which the user can choose. This constitutes the *main menu* of an application. As a general rule, a menu bar is positioned at the top of a window. Although different in appearance, menu bars are handled in much the same way as are the other controls.

While it is possible to manually position components within a window, doing so is quite tedious. The layout manager automates this task. For the first part of this chapter, which introduces various controls, a simple layout manager will be used that displays components in a container using line-by-line, top-to-bottom organization. Once the controls have been covered, several other layout managers will be examined. There, you will see ways to better manage the positioning of controls.

Before continuing, it is important to emphasize that today you will seldom create GUIs based solely on the AWT because more powerful GUI frameworks have been developed for Java. However, the material presented here remains important for the following reasons. First, much of the information and many of the techniques related to controls and event handling are generalizable to Swing. (As mentioned in the previous chapter, Swing is built upon the AWT.) Second, the layout managers described here are also used by Swing. Third, for some small applications, the AWT components might be the appropriate choice.

Finally, you may encounter legacy code that uses the AWT. Therefore, a basic understanding of the AWT is important for all Java programmers.

AWT Control Fundamentals

The AWT supports the following types of controls:

- Labels
- Push buttons
- Check boxes
- Choice lists
- Lists
- Scroll bars
- Text Editing

All AWT controls are subclasses of **Component**. Although the set of controls provided by the AWT is not particularly rich, it is sufficient for simple applications, such as short utility programs intended for your own use. It is also quite useful for introducing the basic concepts and techniques related to handling events in controls. It is important to point out, however, that Swing provides a substantially larger, more sophisticated set of controls better suited for creating commercial applications.

Adding and Removing Controls

To include a control in a window, you must add it to the window. To do this, you must first create an instance of the desired control and then add it to a window by calling **add( )**, which is defined by **Container**. The **add( )** method has several forms. The following form is the one that is used for the first part of this chapter:

Component add(Component *compRef*)

Here, *compRef* is a reference to an instance of the control that you want to add. A reference to the object is returned. Once a control has been added, it will automatically be visible whenever its parent window is displayed.

Sometimes you will want to remove a control from a window when the control is no longer needed. To do this, call **remove( )**. This method is also defined by **Container**. Here is one of its forms:

void remove(Component *compRef*)

Here, *compRef* is a reference to the control you want to remove. You can remove all controls by calling **removeAll( )**.

Responding to Controls

Except for labels, which are passive, all other controls generate events when they are accessed by the user. For example, when the user clicks on a push button, an event is sent that identifies the push button. In general, your program simply implements the appropriate interface and then registers an event listener for each control that you need to monitor. As explained in Chapter 24, once a listener has been installed, events are automatically sent to it. In the sections that follow, the appropriate interface for each control is specified.

The HeadlessException

Most of the AWT controls described in this chapter have constructors that can throw a **HeadlessException** when an attempt is made to instantiate a GUI component in a non-interactive environment (such as one in which no display, mouse, or keyboard is present). You can use this exception to write code that can adapt to non-interactive environments. (Of course, this is not always possible.) This exception is not handled by the programs in this chapter because an interactive environment is required to demonstrate the AWT controls.

Labels

The easiest control to use is a label. A *label* is an object of type **Label**, and it contains a string, which it displays. Labels are passive controls that do not support any interaction with the user. **Label** defines the following constructors:

Label( ) throws HeadlessException
Label(String *str*) throws HeadlessException
Label(String *str*, int *how*) throws HeadlessException

The first version creates a blank label. The second version creates a label that contains the string specified by *str*. This string is left-justified. The third version creates a label that contains the string specified by *str* using the alignment specified by *how*. The value of *how* must be one of these three constants:

Label.LEFT, **Label.RIGHT**, or **Label.CENTER**.

You can set or change the text in a label by using the **setText( )** method. You can obtain the current label by calling **getText( )**. These methods are shown here:

```
void setText(String str)
String getText( )
```

For **setText( )**, *str* specifies the new label. For **getText( )**, the current label is returned.

You can set the alignment of the string within the label by calling **setAlignment( )**. To obtain the current alignment, call **getAlignment( )**. The methods are as follows:

```
void setAlignment(int how)
int getAlignment( )
```

Here, *how* must be one of the alignment constants shown earlier.

The following example creates three labels and adds them to a **Frame**.

```
// Demonstrate Labels.
import java.awt.*;
import java.awt.event.*;

public class LabelDemo extends Frame {
  public LabelDemo() {

    // Use a flow layout.
    setLayout(new FlowLayout());

    Label one = new Label("One");
    Label two = new Label("Two");
    Label three = new Label("Three");

    // Add labels to frame.
    add(one);
    add(two);
    add(three);

    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }

  public static void main(String[] args) {
    LabelDemo appwin = new LabelDemo();

    appwin.setSize(new Dimension(300, 100));
    appwin.setTitle("LabelDemo");
    appwin.setVisible(true);
  }
}
```

Here is sample output from the **LabelDemo** program.

Notice that the labels are organized in the window left-to-right. This is handled automatically by the **FlowLayout** layout manager, which is one of the layout managers provided by the AWT. Here it is used in its default configuration, which lays out components line-by-line, left-to-right, top-to-bottom, and centered. As you will see later in this chapter, it supports several options, but for now, its default behavior is sufficient. Notice that **FlowLayout** is selected as the layout manager by use of **setLayout( )**. This method sets the layout manager associated with the container, which in this case is the enclosing frame. Although **FlowLayout** is very convenient and sufficient for our purposes at this time, it does not give you detailed control over the placement of components within a window. Later in this chapter, when the topic of layout managers is examined in detail, you will see how to gain more control over the organization of your windows.

Using Buttons

Perhaps the most widely used control is the push button. A *push button* is a component that contains a label and generates an event when it is pressed. Push buttons are objects of type **Button**. **Button** defines these two constructors:

Button( ) throws HeadlessException
Button(String *str*) throws HeadlessException

The first version creates an empty button. The second creates a button that contains *str* as a label.

After a button has been created, you can set its label by calling **setLabel( )**. You can retrieve its label by calling **getLabel( )**. These methods are as follows:

void setLabel(String *str*)
String getLabel( )

Here, *str* becomes the new label for the button.

Handling Buttons

Each time a button is pressed, an action event is generated. This is sent to any listeners that previously registered an interest in receiving action event notifications from that component. Each listener implements the **ActionListener** interface. That interface defines the **actionPerformed( )** method, which is called

when an event occurs. An **ActionEvent** object is supplied as the argument to this method. It contains both a reference to the button that generated the event and a reference to the *action command string* associated with the button. By default, the action command string is the label of the button. Either the button reference or the action command string can be used to identify the button. (You will soon see examples of each approach.)

Here is an example that creates three buttons labeled "Yes", "No", and "Undecided". Each time one is pressed, a message is displayed that reports which button has been pressed. In this version, the action command of the button (which, by default, is its label) is used to determine which button has been pressed. The label is obtained by calling the **getActionCommand( )** method on the **ActionEvent** object passed to **actionPerformed( )**.

```
// Demonstrate Buttons.
import java.awt.*;
import java.awt.event.*;

public class ButtonDemo extends Frame implements ActionListener {
  String msg = "";
  Button yes, no, maybe;

  public ButtonDemo() {

    // Use a flow layout.
    setLayout(new FlowLayout());

    // Create some buttons.
    yes = new Button("Yes");
    no = new Button("No");
    maybe = new Button("Undecided");

    // Add them to the frame.
    add(yes);
    add(no);
    add(maybe);

    // Add action listeners for the buttons.
    yes.addActionListener(this);
    no.addActionListener(this);
    maybe.addActionListener(this);

    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }
```

```
  // Handle button action events.
  public void actionPerformed(ActionEvent ae) {
    String str = ae.getActionCommand();
    if(str.equals("Yes")) {
      msg = "You pressed Yes.";
    }
    else if(str.equals("No")) {
      msg = "You pressed No.";
    }
    else {
      msg = "You pressed Undecided.";
    }

    repaint();
  }

  public void paint(Graphics g) {
     g.drawString(msg, 20, 100);
  }

  public static void main(String[] args) {
    ButtonDemo appwin = new ButtonDemo();

    appwin.setSize(new Dimension(250, 150));
    appwin.setTitle("ButtonDemo");
    appwin.setVisible(true);
  }
}
```

Sample output from the **ButtonDemo** program is shown in Figure 26-1.

Figure 26-1 Sample output from the **ButtonDemo** program

As mentioned, in addition to comparing button action command strings, you

can also determine which button has been pressed by comparing the object obtained from the **getSource( )** method to the button objects that you added to the window. To do this, you must keep a list of the objects when they are added. The following program shows this approach:

```
// Recognize Button objects.
import java.awt.*;
import java.awt.event.*;

public class ButtonList extends Frame implements ActionListener {
  String msg = "";
  Button bList[] = new Button[3];

  public ButtonList() {

    // Use a flow layout.
    setLayout(new FlowLayout());

    // Create some buttons.
    Button yes = new Button("Yes");
    Button no = new Button("No");
    Button maybe = new Button("Undecided");

    // Store references to buttons as added.
    bList[0] = (Button) add(yes);
    bList[1] = (Button) add(no);
    bList[2] = (Button) add(maybe);

    // Register to receive action events.
    for(int i = 0; i < 3; i++) {
      bList[i].addActionListener(this);
    }
```

```
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }

  // Handle button action events.
  public void actionPerformed(ActionEvent ae) {
    for(int i = 0; i < 3; i++) {
      if(ae.getSource() == bList[i]) {
        msg = "You pressed " + bList[i].getLabel();
      }
    }
    repaint();
  }

  public void paint(Graphics g) {
     g.drawString(msg, 20, 100);
  }

  public static void main(String[] args) {
    ButtonList appwin = new ButtonList();

    appwin.setSize(new Dimension(250, 150));
    appwin.setTitle("ButtonList");
    appwin.setVisible(true);
  }
}
```

In this version, the program stores each button reference in an array when the buttons are added to the frame. (Recall that the **add( )** method returns a reference to the button when it is added.) Inside **actionPerformed( )**, this array is then used to determine which button has been pressed.

For simple programs, it is usually easier to recognize buttons by their labels. However, in situations in which you will be changing the label inside a button during the execution of your program, or using buttons that have the same label, it may be easier to determine which button has been pushed by using its object reference. It is also possible to set the action command string associated with a button to something other than its label by calling **setActionCommand( )**. This method changes the action command string, but does not affect the string used to

label the button. Thus, setting the action command enables the action command and the label of a button to differ.

In some cases, you can handle the action events generated by a button (or some other type of control) by use of an anonymous inner class (as described in Chapter 24) or a lambda expression (discussed in Chapter 15). For example, assuming the previous programs, here is a set of action event handlers that use lambda expressions:

```
// Use lambda expressions to handle action events.
yes.addActionListener((ae) -> {
  msg = "You pressed " + ae.getActionCommand();
  repaint();
});

no.addActionListener((ae) -> {
  msg = "You pressed " + ae.getActionCommand();
  repaint();
});

maybe.addActionListener((ae) -> {
  msg = "You pressed " + ae.getActionCommand();
  repaint();
});
```

This code works because **ActionListener** defines a functional interface, which is an interface with exactly one abstract method. Thus, it can be used by a lambda expression. In general, you can use a lambda expression to handle an AWT event when its listener defines a functional interface. For example, **ItemListener** is also a functional interface. Of course, whether you use the traditional approach, an anonymous inner class, or a lambda expression will be determined by the precise nature of your application.

Applying Check Boxes

A *check box* is a control that is used to turn an option on or off. It consists of a small box that can either contain a check mark or not. There is a label associated with each check box that describes what option the box represents. You change the state of a check box by clicking on it. Check boxes can be used individually or as part of a group. Check boxes are objects of the **Checkbox** class.

Checkbox supports these constructors:

Checkbox( ) throws HeadlessException
Checkbox(String *str*) throws HeadlessException
Checkbox(String *str*, boolean *on*) throws HeadlessException
Checkbox(String *str*, boolean *on*, CheckboxGroup *cbGroup*) throws HeadlessException
Checkbox(String *str*, CheckboxGroup *cbGroup*, boolean *on*) throws HeadlessException

The first form creates a check box whose label is initially blank. The state of the check box is unchecked. The second form creates a check box whose label is specified by *str*. The state of the check box is unchecked. The third form allows you to set the initial state of the check box. If *on* is **true**, the check box is initially checked; otherwise, it is cleared. The fourth and fifth forms create a check box whose label is specified by *str* and whose group is specified by *cbGroup*. If this check box is not part of a group, then *cbGroup* must be **null**. (Check box groups are described in the next section.) The value of *on* determines the initial state of the check box.

To retrieve the current state of a check box, call **getState( )**. To set its state, call **setState( )**. You can obtain the current label associated with a check box by calling **getLabel( )**. To set the label, call **setLabel( )**. These methods are as follows:

boolean getState( )
void setState(boolean *on*)
String getLabel( )
void setLabel(String *str*)

Here, if *on* is **true**, the box is checked. If it is **false**, the box is cleared. The string passed in *str* becomes the new label associated with the invoking check box.

Handling Check Boxes

Each time a check box is selected or deselected, an item event is generated. This is sent to any listeners that previously registered an interest in receiving item event notifications from that component. Each listener implements the **ItemListener** interface. That interface defines the **itemStateChanged( )** method. An **ItemEvent** object is supplied as the argument to this method. It contains information about the event (for example, whether it was a selection or deselection).

The following program creates four check boxes. The initial state of the first

box is checked. The status of each check box is displayed. Each time you change the state of a check box, the status display is updated.

```
// Demonstrate check boxes.
import java.awt.*;
import java.awt.event.*;

public class CheckboxDemo extends Frame implements ItemListener {
  String msg = "";
  Checkbox windows, android, solaris, mac;

  public CheckboxDemo() {

    // Use a flow layout.
    setLayout(new FlowLayout());

    // Create check boxes.
    windows = new Checkbox("Windows", true);
    android = new Checkbox("Android");
    solaris = new Checkbox("Solaris");
    mac = new Checkbox("Mac OS");

    // Add the check boxes to the frame.
    add(windows);
    add(android);
    add(solaris);
    add(mac);

    // Add item listeners.
    windows.addItemListener(this);
    android.addItemListener(this);
    solaris.addItemListener(this);
    mac.addItemListener(this);

    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }
```

```
  public void itemStateChanged(ItemEvent ie) {
    repaint();
  }

  // Display current state of the check boxes.
  public void paint(Graphics g) {
    msg = "Current state: ";
    g.drawString(msg, 20, 120);
    msg = "   Windows: " + windows.getState();
    g.drawString(msg, 20, 140);
    msg = "   Android: " + android.getState();
    g.drawString(msg, 20, 160);
    msg = "   Solaris: " + solaris.getState();
    g.drawString(msg, 20, 180);
    msg = "   Mac OS: " + mac.getState();
    g.drawString(msg, 20, 200);
  }

  public static void main(String[] args) {
    CheckboxDemo appwin = new CheckboxDemo();

    appwin.setSize(new Dimension(240, 220));
    appwin.setTitle("CheckboxDemo");
    appwin.setVisible(true);
  }
}
```

Sample output is shown in Figure 26-2.

Figure 26-2 Sample output from the **CheckboxDemo** program

CheckboxGroup

It is possible to create a set of mutually exclusive check boxes in which one and only one check box in the group can be checked at any one time. These check boxes are often called *radio buttons,* because they act like the station selector on a car radio—only one station can be selected at any one time. To create a set of mutually exclusive check boxes, you must first define the group to which they will belong and then specify that group when you construct the check boxes. Check box groups are objects of type **CheckboxGroup**. Only the default constructor is defined, which creates an empty group.

You can determine which check box in a group is currently selected by calling **getSelectedCheckbox( )**. You can set a check box by calling **setSelectedCheckbox( )**. These methods are as follows:

Checkbox getSelectedCheckbox( )
void setSelectedCheckbox(Checkbox *which*)

Here, *which* is the check box that you want to be selected. The previously selected check box will be turned off.

Here is a program that uses check boxes that are part of a group:

```
// Demonstrate check box group.
import java.awt.*;
import java.awt.event.*;

public class CBGroup extends Frame implements ItemListener {
  String msg = "";
  Checkbox windows, android, solaris, mac;
  CheckboxGroup cbg;

  public CBGroup() {

    // Use a flow layout.
    setLayout(new FlowLayout());

    // Create a check box group.
    cbg = new CheckboxGroup();

    // Create the check boxes and include them
    // in the group.
    windows = new Checkbox("Windows", cbg, true);
    android = new Checkbox("Android", cbg, false);
    solaris = new Checkbox("Solaris", cbg, false);
    mac = new Checkbox("Mac OS", cbg, false);

   // Add the check boxes to the frame.
    add(windows);
    add(android);
    add(solaris);
    add(mac);
```

```
    // Add item listeners.
    windows.addItemListener(this);
    android.addItemListener(this);
    solaris.addItemListener(this);
    mac.addItemListener(this);

    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }

  public void itemStateChanged(ItemEvent ie) {
    repaint();
  }

  // Display current state of the check boxes.
  public void paint(Graphics g) {
    msg = "Current selection: ";
    msg += cbg.getSelectedCheckbox().getLabel();
    g.drawString(msg, 20, 120);
  }

  public static void main(String[] args) {
    CBGroup appwin = new CBGroup();

    appwin.setSize(new Dimension(240, 180));
    appwin.setTitle("CBGroup");
    appwin.setVisible(true);
  }
}
```

Sample output generated by the **CBGroup** program is shown in Figure 26-3. Notice that the check boxes are now circular in shape.

Figure 26-3 Sample output from the **CBGroup** program

Choice Controls

The **Choice** class is used to create a *pop-up list* of items from which the user may choose. Thus, a **Choice** control is a form of menu. When inactive, a **Choice** component takes up only enough space to show the currently selected item. When the user clicks on it, the whole list of choices pops up, and a new selection can be made. Each item in the list is a string that appears as a left-justified label in the order it is added to the **Choice** object. **Choice** defines only the default constructor, which creates an empty list.

To add a selection to the list, call **add( )**. It has this general form:

```
void add(String name)
```

Here, *name* is the name of the item being added. Items are added to the list in the order in which calls to **add( )** occur.

To determine which item is currently selected, you may call either **getSelectedItem( )** or **getSelectedIndex( )**. These methods are shown here:

```
String getSelectedItem( )
int getSelectedIndex( )
```

The **getSelectedItem( )** method returns a string containing the name of the item. **getSelectedIndex( )** returns the index of the item. The first item is at index 0. By default, the first item added to the list is selected.

To obtain the number of items in the list, call **getItemCount( )**. You can set the currently selected item using the **select( )** method with either a zero-based integer index or a string that will match a name in the list. These methods are shown here:

int getItemCount( )
void select(int *index*)
void select(String *name*)

Given an index, you can obtain the name associated with the item at that index by calling **getItem( )**, which has this general form:

String getItem(int *index*)

Here, *index* specifies the index of the desired item.

Handling Choice Lists

Each time a choice is selected, an item event is generated. This is sent to any listeners that previously registered an interest in receiving item event notifications from that component. Each listener implements the **ItemListener** interface. That interface defines the **itemStateChanged( )** method. An **ItemEvent** object is supplied as the argument to this method.

Here is an example that creates two **Choice** menus. One selects the operating system. The other selects the browser.

```
// Demonstrate Choice lists.
import java.awt.*;
import java.awt.event.*;

public class ChoiceDemo extends Frame implements ItemListener {
  Choice os, browser;
  String msg = "";

  public ChoiceDemo() {

    // Use a flow layout.
    setLayout(new FlowLayout());

    // Create choice lists.
    os = new Choice();
    browser = new Choice();

    // Add items to os list.
    os.add("Windows");
    os.add("Android");
    os.add("Solaris");
    os.add("Mac OS");

    // Add items to browser list.
    browser.add("Internet Explorer");
    browser.add("Firefox");
    browser.add("Chrome");

    // Add choice lists to window.
    add(os);
    add(browser);

    // Add item listeners.
    os.addItemListener(this);
    browser.addItemListener(this);
```

```
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }

  public void itemStateChanged(ItemEvent ie) {
    repaint();
  }

  // Display current selections.
  public void paint(Graphics g) {
    msg = "Current OS: ";
    msg += os.getSelectedItem();
    g.drawString(msg, 20, 120);
    msg = "Current Browser: ";
    msg += browser.getSelectedItem();
    g.drawString(msg, 20, 140);
  }

  public static void main(String[] args) {
    ChoiceDemo appwin = new ChoiceDemo();

    appwin.setSize(new Dimension(240, 180));
    appwin.setTitle("ChoiceDemo");
    appwin.setVisible(true);
  }
}
```

Sample output is shown in Figure 26-4.

Figure 26-4 Sample output from the **ChoiceDemo** program

Using Lists

The **List** class provides a compact, multiple-choice, scrolling selection list. Unlike the **Choice** object, which shows only the single selected item in the menu, a **List** object can be constructed to show any number of choices in the visible window. It can also be created to allow multiple selections. **List** provides these constructors:

List( ) throws HeadlessException
List(int *numRows*) throws HeadlessException
List(int *numRows*, boolean *multipleSelect*) throws HeadlessException

The first version creates a **List** control that allows only one item to be selected at any one time. In the second form, the value of *numRows* specifies the number of entries in the list that will always be visible (others can be scrolled into view as needed). In the third form, if *multipleSelect* is **true**, then the user may select two or more items at a time. If it is **false**, then only one item may be selected.

To add a selection to the list, call **add( )**. It has the following two forms:

void add(String *name*)
void add(String *name*, int *index*)

Here, *name* is the name of the item added to the list. The first form adds items to the end of the list. The second form adds the item at the index specified by *index*. Indexing begins at zero. You can specify –1 to add the item to the end of the list.

For lists that allow only single selection, you can determine which item is currently selected by calling either **getSelectedItem( )** or **getSelectedIndex( )**. These methods are shown here:

String getSelectedItem( )
int getSelectedIndex( )

The **getSelectedItem( )** method returns a string containing the name of the item. If more than one item is selected, or if no selection has yet been made, **null** is returned. **getSelectedIndex( )** returns the index of the item. The first item is at index 0. If more than one item is selected, or if no selection has yet been made, –1 is returned.

For lists that allow multiple selection, you must use either **getSelectedItems(**

) or **getSelectedIndexes( )**, shown here, to determine the current selections:

```
String[ ] getSelectedItems( )
int[ ] getSelectedIndexes( )
```

getSelectedItems( ) returns an array containing the names of the currently selected items. **getSelectedIndexes( )** returns an array containing the indexes of the currently selected items.

To obtain the number of items in the list, call **getItemCount( )**. You can set the currently selected item by using the **select( )** method with a zero-based integer index. These methods are shown here:

```
int getItemCount( )
void select(int index)
```

Given an index, you can obtain the name associated with the item at that index by calling **getItem( )**, which has this general form:

```
String getItem(int index)
```

Here, *index* specifies the index of the desired item.

Handling Lists

To process list events, you will need to implement the **ActionListener** interface. Each time a **List** item is double-clicked, an **ActionEvent** object is generated. Its **getActionCommand( )** method can be used to retrieve the name of the newly selected item. Also, each time an item is selected or deselected with a single click, an **ItemEvent** object is generated. Its **getStateChange( )** method can be used to determine whether a selection or deselection triggered this event. **getItemSelectable( )** returns a reference to the object that triggered this event.

Here is an example that converts the **Choice** controls in the preceding section into **List** components, one multiple choice and the other single choice:

```
// Demonstrate Lists.
import java.awt.*;
import java.awt.event.*;

public class ListDemo extends Frame implements ActionListener {
  List os, browser;
  String msg = "";

  public ListDemo() {

    // Use a flow layout.
    setLayout(new FlowLayout());

    // Create a multi-selection list.
    os = new List(4, true);

    // Create a single-selection list.
    browser = new List(4);

    // Add items to os list.
    os.add("Windows");
    os.add("Android");
    os.add("Solaris");
    os.add("Mac OS");

    // Add items to browser list.
    browser.add("Internet Explorer");
    browser.add("Firefox");
    browser.add("Chrome");

    // Make initial selections.
    browser.select(1);
    os.select(0);

    // Add lists to the frame.
    add(os);
    add(browser);
```

```
    // Add action listeners.
    os.addActionListener(this);
    browser.addActionListener(this);

    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }

  public void actionPerformed(ActionEvent ae) {
    repaint();
  }

  // Display current selections.
  public void paint(Graphics g) {
    int idx[];

    msg = "Current OS: ";
    idx = os.getSelectedIndexes();
    for(int i=0; i<idx.length; i++)
      msg += os.getItem(idx[i]) + "  ";
    g.drawString(msg, 6, 120);
    msg = "Current Browser: ";
    msg += browser.getSelectedItem();
    g.drawString(msg, 6, 140);
  }

  public static void main(String[] args) {
    ListDemo appwin = new ListDemo();

    appwin.setSize(new Dimension(300, 180));
    appwin.setTitle("ListDemo");
    appwin.setVisible(true);
  }
}
```

Sample output generated by the **ListDemo** program is shown in Figure 26-5.

Figure 26-5 Sample output from the **ListDemo** program

Managing Scroll Bars

Scroll bars are used to select continuous values between a specified minimum and maximum. Scroll bars may be oriented horizontally or vertically. A scroll bar is actually a composite of several individual parts. Each end has an arrow that you can click to move the current value of the scroll bar one unit in the direction of the arrow. The current value of the scroll bar relative to its minimum and maximum values is indicated by the *slider box* (or *thumb*) for the scroll bar. The slider box can be dragged by the user to a new position. The scroll bar will then reflect this value. In the background space on either side of the thumb, the user can click to cause the thumb to jump in that direction by some increment larger than 1. Typically, this action translates into some form of page up and page down. Scroll bars are encapsulated by the **Scrollbar** class.

Scrollbar defines the following constructors:

Scrollbar( ) throws HeadlessException
Scrollbar(int *style*) throws HeadlessException
Scrollbar(int *style*, int *initialValue*, int *thumbSize*, int *min*, int *max*)
throws HeadlessException

The first form creates a vertical scroll bar. The second and third forms allow you to specify the orientation of the scroll bar. If *style* is **Scrollbar.VERTICAL**, a vertical scroll bar is created. If *style* is **Scrollbar.HORIZONTAL**, the scroll bar is horizontal. In the third form of the constructor, the initial value of the scroll bar is passed in *initialValue*. The number of units represented by the height of the thumb is passed in *thumbSize*. The minimum and maximum values for the scroll bar are specified by *min* and *max*.

If you construct a scroll bar by using one of the first two constructors, then you need to set its parameters by using **setValues( )**, shown here, before it can be used:

void setValues(int *initialValue*, int *thumbSize*, int *min*, int *max*)

The parameters have the same meaning as they have in the third constructor just described.

To obtain the current value of the scroll bar, call **getValue( )**. It returns the current setting. To set the current value, call **setValue( )**. These methods are as follows:

int getValue( )
void setValue(int *newValue*)

Here, *newValue* specifies the new value for the scroll bar. When you set a value, the slider box inside the scroll bar will be positioned to reflect the new value.

You can also retrieve the minimum and maximum values via **getMinimum( )** and **getMaximum( )**, shown here:

int getMinimum( )
int getMaximum( )

They return the requested quantity.

By default, 1 is the increment added to or subtracted from the scroll bar each time it is scrolled up or down one line. You can change this increment by calling **setUnitIncrement( )**. By default, page-up and page-down increments are 10. You can change this value by calling **setBlockIncrement( )**. These methods are shown here:

void setUnitIncrement(int *newIncr*)
void setBlockIncrement(int *newIncr*)

Handling Scroll Bars

To process scroll bar events, you need to implement the **AdjustmentListener** interface. Each time a user interacts with a scroll bar, an **AdjustmentEvent** object is generated. Its **getAdjustmentType( )** method can be used to determine the type of the adjustment. The types of adjustment events are as follows:

BLOCK_DECREMENT	A page-down event has been generated.
BLOCK_INCREMENT	A page-up event has been generated.
TRACK	An absolute tracking event has been generated.
UNIT_DECREMENT	The line-down button in a scroll bar has been pressed.
UNIT_INCREMENT	The line-up button in a scroll bar has been pressed.

The following example creates both a vertical and a horizontal scroll bar. The current settings of the scroll bars are displayed. If you drag the mouse while inside the window, the coordinates of each drag event are used to update the scroll bars. An asterisk is displayed at the current drag position. Notice the use of **setPreferredSize( )** to set the size of the scrollbars.

```
// Demonstrate scroll bars.
import java.awt.*;
import java.awt.event.*;

public class SBDemo extends Frame
  implements AdjustmentListener {

  String msg = "";
  Scrollbar vertSB, horzSB;

  public SBDemo() {

    // Use a flow layout.
    setLayout(new FlowLayout());

    // Create scroll bars and set preferred size.
    vertSB = new Scrollbar(Scrollbar.VERTICAL,
                           0, 1, 0, 200);
    vertSB.setPreferredSize(new Dimension(20, 100));

    horzSB = new Scrollbar(Scrollbar.HORIZONTAL,
                           0, 1, 0, 100);
    horzSB.setPreferredSize(new Dimension(100, 20));

    // Add the scroll bars to the frame.
    add(vertSB);
    add(horzSB);

    // Add AdjustmentListeners for the scroll bars.
    vertSB.addAdjustmentListener(this);
    horzSB.addAdjustmentListener(this);
```

```
    // Add MouseMotionListener.
    addMouseMotionListener(new MouseAdapter() {
      // Update scroll bars to reflect mouse dragging.
      public void mouseDragged(MouseEvent me) {
        int x = me.getX();
        int y = me.getY();
        vertSB.setValue(y);
        horzSB.setValue(x);
        repaint();
      }
    });

    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }

  public void adjustmentValueChanged(AdjustmentEvent ae) {
    repaint();
  }

  // Display current value of scroll bars.
  public void paint(Graphics g) {
     msg = "Vertical: " + vertSB.getValue();
     msg += ",  Horizontal: " + horzSB.getValue();
     g.drawString(msg, 20, 160);

     // show current mouse drag position
     g.drawString("*", horzSB.getValue(),
                  vertSB.getValue());
  }

  public static void main(String[] args) {
    SBDemo appwin = new SBDemo();

    appwin.setSize(new Dimension(300, 180));
    appwin.setTitle("SBDemo");
    appwin.setVisible(true);
  }
}
```

Sample output from the **SBDemo** program is shown in Figure 26-6.

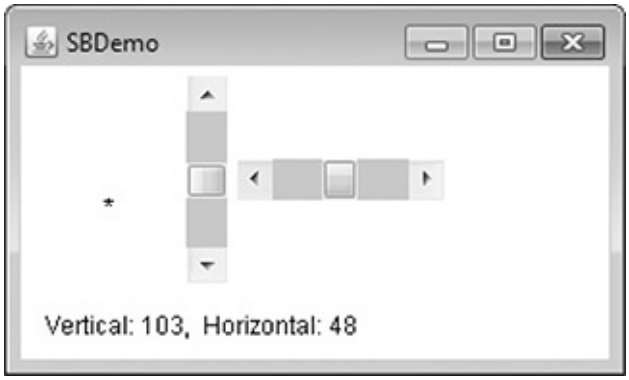


Figure 26-6 Sample output from the **SBDemo** program

Using a TextField

The **TextField** class implements a single-line text-entry area, usually called an *edit control*. Text fields allow the user to enter strings and to edit the text using the arrow keys, cut and paste keys, and mouse selections. **TextField** is a subclass of **TextComponent**. **TextField** defines the following constructors:

TextField( ) throws HeadlessException
TextField(int *numChars*) throws HeadlessException
TextField(String *str*) throws HeadlessException
TextField(String *str*, int *numChars*) throws HeadlessException

The first version creates a default text field. The second form creates a text field that is *numChars* characters wide. The third form initializes the text field with the string contained in *str*. The fourth form initializes a text field and sets its width.

TextField (and its superclass **TextComponent**) provides several methods that allow you to utilize a text field. To obtain the string currently contained in the text field, call **getText( )**. To set the text, call **setText( )**. These methods are as follows:

String getText( )
void setText(String *str*)

Here, *str* is the new string.

The user can select a portion of the text in a text field. Also, you can select a

portion of text under program control by using **select( )**. Your program can obtain the currently selected text by calling **getSelectedText( )**. These methods are shown here:

String getSelectedText( )
void select(int *startIndex*, int *endIndex*)

getSelectedText( ) returns the selected text. The **select( )** method selects the characters beginning at *startIndex* and ending at *endIndex* –1.

You can control whether the contents of a text field may be modified by the user by calling **setEditable( )**. You can determine editability by calling **isEditable( )**. These methods are shown here:

boolean isEditable( )
void setEditable(boolean *canEdit*)

isEditable( ) returns **true** if the text may be changed and **false** if not. In **setEditable( )**, if *canEdit* is **true**, the text may be changed. If it is **false**, the text cannot be altered.

There may be times when you will want the user to enter text that is not displayed, such as a password. You can disable the echoing of the characters as they are typed by calling **setEchoChar( )**. This method specifies a single character that the **TextField** will display when characters are entered (thus, the actual characters typed will not be shown). You can check a text field to see if it is in this mode with the **echoCharIsSet( )** method. You can retrieve the echo character by calling the **getEchoChar( )** method. These methods are as follows:

void setEchoChar(char *ch*)
boolean echoCharIsSet( )
char getEchoChar( )

Here, *ch* specifies the character to be echoed. If *ch* is zero, then normal echoing is restored.

Handling a TextField

Since text fields perform their own editing functions, your program generally will not respond to individual key events that occur within a text field. However, you may want to respond when the user presses ENTER. When this occurs, an action event is generated.

Here is an example that creates the classic user name and password screen:

```
// Demonstrate text field.
import java.awt.*;
import java.awt.event.*;

public class TextFieldDemo extends Frame
  implements ActionListener {

  TextField name, pass;

  public TextFieldDemo() {

    // Use a flow layout.
    setLayout(new FlowLayout());

    // Create controls.
    Label namep = new Label("Name: ", Label.RIGHT);
    Label passp = new Label("Password: ", Label.RIGHT);
    name = new TextField(12);
    pass = new TextField(8);
    pass.setEchoChar('?');

    // Add the controls to the frame.
    add(namep);
    add(name);
    add(passp);
    add(pass);

    // Add action event handlers.
    name.addActionListener(this);
    pass.addActionListener(this);
```

```
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }

  // User pressed Enter.
  public void actionPerformed(ActionEvent ae) {
    repaint();
  }

  public void paint(Graphics g) {
     g.drawString("Name: " + name.getText(), 20, 100);
     g.drawString("Selected text in name: "
                  + name.getSelectedText(), 20, 120);
     g.drawString("Password: " + pass.getText(), 20, 140);
  }

  public static void main(String[] args) {
    TextFieldDemo appwin = new TextFieldDemo();

    appwin.setSize(new Dimension(380, 180));
    appwin.setTitle("TextFieldDemo");
    appwin.setVisible(true);
  }
}
```

Sample output from the **TextFieldDemo** program is shown in Figure 26-7.

Figure 26-7 Sample output from the **TextFieldDemo** program

Using a TextArea

Sometimes a single line of text input is not enough for a given task. To handle these situations, the AWT includes a simple multiline editor called **TextArea**. Following are the constructors for **TextArea**:

TextArea( ) throws HeadlessException
TextArea(int *numLines*, int *numChars*) throws HeadlessException
TextArea(String *str*) throws HeadlessException
TextArea(String *str*, int *numLines*, int *numChars*) throws HeadlessException
TextArea(String *str*, int *numLines*, int *numChars*, int *sBars*) throws HeadlessException

Here, *numLines* specifies the height, in lines, of the text area, and *numChars* specifies its width, in characters. Initial text can be specified by *str*. In the fifth form, you can specify the scroll bars that you want the control to have. *sBars* must be one of these values:

SCROLLBARS_BOTH	SCROLLBARS_NONE
SCROLLBARS_HORIZONTAL_ONLY	SCROLLBARS_VERTICAL_ONLY

TextArea is a subclass of **TextComponent**. Therefore, it supports the **getText( )**, **setText( )**, **getSelectedText( )**, **select( )**, **isEditable( )**, and **setEditable( )** methods described in the preceding section.

TextArea adds the following editing methods:

void append(String *str*)
void insert(String *str*, int *index*)
void replaceRange(String *str*, int *startIndex*, int *endIndex*)

The **append( )** method appends the string specified by *str* to the end of the current text. **insert( )** inserts the string passed in *str* at the specified index. To replace text, call **replaceRange( )**. It replaces the characters from *startIndex* to *endIndex*–1, with the replacement text passed in *str*.

Text areas are almost self-contained controls. Your program incurs virtually no management overhead. Normally, your program simply obtains the current text when it is needed. You can, however, listen for **TextEvent**s, if you choose.

The following program creates a **TextArea** control:

```
// Demonstrate TextArea.
import java.awt.*;
import java.awt.event.*;

public class TextAreaDemo extends Frame {

  public TextAreaDemo() {

    // Use a flow layout.
    setLayout(new FlowLayout());

    String val =
      "Java 9 is the latest version of the most\n" +
      "widely-used computer language for Internet programming.\n" +
      "Building on a rich heritage, Java has advanced both\n" +
      "the art and science of computer language design.\n\n" +
      "One of the reasons for Java's ongoing success is its\n" +
      "constant, steady rate of evolution. Java has never stood\n" +
      "still. Instead, Java has consistently adapted to the\n" +
      "rapidly changing landscape of the networked world.\n" +
      "Moreover, Java has often led the way, charting the\n" +
      "course for others to follow.";

    TextArea text = new TextArea(val, 10, 30);
    add(text);

    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }

  public static void main(String[] args) {
    TextAreaDemo appwin = new TextAreaDemo();

    appwin.setSize(new Dimension(300, 220));
    appwin.setTitle("TextAreaDemo");
    appwin.setVisible(true);
  }
}
```

Here is sample output from the **TextAreaDemo** program:

Understanding Layout Managers

All of the components that we have shown so far have been positioned by the **FlowLayout** layout manager. As we mentioned at the beginning of this chapter, a layout manager automatically arranges your controls within a window by using some type of algorithm. While it is possible to lay out Java controls by hand, you generally won't want to, for two main reasons. First, it is very tedious to manually lay out a large number of components. Second, sometimes the width and height information is not yet available when you need to arrange some control, because the native toolkit components haven't been realized. This is a chicken-and-egg situation; it is pretty confusing to figure out when it is okay to use the size of a given component to position it relative to another.

Each **Container** object has a layout manager associated with it. A layout manager is an instance of any class that implements the **LayoutManager** interface. The layout manager is set by the **setLayout( )** method. If no call to **setLayout( )** is made, then the default layout manager is used. Whenever a container is resized (or sized for the first time), the layout manager is used to position each of the components within it.

The **setLayout( )** method has the following general form:

void setLayout(LayoutManager *layoutObj*)

Here, *layoutObj* is a reference to the desired layout manager. If you wish to disable the layout manager and position components manually, pass **null** for *layoutObj*. If you do this, you will need to determine the shape and position of each component manually, using the **setBounds( )** method defined by **Component**. Normally, you will want to use a layout manager.

Each layout manager keeps track of a list of components that are stored by their names. The layout manager is notified each time you add a component to a container. Whenever the container needs to be resized, the layout manager is consulted via its **minimumLayoutSize( )** and **preferredLayoutSize( )** methods. Each component that is being managed by a layout manager contains the **getPreferredSize( )** and **getMinimumSize( )** methods. These return the preferred and minimum size required to display each component. The layout manager will honor these requests if at all possible, while maintaining the integrity of the layout policy. You may override these methods for controls that you subclass. Default values are provided otherwise.

Java has several predefined **LayoutManager** classes, several of which are described next. You can use the layout manager that best fits your application.

FlowLayout

You have already seen **FlowLayout** in action. It is the layout manager that the preceding examples have used. **FlowLayout** implements a simple layout style, which is similar to how words flow in a text editor. The direction of the layout is governed by the container's component orientation property, which, by default, is left to right, top to bottom. Therefore, by default, components are laid out line-by-line beginning at the upper-left corner. In all cases, when a line is filled, layout advances to the next line. A small space is left between each component, above and below, as well as left and right. Here are the constructors for **FlowLayout**:

FlowLayout( )
FlowLayout(int *how*)
FlowLayout(int *how*, int *horz*, int *vert*)

The first form creates the default layout, which centers components and leaves five pixels of space between each component. The second form lets you specify how each line is aligned. Valid values for *how* are as follows:

FlowLayout.LEFT
FlowLayout.CENTER
FlowLayout.RIGHT
FlowLayout.LEADING
FlowLayout.TRAILING

These values specify left, center, right, leading edge, and trailing edge

alignment, respectively. The third constructor allows you to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively.

You can see the effect of specifying an alignment with **FlowLayout** by substituting this line in the **CheckboxDemo** program shown earlier:

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

After making this change, the output will look like that shown here. Compare this with the original output, shown in Figure 26-2.

BorderLayout

The **BorderLayout** class implements a layout style that has four narrow, fixed-width components at the edges and one large area in the center. The four sides are referred to as north, south, east, and west. The middle area is called the center. **BorderLayout** is the default layout manager for **Frame**. Here are the constructors defined by **BorderLayout**:

BorderLayout( )
BorderLayout(int *horz*, int *vert*)

The first form creates a default border layout. The second allows you to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively.

BorderLayout defines the following commonly used constants that specify the regions:

BorderLayout.CENTER	BorderLayout.SOUTH
BorderLayout.EAST	BorderLayout.WEST
BorderLayout.NORTH	

When adding components, you will use these constants with the following form of **add( )**, which is defined by **Container**:

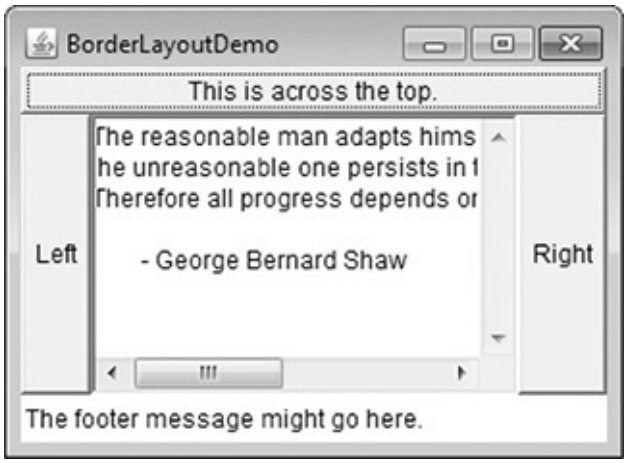
void add(Component *compRef,* Object *region*)

Here, *compRef* is a reference to the component to be added, and *region* specifies where the component will be added.

Here is an example of a **BorderLayout** with a component in each layout area:

```
// Demonstrate BorderLayout.
import java.awt.*;
import java.awt.event.*;

public class BorderLayoutDemo extends Frame {
  public BorderLayoutDemo() {

    // Here, BorderLayout is used by default.

    add(new Button("This is across the top."),
        BorderLayout.NORTH);
    add(new Label("The footer message might go here."),
        BorderLayout.SOUTH);
    add(new Button("Right"), BorderLayout.EAST);
    add(new Button("Left"), BorderLayout.WEST);

    String msg = "The reasonable man adapts " +
      "himself to the world;\n" +
      "the unreasonable one persists in " +
      "trying to adapt the world to himself.\n" +
      "Therefore all progress depends " +
      "on the unreasonable man.\n\n" +
      "         - George Bernard Shaw\n\n";

    add(new TextArea(msg), BorderLayout.CENTER);

    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }

  public static void main(String[] args) {
    BorderLayoutDemo appwin = new BorderLayoutDemo();

    appwin.setSize(new Dimension(300, 220));
    appwin.setTitle("BorderLayoutDemo");
    appwin.setVisible(true);
  }
}
```

Sample output from the **BorderLayoutDemo** program is shown here:



Using Insets

Sometimes you will want to leave a small amount of space between the container that holds your components and the window that contains it. To do this, override the **getInsets( )** method that is defined by **Container**. This method returns an **Insets** object that contains the top, bottom, left, and right inset to be used when the container is displayed. These values are used by the layout manager to inset the components when it lays out the window. The constructor for **Insets** is shown here:

Insets(int *top*, int *left*, int *bottom*, int *right*)

The values passed in *top*, *left*, *bottom*, and *right* specify the amount of space between the container and its enclosing window.

The **getInsets( )** method has this general form:

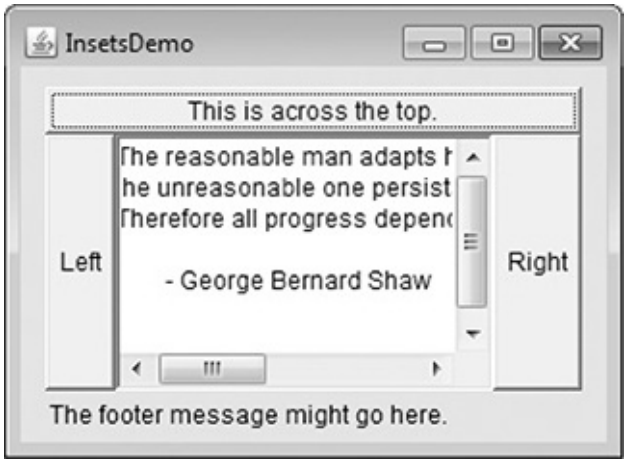
Insets getInsets( )

When overriding this method, you must return a new **Insets** object that contains the inset spacing you desire.

Here is the preceding **BorderLayout** example modified so that it insets its components. The background color has been set to cyan to help make the insets more visible.

```
// Demonstrate BorderLayout with insets.
import java.awt.*;
import java.awt.event.*;

public class InsetsDemo extends Frame {

  public InsetsDemo() {
    // Here, BorderLayout is used by default.

    // set background color so insets can be easily seen
    setBackground(Color.cyan);

    setLayout(new BorderLayout());
    add(new Button("This is across the top."),
        BorderLayout.NORTH);
    add(new Label("The footer message might go here."),
        BorderLayout.SOUTH);
    add(new Button("Right"), BorderLayout.EAST);
    add(new Button("Left"), BorderLayout.WEST);

    String msg = "The reasonable man adapts " +
      "himself to the world;\n" +
      "the unreasonable one persists in " +
      "trying to adapt the world to himself.\n" +
      "Therefore all progress depends " +
      "on the unreasonable man.\n\n" +
      "         - George Bernard Shaw\n\n";

    add(new TextArea(msg), BorderLayout.CENTER);

    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }

  // Override getInsets to add inset values.
  public Insets getInsets() {
    return new Insets(40, 20, 10, 20);
  }

  public static void main(String[] args) {
    InsetsDemo appwin = new InsetsDemo();

    appwin.setSize(new Dimension(300, 220));
    appwin.setTitle("InsetsDemo");
    appwin.setVisible(true);
  }
}
```

Sample output from the **InsetsDemo** program is shown here:



GridLayout

GridLayout lays out components in a two-dimensional grid. When you instantiate a **GridLayout**, you define the number of rows and columns. The constructors supported by **GridLayout** are shown here:

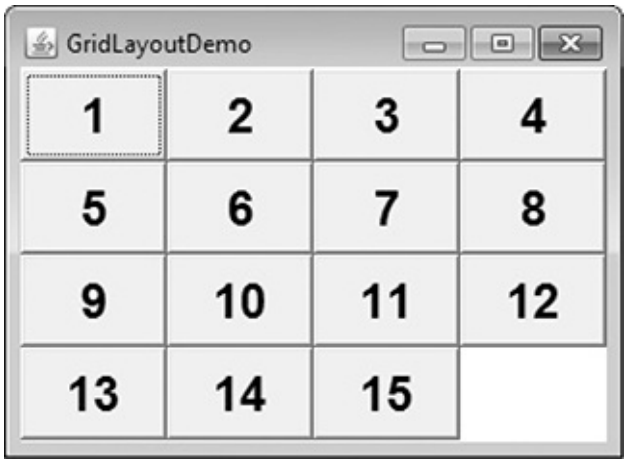
GridLayout( )
GridLayout(int *numRows*, int *numColumns*)
GridLayout(int *numRows*, int *numColumns*, int *horz*, int *vert*)

The first form creates a single-column grid layout. The second form creates a grid layout with the specified number of rows and columns. The third form allows you to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively. Either *numRows* or *numColumns* can be zero. Specifying *numRows* as zero allows for unlimited-length columns. Specifying *numColumns* as zero allows for unlimited-length rows.

Here is a sample program that creates a 4×4 grid and fills it in with 15 buttons, each labeled with its index:

```
// Demonstrate GridLayout
import java.awt.*;
import java.awt.event.*;

public class GridLayoutDemo extends Frame {
  static final int n = 4;

  public GridLayoutDemo() {

    // Use GridLayout.
    setLayout(new GridLayout(n, n));

    setFont(new Font("SansSerif", Font.BOLD, 24));

    for(int i = 0; i < n; i++) {
      for(int j = 0; j < n; j++) {
        int k = i * n + j;
        if(k > 0)
          add(new Button("" + k));
      }
    }

    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }

  public static void main(String[] args) {
    GridLayoutDemo appwin = new GridLayoutDemo();
    appwin.setSize(new Dimension(300, 220));
    appwin.setTitle("GridLayoutDemo");
    appwin.setVisible(true);
  }
}
```

Following is sample output generated by the **GridLayoutDemo** program:



TIP You might try using this example as the starting point for a 15-square puzzle.

CardLayout

The **CardLayout** class is unique among the other layout managers in that it stores several different layouts. Each layout can be thought of as being on a separate index card in a deck that can be shuffled so that any card is on top at a given time. This can be useful for user interfaces with optional components that can be dynamically enabled and disabled upon user input. You can prepare the other layouts and have them hidden, ready to be activated when needed.

CardLayout provides these two constructors:

CardLayout( )
CardLayout(int *horz*, int *vert*)

The first form creates a default card layout. The second form allows you to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively.

Use of a card layout requires a bit more work than the other layouts. The cards are typically held in an object of type **Panel**. This panel must have **CardLayout** selected as its layout manager. The cards that form the deck are also typically objects of type **Panel**. Thus, you must create a panel that contains the deck and a panel for each card in the deck. Next, you add to the appropriate panel the components that form each card. You then add these panels to the panel for which **CardLayout** is the layout manager. Finally, you add this panel to the window. Once these steps are complete, you must provide some way for the user to select between cards. One common approach is to include one push button for each card in the deck.

When card panels are added to a panel, they are usually given a name. One way to do this is to use this form of **add( )** when adding cards to a panel:

void add(Component *panelRef*, Object *name*)

Here, *name* is a string that specifies the name of the card whose panel is specified by *panelRef*.

After you have created a deck, your program activates a card by calling one of the following methods defined by **CardLayout**:

void first(Container *deck*)
void last(Container *deck*)
void next(Container *deck*)
void previous(Container *deck*)
void show(Container *deck*, String *cardName*)

Here, *deck* is a reference to the container (usually a panel) that holds the cards, and *cardName* is the name of a card. Calling **first( )** causes the first card in the deck to be shown. To show the last card, call **last( )**. To show the next card, call **next( )**. To show the previous card, call **previous( )**. Both **next( )** and **previous( )** automatically cycle back to the top or bottom of the deck, respectively. The **show( )** method displays the card whose name is passed in *cardName*.

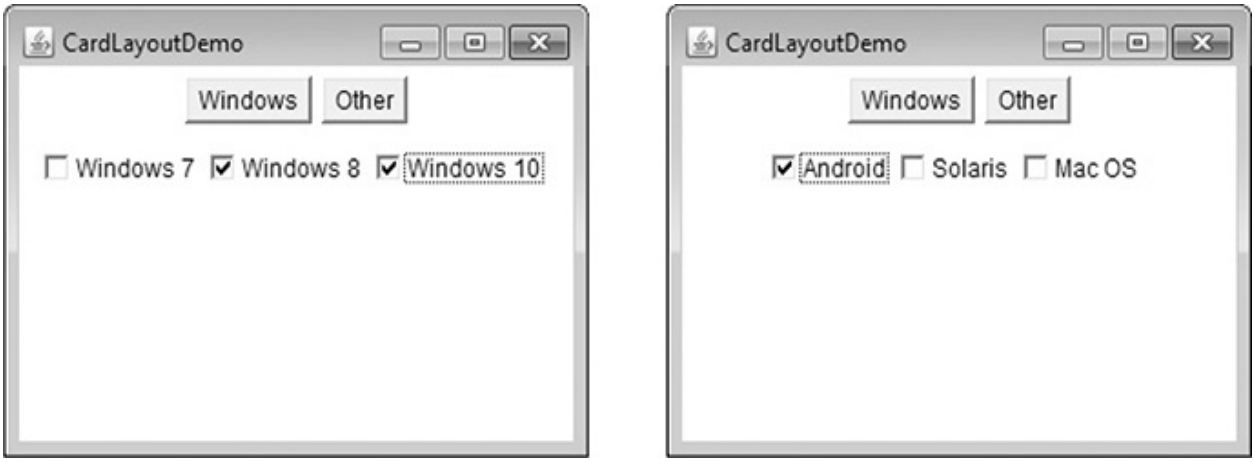
The following example creates a two-level card deck that allows the user to select an operating system. Windows-based operating systems are displayed in one card. Mac OS, Android, and Solaris are displayed in the other card.

```
// Demonstrate CardLayout.
import java.awt.*;
import java.awt.event.*;

public class CardLayoutDemo extends Frame {

  Checkbox windows10, windows7, windows8, android, solaris, mac;
  Panel osCards;
  CardLayout cardLO;
  Button Win, Other;

  public CardLayoutDemo() {

    // Use a flow layout for the main frame.
    setLayout(new FlowLayout());

    Win = new Button("Windows");
    Other = new Button("Other");
    add(Win);
    add(Other);

    // Set osCards panel to use CardLayout.
    cardLO = new CardLayout();
    osCards = new Panel();
    osCards.setLayout(cardLO);

    windows7 = new Checkbox("Windows 7", true);
    windows8 = new Checkbox("Windows 8");
    windows10 = new Checkbox("Windows 10");
    android = new Checkbox("Android");
    solaris = new Checkbox("Solaris");
    mac = new Checkbox("Mac OS");

    // Add Windows check boxes to a panel.
    Panel winPan = new Panel();
    winPan.add(windows7);
    winPan.add(windows8);
    winPan.add(windows10);
```

```
    // Add other OS check boxes to a panel.
    Panel otherPan = new Panel();
    otherPan.add(android);
    otherPan.add(solaris);
    otherPan.add(mac);

    // Add panels to card deck panel.
    osCards.add(winPan, "Windows");
    osCards.add(otherPan, "Other");

    // Add cards to main frame panel.
    add(osCards);

    // Use lambda expressions to handle button events.
    Win.addActionListener((ae) -> cardLO.show(osCards, "Windows"));
    Other.addActionListener((ae) -> cardLO.show(osCards, "Other"));

    // Register for mouse pressed events.
    addMouseListener(new MouseAdapter() {
      // Cycle through panels.
      public void mousePressed(MouseEvent me) {
        cardLO.next(osCards);
      }
    });

    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }

  public static void main(String[] args) {
    CardLayoutDemo appwin = new CardLayoutDemo();

    appwin.setSize(new Dimension(300, 220));
    appwin.setTitle("CardLayoutDemo");
    appwin.setVisible(true);
  }
}
```

Here is sample output generated by the **CardLayoutDemo** program. Each card

is activated by pushing its button. You can also cycle through the cards by clicking the mouse.



GridBagLayout

Although the preceding layouts are perfectly acceptable for many uses, some situations will require that you take a bit more control over how the components are arranged. A good way to do this is to use a grid bag layout, which is specified by the **GridBagLayout** class. What makes the grid bag useful is that you can specify the relative placement of components by specifying their positions within cells inside a grid. The key to the grid bag is that each component can be a different size, and each row in the grid can have a different number of columns. This is why the layout is called a *grid bag*. It's a collection of small grids joined together.

The location and size of each component in a grid bag are determined by a set of constraints linked to it. The constraints are contained in an object of type **GridBagConstraints**. Constraints include the height and width of a cell, and the placement of a component, its alignment, and its anchor point within the cell.

The general procedure for using a grid bag is to first create a new **GridBagLayout** object and to make it the current layout manager. Then, set the constraints that apply to each component that will be added to the grid bag. Finally, add the components to the layout manager. Although **GridBagLayout** is a bit more complicated than the other layout managers, it is still quite easy to use once you understand how it works.

GridBagLayout defines only one constructor, which is shown here:

GridBagLayout( )

GridBagLayout defines several methods, of which many are protected and not for general use. There is one method, however, that you must use: **setConstraints( )**. It is shown here:

void setConstraints(Component *comp*, GridBagConstraints *cons*)

Here, *comp* is the component for which the constraints specified by *cons* apply. This method sets the constraints that apply to each component in the grid bag.

The key to successfully using **GridBagLayout** is the proper setting of the constraints, which are stored in a **GridBagConstraints** object. **GridBagConstraints** defines several fields that you can set to govern the size, placement, and spacing of a component. These are shown in Table 26-1. Several are described in greater detail in the following discussion.

Field	Purpose
int anchor	Specifies the location of a component within a cell. The default is **GridBagConstraints.CENTER**.
int fill	Specifies how a component is resized if the component is smaller than its cell. Valid values are **GridBagConstraints.NONE** (the default), **GridBagConstraints.HORIZONTAL**, **GridBagConstraints.VERTICAL**, **GridBagConstraints.BOTH**.
int gridheight	Specifies the height of component in terms of cells. The default is 1.
int gridwidth	Specifies the width of component in terms of cells. The default is 1.
int gridx	Specifies the X coordinate of the cell to which the component will be added. The default value is **GridBagConstraints.RELATIVE**.
int gridy	Specifies the Y coordinate of the cell to which the component will be added. The default value is **GridBagConstraints.RELATIVE**.
Insets insets	Specifies the insets. Default insets are all zero.
int ipadx	Specifies extra horizontal space that surrounds a component within a cell. The default is 0.
int ipady	Specifies extra vertical space that surrounds a component within a cell. The default is 0.
double weightx	Specifies a weight value that determines the horizontal spacing between cells and the edges of the container that holds them. The default value is 0.0. The greater the weight, the more space that is allocated. If all values are 0.0, extra space is distributed evenly between the edges of the window.
double weighty	Specifies a weight value that determines the vertical spacing between cells and the edges of the container that holds them. The default value is 0.0. The greater the weight, the more space that is allocated. If all values are 0.0, extra space is distributed evenly between the edges of the window.

Table 26-1 Constraint Fields Defined by **GridBagConstraints**

GridBagConstraints also defines several static fields that contain standard constraint values, such as **GridBagConstraints.CENTER** and **GridBagConstraints.VERTICAL**.

When a component is smaller than its cell, you can use the **anchor** field to specify where within the cell the component's top-left corner will be located. There are three types of values that you can give to **anchor**. The first are absolute:

GridBagConstraints.CENTER	GridBagConstraints.SOUTH
GridBagConstraints.EAST	GridBagConstraints.SOUTHEAST
GridBagConstraints.NORTH	GridBagConstraints.SOUTHWEST
GridBagConstraints.NORTHEAST	GridBagConstraints.WEST
GridBagConstraints.NORTHWEST	

As their names imply, these values cause the component to be placed at the specific locations.

The second type of values that can be given to **anchor** is relative, which means the values are relative to the container's orientation, which might differ for non-Western languages. The relative values are shown here:

GridBagConstraints.FIRST_LINE_END	GridBagConstraints.LINE_END
GridBagConstraints.FIRST_LINE_START	GridBagConstraints.LINE_START
GridBagConstraints.LAST_LINE_END	GridBagConstraints.PAGE_END
GridBagConstraints.LAST_LINE_START	GridBagConstraints.PAGE_START

Their names describe the placement.

The third type of values that can be given to **anchor** allows you to position components relative to the baseline of the row. These values are shown here:

GridBagConstraints.BASELINE	GridBagConstraints.BASELINE_LEADING
GridBagConstraints.BASELINE_TRAILING	GridBagConstraints.ABOVE_BASELINE
GridBagConstraints.ABOVE_BASELINE_LEADING	GridBagConstraints.ABOVE_BASELINE_ TRAILING
GridBagConstraints.BELOW_BASELINE	GridBagConstraints.BELOW_BASELINE_ LEADING
GridBagConstraints. BELOW_BASELINE_TRAILING	

The horizontal position can be either centered, against the leading edge (LEADING), or against the trailing edge (TRAILING).

The **weightx** and **weighty** fields are both quite important and quite confusing at first glance. In general, their values determine how much of the extra space within a container is allocated to each row and column. By default, both these values are zero. When all values within a row or a column are zero, extra space is distributed evenly between the edges of the window. By increasing the weight, you increase that row or column's allocation of space proportional to the other rows or columns. The best way to understand how these values work is to experiment with them a bit.

The **gridwidth** variable lets you specify the width of a cell in terms of cell units. The default is 1. To specify that a component use the remaining space in a row, use **GridBagConstraints.REMAINDER**. To specify that a component use the next-to-last cell in a row, use **GridBagConstraints.RELATIVE**. The **gridheight** constraint works the same way, but in the vertical direction.

You can specify a padding value that will be used to increase the minimum size of a cell. To pad horizontally, assign a value to **ipadx**. To pad vertically, assign a value to **ipady**.

Here is an example that uses **GridBagLayout** to demonstrate several of the points just discussed:

```
// Use GridBagLayout.
import java.awt.*;
import java.awt.event.*;

public class GridBagDemo extends Frame
  implements ItemListener {

  String msg = "";
  Checkbox windows, android, solaris, mac;

  public GridBagDemo() {

    // Use a GridBagLayout
    GridBagLayout gbag = new GridBagLayout();
    GridBagConstraints gbc = new GridBagConstraints();
    setLayout(gbag);

    // Define check boxes.
    windows = new Checkbox("Windows ", true);
    android = new Checkbox("Android");
    solaris = new Checkbox("Solaris");
    mac = new Checkbox("Mac OS");

    // Define the grid bag.

    // Use default row weight of 0 for first row.
    gbc.weightx = 1.0; // use a column weight of 1
    gbc.ipadx = 200; // pad by 200 units
    gbc.insets = new Insets(0, 6, 0, 0); // inset slightly from left

    gbc.anchor = GridBagConstraints.NORTHEAST;

    gbc.gridwidth = GridBagConstraints.RELATIVE;
    gbag.setConstraints(windows, gbc);
```

```
    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbag.setConstraints(android, gbc);

    // Give second row a weight of 1.
    gbc.weighty = 1.0;

    gbc.gridwidth = GridBagConstraints.RELATIVE;
    gbag.setConstraints(solaris, gbc);

    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbag.setConstraints(mac, gbc);

    // Add the components.
    add(windows);
    add(android);
    add(solaris);
    add(mac);

    // Register to receive item events.
    windows.addItemListener(this);
    android.addItemListener(this);
    solaris.addItemListener(this);
    mac.addItemListener(this);

    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
}
```

```
  // Repaint when status of a check box changes.
  public void itemStateChanged(ItemEvent ie) {
    repaint();
  }

  // Display current state of the check boxes.
  public void paint(Graphics g) {
    msg = "Current state: ";
    g.drawString(msg, 20, 100);
    msg = "  Windows: " + windows.getState();
    g.drawString(msg, 30, 120);
    msg = "  Android: " + android.getState();
    g.drawString(msg, 30, 140);
    msg = "  Solaris: " + solaris.getState();
    g.drawString(msg, 30, 160);
    msg = "  Mac: " + mac.getState();
    g.drawString(msg, 30, 180);
  }

  public static void main(String[] args) {
    GridBagDemo appwin = new GridBagDemo();

    appwin.setSize(new Dimension(250, 200));
    appwin.setTitle("GridBagDemo");
    appwin.setVisible(true);
  }
}
```

Sample output produced by the program is shown here.

In this layout, the operating system check boxes are positioned in a 2×2 grid.

Each cell has a horizontal padding of 200. Each component is inset slightly (by 6 units) from the left. The column weight is set to 1, which causes any extra horizontal space to be distributed evenly between the columns. The first row uses a default weight of 0; the second has a weight of 1. This means that any extra vertical space is added to the second row.

GridBagLayout is a powerful layout manager. It is worth taking some time to experiment with and explore. Once you understand what the various settings do, you can use **GridBagLayout** to position components with a high degree of precision.

Menu Bars and Menus

A top-level window can have a menu bar associated with it. A menu bar displays a list of top-level menu choices. Each choice is associated with a drop-down menu. This concept is implemented in the AWT by the following classes: **MenuBar**, **Menu**, and **MenuItem**. In general, a menu bar contains one or more **Menu** objects. Each **Menu** object contains a list of **MenuItem** objects. Each **MenuItem** object represents something that can be selected by the user. Since **Menu** is a subclass of **MenuItem**, a hierarchy of nested submenus can be created. It is also possible to include checkable menu items. These are menu options of type **CheckboxMenuItem** and will have a check mark next to them when they are selected.

To create a menu bar, first create an instance of **MenuBar**. This class defines only the default constructor. Next, create instances of **Menu** that will define the selections displayed on the bar. Following are the constructors for **Menu**:

Menu( ) throws HeadlessException
Menu(String *optionName*) throws HeadlessException
Menu(String *optionName*, boolean *removable*) throws HeadlessException

Here, *optionName* specifies the name of the menu selection. If *removable* is **true**, the menu can be removed and allowed to float free. Otherwise, it will remain attached to the menu bar. (Removable menus are implementation-dependent.) The first form creates an empty menu.

Individual menu items are of type **MenuItem**. It defines these constructors:

MenuItem( ) throws HeadlessException
MenuItem(String *itemName*) throws HeadlessException
MenuItem(String *itemName*, MenuShortcut *keyAccel*) throws

HeadlessException

Here, *itemName* is the name shown in the menu, and *keyAccel* is the menu shortcut for this item.

You can disable or enable a menu item by using the **setEnabled( )** method. Its form is shown here:

void setEnabled(boolean *enabledFlag*)

If the argument *enabledFlag* is **true**, the menu item is enabled. If **false**, the menu item is disabled.

You can determine an item's status by calling **isEnabled( )**. This method is shown here:

boolean isEnabled( )

isEnabled( ) returns **true** if the menu item on which it is called is enabled. Otherwise, it returns **false**.

You can change the name of a menu item by calling **setLabel( )**. You can retrieve the current name by using **getLabel( )**. These methods are as follows:

void setLabel(String *newName*)
String getLabel( )

Here, *newName* becomes the new name of the invoking menu item. **getLabel( )** returns the current name.

You can create a checkable menu item by using a subclass of **MenuItem** called **CheckboxMenuItem**. It has these constructors:

CheckboxMenuItem( ) throws HeadlessException
CheckboxMenuItem(String *itemName*) throws HeadlessException
CheckboxMenuItem(String *itemName*, boolean *on*) throws
HeadlessException

Here, *itemName* is the name shown in the menu. Checkable items operate as toggles. Each time one is selected, its state changes. In the first two forms, the checkable entry is unchecked. In the third form, if *on* is **true**, the checkable entry is initially checked. Otherwise, it is cleared.

You can obtain the status of a checkable item by calling **getState( )**. You can set it to a known state by using **setState( )**. These methods are shown here:

boolean getState( )

void setState(boolean *checked*)

If the item is checked, **getState( )** returns **true**. Otherwise, it returns **false**. To check an item, pass **true** to **setState( )**. To clear an item, pass **false**.

Once you have created a menu item, you must add the item to a **Menu** object by using **add( )**, which has the following general form:

MenuItem add(MenuItem *item*)

Here, *item* is the item being added. Items are added to a menu in the order in which the calls to **add( )** take place. The *item* is returned.

Once you have added all items to a **Menu** object, you can add that object to the menu bar by using this version of **add( )** defined by **MenuBar**:

Menu add(Menu *menu*)

Here, *menu* is the menu being added. The *menu* is returned.

Menus generate events only when an item of type **MenuItem** or **CheckboxMenuItem** is selected. They do not generate events when a menu bar is accessed to display a drop-down menu, for example. Each time a menu item is selected, an **ActionEvent** object is generated. By default, the action command string is the name of the menu item. However, you can specify a different action command string by calling **setActionCommand( )** on the menu item. Each time a check box menu item is checked or unchecked, an **ItemEvent** object is generated. Thus, you must implement the **ActionListener** and/or **ItemListener** interfaces in order to handle these menu events.

The **getItem( )** method of **ItemEvent** returns a reference to the item that generated this event. The general form of this method is shown here:

Object getItem( )

Following is an example that adds a series of nested menus to a pop-up window. The item selected is displayed in the window. The state of the two check box menu items is also displayed.

```
// Illustrate menus.
import java.awt.*;
import java.awt.event.*;

class MenuDemo extends Frame {
  String msg = "";
  CheckboxMenuItem debug, test;

  public MenuDemo() {

    // Create menu bar and add it to frame.
    MenuBar mbar = new MenuBar();
    setMenuBar(mbar);

    // Create the menu items.
    Menu file = new Menu("File");
    MenuItem item1, item2, item3, item4, item5;
    file.add(item1 = new MenuItem("New..."));
    file.add(item2 = new MenuItem("Open..."));
    file.add(item3 = new MenuItem("Close"));
    file.add(item4 = new MenuItem("-"));
    file.add(item5 = new MenuItem("Quit..."));
    mbar.add(file);

    Menu edit = new Menu("Edit");
    MenuItem item6, item7, item8, item9;
    edit.add(item6 = new MenuItem("Cut"));
    edit.add(item7 = new MenuItem("Copy"));
    edit.add(item8 = new MenuItem("Paste"));
    edit.add(item9 = new MenuItem("-"));

    Menu sub = new Menu("Special");
    MenuItem item10, item11, item12;
    sub.add(item10 = new MenuItem("First"));
    sub.add(item11 = new MenuItem("Second"));
    sub.add(item12 = new MenuItem("Third"));
    edit.add(sub);
```

```
    // These are checkable menu items.
    debug = new CheckboxMenuItem("Debug");
    edit.add(debug);
    test = new CheckboxMenuItem("Testing");
    edit.add(test);

    mbar.add(edit);

    // Create an object to handle action and item events.
    MyMenuHandler handler = new MyMenuHandler();

    // Register to receive those events.
    item1.addActionListener(handler);
    item2.addActionListener(handler);
    item3.addActionListener(handler);
    item4.addActionListener(handler);
    item6.addActionListener(handler);
    item7.addActionListener(handler);
    item8.addActionListener(handler);
    item9.addActionListener(handler);
    item10.addActionListener(handler);
    item11.addActionListener(handler);
    item12.addActionListener(handler);
    debug.addItemListener(handler);
    test.addItemListener(handler);

    // Use a lambda expression to handle the Quit selection.
    item5.addActionListener((ae) -> System.exit(0));

    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
}
```

```
public void paint(Graphics g) {
  g.drawString(msg, 10, 220);

 if(debug.getState())
    g.drawString("Debug is on.", 10, 240);
  else
    g.drawString("Debug is off.", 10, 240);

  if(test.getState())
    g.drawString("Testing is on.", 10, 260);
  else
    g.drawString("Testing is off.", 10, 260);
}

public static void main(String[] args) {
  MenuDemo appwin = new MenuDemo();

  appwin.setSize(new Dimension(250, 300));
  appwin.setTitle("MenuDemo");
  appwin.setVisible(true);
}

// An inner class for handling action and item events
// for the menu.
class MyMenuHandler implements ActionListener, ItemListener {
```

```
    // Handle action events.
    public void actionPerformed(ActionEvent ae) {
        msg = "You selected ";
        String arg = ae.getActionCommand();

        if(arg.equals("New..."))
          msg += "New.";
        else if(arg.equals("Open..."))
          msg += "Open.";
        else if(arg.equals("Close"))
          msg += "Close.";
        else if(arg.equals("Edit"))
          msg += "Edit.";
        else if(arg.equals("Cut"))
          msg += "Cut.";
        else if(arg.equals("Copy"))
          msg += "Copy.";
        else if(arg.equals("Paste"))
          msg += "Paste.";
        else if(arg.equals("First"))
          msg += "First.";
        else if(arg.equals("Second"))
          msg += "Second.";
        else if(arg.equals("Third"))
          msg += "Third.";
        else if(arg.equals("Debug"))
          msg += "Debug.";
        else if(arg.equals("Testing"))
          msg += "Testing.";

        repaint();
    }

    // Handle item events.
    public void itemStateChanged(ItemEvent ie) {
      repaint();
    }
  }
}
```

Sample output from the **MenuDemo** program is shown in Figure 26-8.

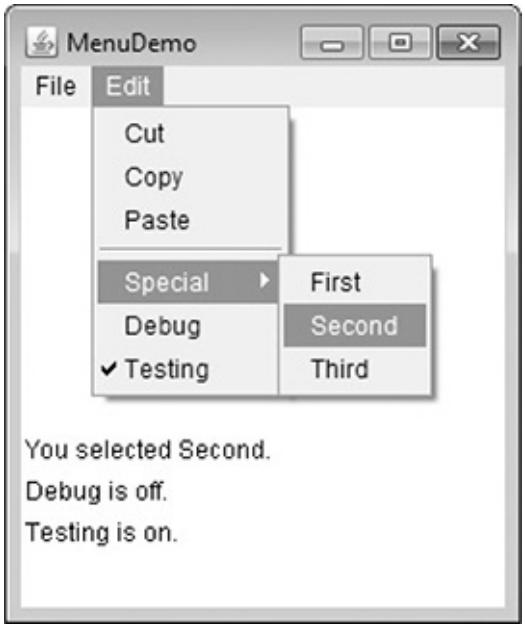


Figure 26-8 Sample output from the **MenuDemo** program

There is one other menu-related class that you might find interesting: **PopupMenu**. It works just like **Menu**, but produces a menu that can be displayed at a specific location. **PopupMenu** provides a flexible, useful alternative for some types of menuing situations.

Dialog Boxes

Often, you will want to use a *dialog box* to hold a set of related controls. Dialog boxes are primarily used to obtain user input and are often child windows of a top-level window. Dialog boxes don't have menu bars, but in other respects, they function like frame windows. (You can add controls to them, for example, in the same way that you add controls to a frame window.) Dialog boxes may be modal or modeless. In general terms, when a *modal* dialog box is active, you cannot access other windows in your program (except for child windows of the dialog window) until you have closed the dialog box. When a *modeless* dialog box is active, input focus can be directed to another window in your program. Thus, other parts of your program remain active and accessible. Beginning with JDK 6, modal dialog boxes can be created with three different types of modality, as specified by the **Dialog.ModalityType** enumeration. The default is **APPLICATION_MODAL**, which prevents the use of other top-level windows in the application. This is the traditional type of modality. Other types are

DOCUMENT_MODAL and **TOOLKIT_MODAL**. The **MODELESS** type is also included.

In the AWT, dialog boxes are of type **Dialog**. Two commonly used constructors are shown here:

Dialog(Frame *parentWindow*, boolean *mode*)
Dialog(Frame *parentWindow*, String *title*, boolean *mode*)

Here, *parentWindow* is the owner of the dialog box. If *mode* is **true**, the dialog box uses the default modality. Otherwise, it is modeless. The title of the dialog box can be passed in *title*. Generally, you will subclass **Dialog**, adding the functionality required by your application.

Following is a modified version of the preceding menu program that displays a modeless dialog box when the New option is chosen. Notice that when the dialog box is closed, **dispose( )** is called. This method is defined by **Window**, and it frees all system resources associated with the dialog box window.

```
// Illustrate a dialog box.
import java.awt.*;
import java.awt.event.*;

class DialogDemo extends Frame {
  String msg = "";
  CheckboxMenuItem debug, test;
  SampleDialog myDialog;

  public DialogDemo() {

    // Create the dialog box.
    myDialog  = new SampleDialog(this, "New Dialog Box");

    // Create menu bar and add it to frame.
    MenuBar mbar = new MenuBar();
    setMenuBar(mbar);

    // Create the menu items.
    Menu file = new Menu("File");
    MenuItem item1, item2, item3, item4, item5;
    file.add(item1 = new MenuItem("New..."));
    file.add(item2 = new MenuItem("Open..."));
    file.add(item3 = new MenuItem("Close"));
    file.add(item4 = new MenuItem("-"));
    file.add(item5 = new MenuItem("Quit..."));
    mbar.add(file);

    Menu edit = new Menu("Edit");
    MenuItem item6, item7, item8, item9;
    edit.add(item6 = new MenuItem("Cut"));
    edit.add(item7 = new MenuItem("Copy"));
    edit.add(item8 = new MenuItem("Paste"));
    edit.add(item9 = new MenuItem("-"));
```

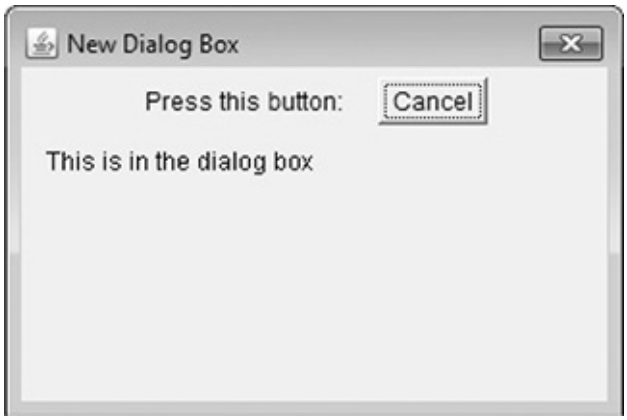
```
Menu sub = new Menu("Special");
MenuItem item10, item11, item12;
sub.add(item10 = new MenuItem("First"));
sub.add(item11 = new MenuItem("Second"));
sub.add(item12 = new MenuItem("Third"));
edit.add(sub);

// These are checkable menu items.
debug = new CheckboxMenuItem("Debug");
edit.add(debug);
test = new CheckboxMenuItem("Testing");
edit.add(test);

mbar.add(edit);

// Create an object to handle action and item events.
MyMenuHandler handler = new MyMenuHandler();

// Register to receive those events.
item1.addActionListener(handler);
item2.addActionListener(handler);
item3.addActionListener(handler);
item4.addActionListener(handler);
item6.addActionListener(handler);
item7.addActionListener(handler);
item8.addActionListener(handler);
item9.addActionListener(handler);
item10.addActionListener(handler);
item11.addActionListener(handler);
item12.addActionListener(handler);
debug.addItemListener(handler);
test.addItemListener(handler);
```

```
  // Use a lambda expression to handle the Quit selection.
  item5.addActionListener((ae) -> System.exit(0));

  addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent we) {
      System.exit(0);
    }
  });
}

public void paint(Graphics g) {
  g.drawString(msg, 10, 220);

 if(debug.getState())
    g.drawString("Debug is on.", 10, 240);
  else
    g.drawString("Debug is off.", 10, 240);

  if(test.getState())
    g.drawString("Testing is on.", 10, 260);
  else
    g.drawString("Testing is off.", 10, 260);
}

public static void main(String[] args) {
  DialogDemo appwin = new DialogDemo();

  appwin.setSize(new Dimension(250, 300));
  appwin.setTitle("DialogDemo");
  appwin.setVisible(true);
}
```

```
// An inner class for handling action and item events
// for the menu.
class MyMenuHandler implements ActionListener, ItemListener {

  // Handle action events.
  public void actionPerformed(ActionEvent ae) {
      msg = "You selected ";
      String arg = ae.getActionCommand();

     if(arg.equals("New...")) {
        msg += "New.";
        myDialog.setVisible(true);
      }
      else if(arg.equals("Open..."))
        msg += "Open.";
      else if(arg.equals("Close"))
        msg += "Close.";
      else if(arg.equals("Edit"))
        msg += "Edit.";
      else if(arg.equals("Cut"))
        msg += "Cut.";
      else if(arg.equals("Copy"))
        msg += "Copy.";
      else if(arg.equals("Paste"))
        msg += "Paste.";
      else if(arg.equals("First"))
        msg += "First.";
      else if(arg.equals("Second"))
        msg += "Second.";
      else if(arg.equals("Third"))
        msg += "Third.";
      else if(arg.equals("Debug"))
        msg += "Debug.";
      else if(arg.equals("Testing"))
        msg += "Testing.";
```

```
        repaint();
      }

      // Handle item events.
      public void itemStateChanged(ItemEvent ie) {
        repaint();
      }
    }
}

// Create a subclass of Dialog.
class SampleDialog extends Dialog {
  SampleDialog(Frame parent, String title) {
    super(parent, title, false);
    setLayout(new FlowLayout());
    setSize(300, 200);

    add(new Label("Press this button:"));

    Button b;
    add(b = new Button("Cancel"));
    b.addActionListener((ae) -> dispose());

    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        dispose();
      }
    });
  }

  public void paint(Graphics g) {
    g.drawString("This is in the dialog box", 20, 80);
  }
}
```

Here is sample output from the **DialogDemo** program:



TIP On your own, try defining dialog boxes for the other options presented by the menus.

A Word About Overriding paint( )

Before concluding our examination of AWT controls, a short word about overriding **paint( )** is in order. Although not relevant to the simple AWT examples shown in this book, when overriding **paint( )**, there are times when it is necessary to call the superclass implementation of **paint( )**. Therefore, for some programs, you will need to use this **paint( )** skeleton:

```
public void paint(Graphics g) {

  // code to repaint this window

  // Call superclass paint()
  super.paint(g);
}
```

In Java, there are two general types of components: heavyweight and lightweight. A heavyweight component has its own native window, which is called its *peer*. A lightweight component is implemented completely in Java code and uses the window provided by an ancestor. The AWT controls described and used in this chapter are all heavyweight. However, if a container holds any lightweight components (that is, has lightweight child components), your override of **paint( )** for that container must call **super.paint( )**. By calling **super.paint( )**, you ensure that any lightweight child components, such as lightweight controls, get properly painted. If you are unsure of a child component's type, you can call **isLightweight( )**, defined by **Component**, to find out. It returns **true** if the component is lightweight, and **false** otherwise.

CHAPTER

Images

This chapter examines the **Image** class and the **java.awt.image** package. Together, they provide support for *imaging* (the display and manipulation of graphical images). An *image* is simply a rectangular graphical object. Images are a key component of web design. In fact, the inclusion of the **<img>** tag in the Mosaic browser at NCSA (National Center for Supercomputer Applications) was a catalyst that helped the Web begin to grow explosively in 1993. This tag was used to include an image *inline* with the flow of hypertext. Java expands upon this basic concept, allowing images to be managed under program control. Because of its importance, Java provides extensive support for imaging.

Images are supported by the **Image** class, which is part of the **java.awt** package. Images are manipulated using the classes found in the **java.awt.image** package. There are a large number of imaging classes and interfaces defined by **java.awt.image**, and it is not possible to examine them all. Instead, we will focus on those that form the foundation of imaging. Here are the **java.awt.image** classes discussed in this chapter:

CropImageFilter	MemoryImageSource
FilteredImageSource	PixelGrabber
ImageFilter	RGBImageFilter

The interfaces that we will use are **ImageConsumer** and **ImageProducer**.

File Formats

Originally, web images could only be in GIF format. The GIF image format was created by CompuServe in 1987 to make it possible for images to be viewed while online, so it was well suited to the Internet. GIF images can have only up to 256 colors each. This limitation caused the major browser vendors to add support for JPEG images in 1995. The JPEG format was created by a group of photographic experts to store full-color-spectrum, continuous-tone images. These images, when properly created, can be of much higher fidelity as well as more highly compressed than a GIF encoding of the same source image. Another

file format is PNG. It too is an alternative to GIF. In almost all cases, you will never care or notice which format is being used in your programs. The Java image classes abstract the differences behind a clean interface.

Image Fundamentals: Creating, Loading, and Displaying

There are three common operations that occur when you work with images: creating an image, loading an image, and displaying an image. In Java, the **Image** class is used to refer to images in memory and to images that must be loaded from external sources. Thus, Java provides ways for you to create a new image object and ways to load one. It also provides a means by which an image can be displayed. Let's look at each.

Creating an Image Object

You might expect that you create a memory image using something like the following:

```
Image test = new Image(200, 100); // Error -- won't work
```

Not so. Because images must eventually be painted on a window to be seen, the **Image** class doesn't have enough information about its environment to create the proper data format for the screen. Therefore, the **Component** class in **java.awt** has a factory method called **createImage( )** that is used to create **Image** objects. (Remember that all of the AWT components are subclasses of **Component**, so all support this method.)

The **createImage( )** method has the following two forms:

Image createImage(ImageProducer *imgProd*)
Image createImage(int *width*, int *height*)

The first form returns an image produced by *imgProd*, which is an object of a class that implements the **ImageProducer** interface. (We will look at image producers later.) The second form returns a blank (that is, empty) image that has the specified width and height. Here is an example:

```
Canvas c = new Canvas();
Image test = c.createImage(200, 100);
```

This creates an instance of **Canvas** and then calls the **createImage( )** method to actually make an **Image** object. At this point, the image is blank. Later, you will see how to write data to it.

Loading an Image

Another way to obtain an image is to load one, either from a file on the local file system or from a URL. Here, we will use the local file system. The easiest way to load an image is to use one of the static methods defined by the **ImageIO** class. **ImageIO** provides extensive support for reading and writing images. It is packaged in **javax.imageio**, and beginning with JDK 9, **javax.imageio** is part of the **java.desktop** module. The method that loads an image is called **read( )**. The form we will use is shown here:

static BufferedImage read(File *imageFile*) throws IOException

Here, *imageFile* specifies the file that contains the image. It returns a reference to the image in the form of a **BufferedImage**, which is a subclass of **Image** that includes a buffer. Null is returned if the file does not contain a valid image.

Displaying an Image

Once you have an image, you can display it by using **drawImage( )**, which is a member of the **Graphics** class. It has several forms. The one we will be using is shown here:

boolean drawImage(Image *imgObj*, int *left*, int *top*, ImageObserver *imgOb*)

This displays the image passed in *imgObj* with its upper-left corner specified by *left* and *top*. *imgOb* is a reference to a class that implements the **ImageObserver** interface. This interface is implemented by all AWT (and Swing) components. An *image observer* is an object that can monitor an image while it loads. When no image observer is needed, *imgOb* can be **null**.

Using **read( )** and **drawImage( )**, it is actually quite easy to load and display an image. Here is a program that loads and displays a single image. The file **Lilies.jpg** is loaded, but you can substitute any image you like (just make sure it is available in the same directory as the program). Sample output is shown in Figure 27-1.

Figure 27-1 Sample output from **SimpleImageLoad**

```
// Load and display an image.
import java.awt.*;
import java.awt.event.*;
import javax.imageio.*;
import java.io.*;

public class SimpleImageLoad extends Frame {
  Image img;

  public SimpleImageLoad() {
```

```
    try {
      File imageFile = new File("Lilies.jpg");

      // Load the image.
      img = ImageIO.read(imageFile);
    } catch (IOException exc) {
      System.out.println("Cannot load image file.");
      System.exit(0);
    }

    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }

  public void paint(Graphics g) {
    g.drawImage(img, getInsets().left, getInsets().top, null);
  }

  public static void main(String[] args) {
    SimpleImageLoad appwin = new SimpleImageLoad();

    appwin.setSize(new Dimension(400, 365));
    appwin.setTitle("SimpleImageLoad");
    appwin.setVisible(true);
  }
}
```

Double Buffering

Not only are images useful for storing pictures, as we've just shown, but you can also use them as offscreen drawing surfaces. This allows you to render any image, including text and graphics, to an offscreen buffer that you can display at a later time. The advantage to doing this is that the image is seen only when it is complete. Drawing a complicated image could take several milliseconds or more, which can be seen by the user as flashing or flickering. This flashing is distracting and causes the user to perceive your rendering as slower than it actually is. Use of an offscreen image to reduce flicker is called *double buffering*, because the screen is considered a buffer for pixels, and the offscreen image is the second buffer, where you can prepare pixels for display.

Earlier in this chapter, you saw how to create a blank **Image** object. Now you will see how to draw on that image rather than the screen. As you recall from earlier chapters, you need a **Graphics** object in order to use any of Java's rendering methods. Conveniently, the **Graphics** object that you can use to draw on an **Image** is available via the **getGraphics( )** method. Here is a code fragment that creates a new image, obtains its graphics context, and fills the entire image with red pixels:

```
Canvas c = new Canvas();
Image test = c.createImage(200, 100);
Graphics gc = test.getGraphics();
gc.setColor(Color.red);
gc.fillRect(0, 0, 200, 100);
```

Once you have constructed and filled an offscreen image, it will still not be visible.
To actually display the image, call **drawImage( )**. Here is an example that draws a time-consuming image to demonstrate the difference that double buffering can make in
perceived drawing time:

```
// Demonstrate the use of an off-screen buffer.
import java.awt.*;
import java.awt.event.*;

public class DoubleBuffer extends Frame {
  int gap = 3;
  int mx, my;
  boolean flicker = true;
  Image buffer = null;
  int w = 400, h = 400;

  public DoubleBuffer() {
    addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseDragged(MouseEvent me) {
        mx = me.getX();
        my = me.getY();
        flicker = false;
        repaint();
      }
      public void mouseMoved(MouseEvent me) {
        mx = me.getX();
        my = me.getY();
        flicker = true;
        repaint();
      }
    });

    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }

  public void paint(Graphics g) {
    Graphics screengc = null;

    if (!flicker) {
      screengc = g;
      g = buffer.getGraphics();
    }

    g.setColor(Color.blue);
    g.fillRect(0, 0, w, h);
```

```
    g.setColor(Color.red);
    for (int i=0; i<w; i+=gap)
      g.drawLine(i, 0, w-i, h);
    for (int i=0; i<h; i+=gap)
      g.drawLine(0, i, w, h-i);

    g.setColor(Color.black);
    g.drawString("Press mouse button to double buffer", 10, h/2);

    g.setColor(Color.yellow);
    g.fillOval(mx - gap, my - gap, gap*2+1, gap*2+1);

    if (!flicker) {
      screengc.drawImage(buffer, 0, 0, null);
    }
  }

  public void update(Graphics g) {
    paint(g);
  }

  public static void main(String[] args) {
    DoubleBuffer appwin = new DoubleBuffer();

    appwin.setSize(new Dimension(400, 400));
    appwin.setTitle("DoubleBuffer");
    appwin.setVisible(true);

    // Create an off-screen buffer.
    appwin.buffer = appwin.createImage(appwin.w, appwin.h);
  }
}
```

This simple program has a complicated **paint( )** method. It fills the background with blue and then draws a red moiré pattern on top of that. It paints some black text on top of that and then paints a yellow circle centered at the coordinates **mx**, **my**. The **mouseMoved( )** and **mouseDragged( )** methods are overridden to track the mouse position. These methods are identical, except for the setting of the

flicker Boolean variable. **mouseMoved( )** sets **flicker** to **true**, and **mouseDragged( )** sets it to **false**. This has the effect of calling **repaint( )** with **flicker** set to **true** when the mouse is moved (but no button is pressed) and set to **false** when the mouse is dragged with any button pressed.

When **paint( )** gets called with **flicker** set to **true**, we see each drawing operation as it is executed on the screen. In the case where a mouse button is pressed and **paint( )** is called with **flicker** set to **false**, we see quite a different picture. The **paint( )** method swaps the **Graphics** reference **g** with the graphics context that refers to the offscreen canvas, **buffer**, which we created in **main( )**. Then all of the drawing operations are invisible. At the end of **paint( )**, we simply call **drawImage( )** to show the results of these drawing methods all at once.

Sample output is shown in Figure 27-2. The left snapshot is what the screen looks like with the mouse button not pressed. As you can see, the image was in the middle of repainting when this snapshot was taken. The right snapshot shows how, when a mouse button is pressed, the image is always complete and clean due to double buffering.

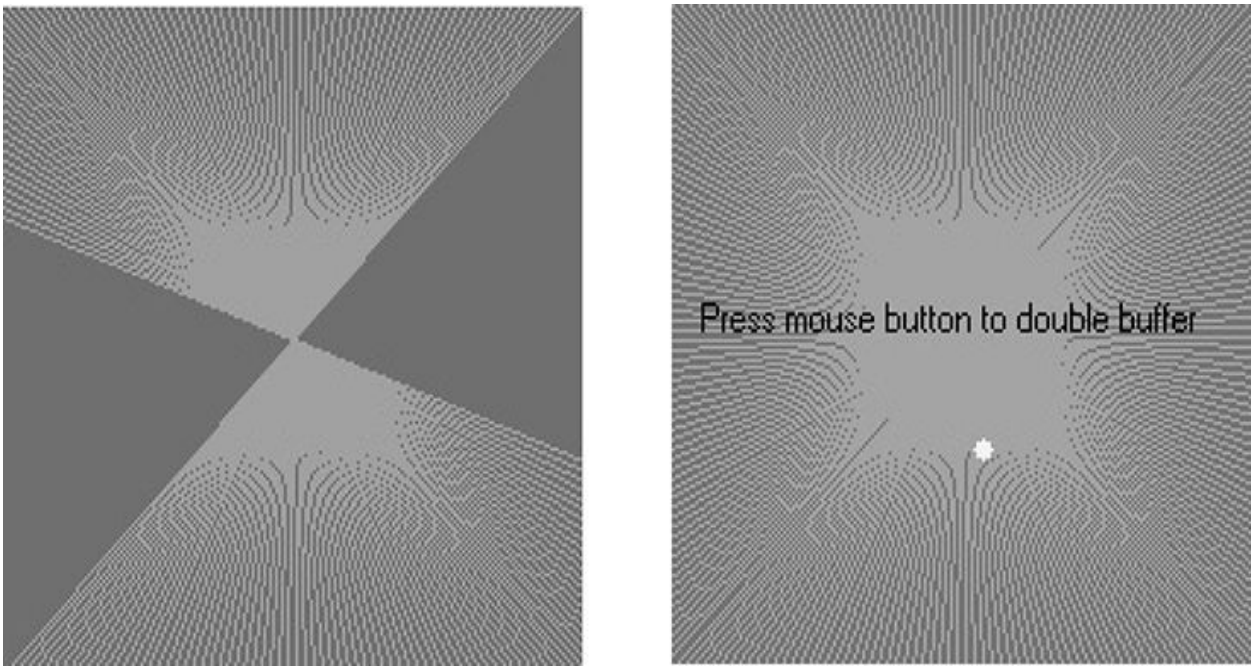


Figure 27-2 Output from **DoubleBuffer** without (left) and with (right) double buffering

ImageProducer

ImageProducer is an interface for objects that want to produce data for images. An object that implements the **ImageProducer** interface will supply integer or byte arrays that represent image data and produce **Image** objects. As you saw earlier, one form of the **createImage( )** method takes an **ImageProducer** object as its argument. There are two image producers contained in **java.awt.image**: **MemoryImageSource** and **FilteredImageSource**. Here, we will examine **MemoryImageSource** and create a new **Image** object from generated data.

MemoryImageSource

MemoryImageSource is a class that creates a new **Image** from an array of data. It defines several constructors. Here is the one we will be using:

MemoryImageSource(int *width*, int *height*, int *pixel*[ ], int *offset*, int *scanLineWidth*)

The **MemoryImageSource** object is constructed out of the array of integers specified by *pixel*, in the default RGB color model to produce data for an **Image** object. In the default color model, a pixel is an integer with Alpha, Red, Green, and Blue (0xAARRGGBB). The Alpha value represents a degree of transparency for the pixel. Fully transparent is 0 and fully opaque is 255. The width and height of the resulting image are passed in *width* and *height*. The starting point in the pixel array to begin reading data is passed in *offset*. The width of a scan line (which is often the same as the width of the image) is passed in *scanLineWidth*.

The following short example generates a **MemoryImageSource** object using a variation on a simple algorithm (a bitwise-exclusive-OR of the x and y address of each pixel) from the book *Beyond Photography: The Digital Darkroom* by Gerard J. Holzmann (Prentice Hall, 1988).

```
// Create an image in memory.
import java.awt.*;
import java.awt.image.*;
import java.awt.event.*;

public class MemoryImageGenerator extends Frame {
  Image img;
  int w = 512;
  int h = 512;

  public MemoryImageGenerator() {
    int pixels[] = new int[w * h];
    int i = 0;

    for(int y=0; y<h; y++) {
      for(int x=0; x<w; x++) {
        int r = (x^y)&0xff;
        int g = (x*2^y*2)&0xff;
        int b = (x*4^y*4)&0xff;
        pixels[i++] = (255 << 24) | (r << 16) | (g << 8) | b;
      }
    }
    img = createImage(new MemoryImageSource(w, h, pixels, 0, w));

    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }

  public void paint(Graphics g) {
    g.drawImage(img, getInsets().left, getInsets().top, null);
  }

  public static void main(String[] args) {
    MemoryImageGenerator appwin = new MemoryImageGenerator();

    appwin.setSize(new Dimension(400, 400));
    appwin.setTitle("MemoryImageGenerator");
    appwin.setVisible(true);
  }
}
```

The data for the new **MemoryImageSource** is created in the constructor. An array of integers is created to hold the pixel values; the data is generated in the nested **for** loops where the **r**, **g**, and **b** values get shifted into a pixel in the **pixels** array. Finally, **createImage( )** is called with a new instance of a **MemoryImageSource** created from the raw pixel data as its parameter. Figure 27-3 shows the image.

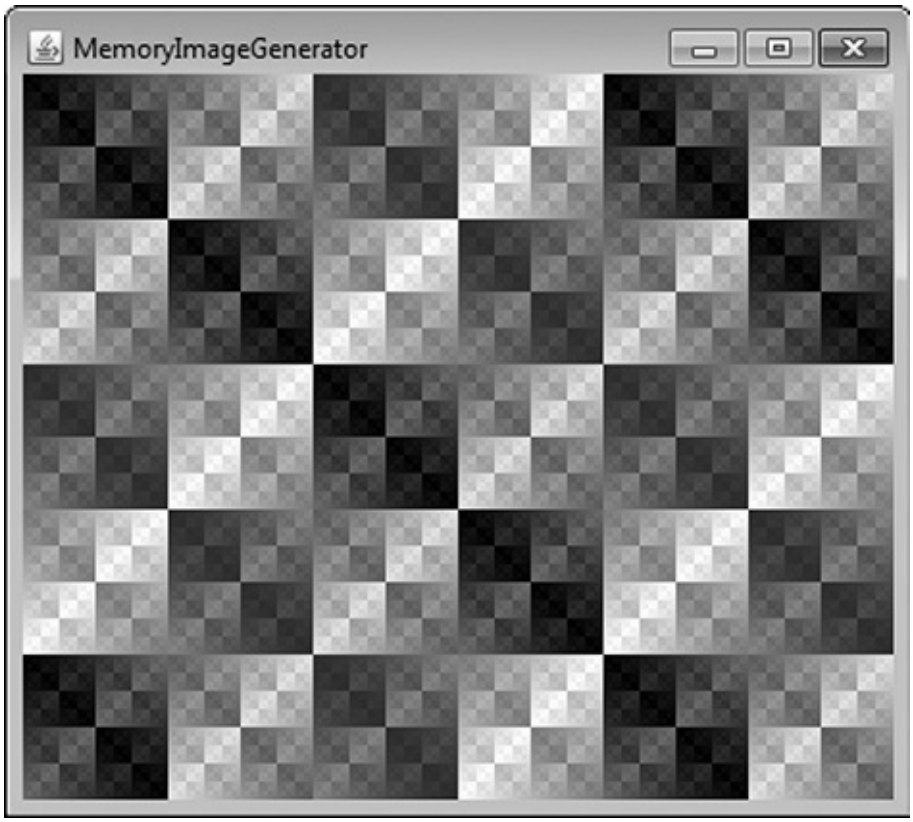


Figure 27-3 Sample output from **MemoryImageGenerator**

ImageConsumer

ImageConsumer is an interface for objects that want to take pixel data from images and supply it as another kind of data. This, obviously, is the opposite of **ImageProducer**, described earlier. An object that implements the **ImageConsumer** interface is going to create **int** or **byte** arrays that represent pixels from an **Image** object. We will examine the **PixelGrabber** class, which is a simple implementation of the **ImageConsumer** interface.

PixelGrabber

The **PixelGrabber** class is defined within **java.lang.image**. It is the inverse of the **MemoryImageSource** class. Rather than constructing an image from an array of pixel values, it takes an existing image and *grabs* the pixel array from it. To use **PixelGrabber**, you first create an array of **int**s big enough to hold the pixel data, and then you create a **PixelGrabber** instance passing in the rectangle that you want to grab. Finally, you call **grabPixels( )** on that instance.

The **PixelGrabber** constructor that is used in this chapter is shown here:

PixelGrabber(Image *imgObj*, int *left*, int *top*, int *width*, int *height*, int *pixel* [ ],
int *offset*, int *scanLineWidth*)

Here, *imgObj* is the object whose pixels are being grabbed. The values of *left* and *top* specify the upper-left corner of the rectangle, and *width* and *height* specify the dimensions of the rectangle from which the pixels will be obtained. The pixels will be stored in *pixel* beginning at *offset*. The width of a scan line (which is often the same as the width of the image) is passed in *scanLineWidth*.

grabPixels( ) is defined like this:

boolean grabPixels( )
throws InterruptedException

boolean grabPixels(long *milliseconds*)
throws InterruptedException

Both methods return **true** if successful and **false** otherwise. In the second form, *milliseconds* specifies how long the method will wait for the pixels. Both throw **InterruptedException** if execution is interrupted by another thread.

Here is an example that grabs the pixels from an image and then creates a histogram of pixel brightness. The *histogram* is simply a count of pixels that are a certain brightness for all brightness settings between 0 and 255. After the program paints the image, it draws the histogram over the top.

```
// Demonstrate PixelGraber.
import java.awt.* ;
import java.awt.event.*;
import java.awt.image.* ;
import javax.imageio.*;
import java.io.*;

public class HistoGrab extends Frame {
  Dimension d;
  Image img;
  int iw, ih;
  int pixels[];
  int hist[] = new int[256];
  int max_hist = 0;
  Insets ins;

  public HistoGrab() {

    try {
      File imageFile = new File("Lilies.jpg");

      // Load the image.
      img = ImageIO.read(imageFile);

      iw = img.getWidth(null);
      ih = img.getHeight(null);
      pixels = new int[iw * ih];
      PixelGrabber pg = new PixelGrabber(img, 0, 0, iw, ih,
                                         pixels, 0, iw);
      pg.grabPixels();
    } catch (InterruptedException e) {
      System.out.println("Interrupted");
      return;
    } catch (IOException exc) {
```

```
      System.out.println("Cannot load image file.");
      System.exit(0);
    }

    for (int i=0; i<iw*ih; i++) {
      int p = pixels[i];
      int r = 0xff & (p >> 16);
      int g = 0xff & (p >> 8);
      int b = 0xff & (p);
      int y = (int) (.33 * r + .56 * g + .11 * b);
      hist[y]++;
    }
    for (int i=0; i<256; i++) {
      if (hist[i] > max_hist)
        max_hist = hist[i];
    }

    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }

  public void paint(Graphics g) {
    // Get the border/header insets.
    ins = getInsets();

    g.drawImage(img, ins.left, ins.top, null);

    int x = (iw - 256) / 2;
    int lasty = ih - ih * hist[0] / max_hist;

    for (int i=0; i<256; i++, x++) {
      int y = ih - ih * hist[i] / max_hist;
      g.setColor(new Color(i, i, i));
      g.fillRect(x+ins.left, y+ins.top, 1, ih-y);
      g.setColor(Color.red);
      g.drawLine((x-1)+ins.left,lasty+ins.top,x+ins.left,y+ins.top);
      lasty = y;
    }
  }

  public static void main(String[] args) {
    HistoGrab appwin = new HistoGrab();

    appwin.setSize(new Dimension(400, 380));
    appwin.setTitle("HistoGrab");
    appwin.setVisible(true);
  }
}
```

Figure 27-4 shows an example image and its histogram.

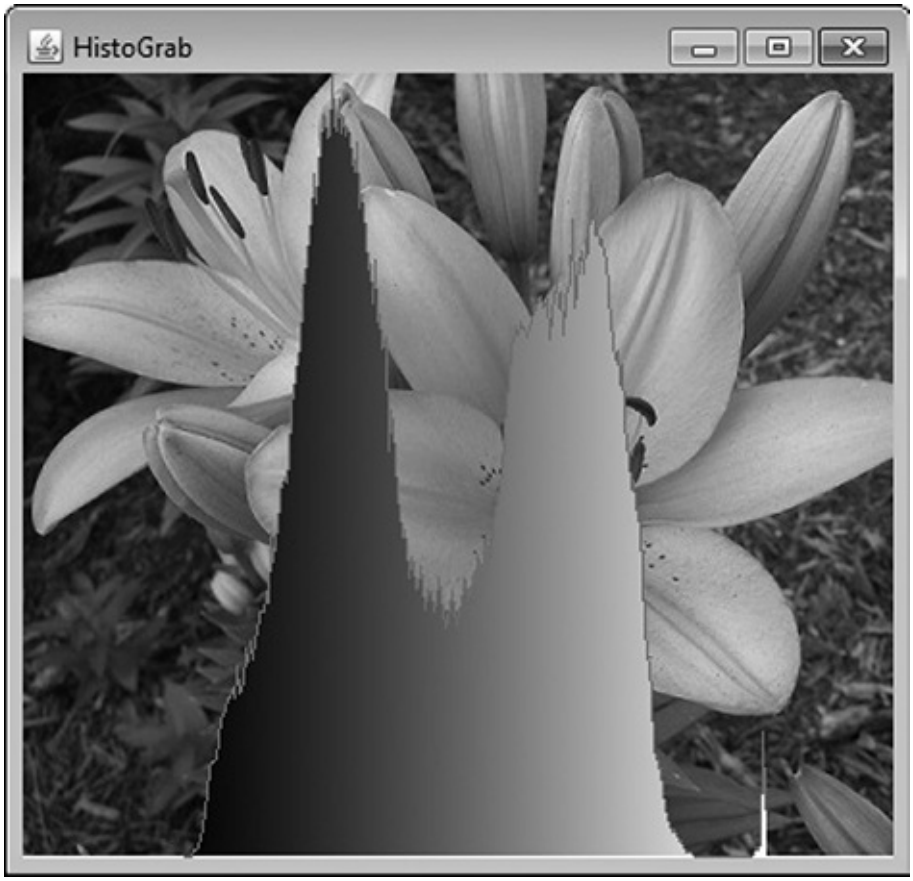


Figure 27-4 Sample output from **HistoGrab**

ImageFilter

Given the **ImageProducer** and **ImageConsumer** interface pair—and their concrete classes **MemoryImageSource** and **PixelGrabber**—you can create an arbitrary set of translation filters that takes a source of pixels, modifies them, and passes them on to an arbitrary consumer. This mechanism is analogous to the way concrete classes are created from the abstract I/O classes **InputStream**, **OutputStream**, **Reader**, and **Writer** (described in Chapter 21). This stream model for images is completed by the introduction of the **ImageFilter** class. Some subclasses of **ImageFilter** in the **java.awt.image** package are **AreaAveragingScaleFilter**, **CropImageFilter**, **ReplicateScaleFilter**, and **RGBImageFilter**. There is also an implementation of **ImageProducer** called **FilteredImageSource**, which takes an arbitrary **ImageFilter** and wraps it

around an **ImageProducer** to filter the pixels it produces. An instance of **FilteredImageSource** can be used as an **ImageProducer** in calls to **createImage( )**, in much the same way that **BufferedInputStream**s can be used as **InputStream**s.

In this chapter, we examine two filters: **CropImageFilter** and **RGBImageFilter**.

CropImageFilter

CropImageFilter filters an image source to extract a rectangular region. One situation in which this filter is valuable is where you want to use several small images from a single, larger source image. Loading twenty 2K images takes much longer than loading a single 40K image that has many frames of an animation tiled into it. If every subimage is the same size, then you can easily extract these images by using **CropImageFilter** to disassemble the block once your program starts. Here is an example that creates 16 images taken from a single image. The tiles are then scrambled by swapping a random pair from the 16 images 32 times.

```
// Demonstrate CropImageFilter.
import java.awt.*;
import java.awt.image.*;
import java.awt.event.*;
import javax.imageio.*;
import java.io.*;

public class TileImage extends Frame {
  Image img;
  Image cell[] = new Image[4*4];
  int iw, ih;
  int tw, th;

  public TileImage() {
    try {
      File imageFile = new File("Lilies.jpg");

      // Load the image.
      img = ImageIO.read(imageFile);

      iw = img.getWidth(null);
      ih = img.getHeight(null);
      tw = iw / 4;
      th = ih / 4;

      CropImageFilter f;
      FilteredImageSource fis;

      for (int y=0; y<4; y++) {
        for (int x=0; x<4; x++) {
          f = new CropImageFilter(tw*x, th*y, tw, th);
          fis = new FilteredImageSource(img.getSource(), f);
          int i = y*4+x;
          cell[i] = createImage(fis);
        }
      }

      for (int i=0; i<32; i++) {
        int si = (int)(Math.random() * 16);
        int di = (int)(Math.random() * 16);
        Image tmp = cell[si];
        cell[si] = cell[di];
        cell[di] = tmp;
      }
    } catch (IOException exc) {
      System.out.println("Cannot load image file.");
      System.exit(0);
    }

    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }
```

```
  public void paint(Graphics g) {
    for (int y=0; y<4; y++) {
      for (int x=0; x<4; x++) {
        g.drawImage(cell[y*4+x], x * tw + getInsets().left,
                    y * th + getInsets().top, null);
      }
    }
  }

  public static void main(String[] args) {
    TileImage appwin = new TileImage();

    appwin.setSize(new Dimension(420, 420));
    appwin.setTitle("TileImage");
    appwin.setVisible(true);
  }
}
```

Figure 27-5 shows the flowers image scrambled by **TileImage**.

Figure 27-5 Sample output from **TileImage**

RGBImageFilter

The **RGBImageFilter** is used to convert one image to another, pixel by pixel, transforming the colors along the way. This filter could be used to brighten an image, to increase its contrast, or even to convert it to grayscale.

To demonstrate **RGBImageFilter**, we have developed a somewhat complicated example that employs a dynamic plug-in strategy for image-processing filters. We've created an interface for generalized image filtering so that a program can simply load these filters at run time without having to know about all of the **ImageFilter**s in advance. This example consists of the main class called **ImageFilterDemo**, the interface called **PlugInFilter**, and a utility class called **LoadedImage**. Also included are three filters—**Grayscale**, **Invert**, and **Contrast**—which simply manipulate the color space of the source image using **RGBImageFilter**s, and two more classes—**Blur** and **Sharpen**—which do more complicated "convolution" filters that change pixel data based on the pixels surrounding each pixel of source data. **Blur** and **Sharpen** are subclasses of an abstract helper class called **Convolver**. Let's look at each part of our example.

ImageFilterDemo.java

The **ImageFilterDemo** class is the main class for the sample image filters. It employs the default **BorderLayout**, with a **Panel** at the *South* position to hold the buttons that will represent each filter. A **Label** object occupies the *North* slot for informational messages about filter progress. The *Center* is where the image (which is encapsulated in the **LoadedImage Canvas** subclass, described later) is put.

The **actionPerformed( )** method is interesting because it uses the label from a button as the name of a filter class that it loads. This method is robust and takes appropriate action if the button does not correspond to a proper class that implements **PlugInFilter**.

```
// Demonstrate image filters.
import java.awt.*;
import java.awt.event.*;
import javax.imageio.*;
import java.io.*;
import java.lang.reflect.*;

public class ImageFilterDemo extends Frame implements ActionListener {
  Image img;
  PlugInFilter pif;
  Image fimg;
  Image curImg;
  LoadedImage lim;
  Label lab;
  Button reset;

  // Names of the filters.
  String[] filters = { "Grayscale", "Invert", "Contrast",
                       "Blur", "Sharpen" };

  public ImageFilterDemo() {
    Panel p = new Panel();
    add(p, BorderLayout.SOUTH);

    // Create Reset button.
    reset = new Button("Reset");
    reset.addActionListener(this);
    p.add(reset);

    // Add the filter buttons.
    for(String fstr: filters) {
      Button b = new Button(fstr);
      b.addActionListener(this);
      p.add(b);
    }
```

```
    // Create the top label.
    lab = new Label("");
    add(lab, BorderLayout.NORTH);

    // Load the image.
    try {
      File imageFile = new File("Lilies.jpg");

      // Load the image.
      img = ImageIO.read(imageFile);
    } catch (IOException exc) {
      System.out.println("Cannot load image file.");
      System.exit(0);
    }

    // Get a LoadedImage and add it to the center.
    lim = new LoadedImage(img);
    add(lim, BorderLayout.CENTER);

    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }

  public void actionPerformed(ActionEvent ae) {
    String a = "";

    try {
      a = ae.getActionCommand();
      if (a.equals("Reset")) {
        lim.set(img);
        lab.setText("Normal");
      }
      else {
        // Get the selected filter.
        pif = (PlugInFilter)
                (Class.forName(a)).getConstructor().newInstance();
        fimg = pif.filter(this, img);
        lim.set(fimg);
        lab.setText("Filtered: " + a);
      }
      repaint();
    } catch (ClassNotFoundException e) {
      lab.setText(a + " not found");
      lim.set(img);
      repaint();
    } catch (InstantiationException e) {
      lab.setText("couldn't new " + a);
    } catch (IllegalAccessException e) {
```

```
      lab.setText("no access: " + a);
    } catch (NoSuchMethodException | InvocationTargetException e) {
      lab.setText("Filter creation error: " + e);
    }
  }

  public static void main(String[] args) {
    ImageFilterDemo appwin = new ImageFilterDemo();

    appwin.setSize(new Dimension(420, 420));
    appwin.setTitle("ImageFilterDemo");
    appwin.setVisible(true);
  }
}
```

Figure 27-6 shows what the program looks like when it is first loaded.

Figure 27-6 Sample normal output from **ImageFilterDemo**

PlugInFilter.java

PlugInFilter is a simple interface used to abstract image filtering. It has only one method, **filter( )**, which takes the frame and the source image and returns a new image that has been filtered in some way.

```
interface PlugInFilter {
  java.awt.Image filter(java.awt.Frame f, java.awt.Image in);
}
```

LoadedImage.java

LoadedImage is a convenient subclass of **Canvas**. It behaves properly under layout manager control, because it overrides the **getPreferredSize( )** and **getMinimumSize( )** methods. Also, it has a method called **set( )** that can be used to set a new **Image** to be displayed in this **Canvas**. That is how the filtered image is displayed after the plug-in is finished.

```
import java.awt.*;

public class LoadedImage extends Canvas {
  Image img;

  public LoadedImage(Image i) {
    set(i);
  }

  void set(Image i) {
    img = i;
    repaint();
  }

  public void paint(Graphics g) {
    if (img == null) {
      g.drawString("no image", 10, 30);
    } else {
      g.drawImage(img, 0, 0, this);
    }
  }

  public Dimension getPreferredSize()  {
    return new Dimension(img.getWidth(this), img.getHeight(this));
  }

  public Dimension getMinimumSize()  {
    return getPreferredSize();
  }
}
```

Grayscale.java

The **Grayscale** filter is a subclass of **RGBImageFilter**, which means that **Grayscale** can use itself as the **ImageFilter** parameter to **FilteredImageSource**'s constructor. Then all it needs to do is override **filterRGB( )** to change the incoming color values. It takes the red, green, and blue values and computes the brightness of the pixel, using the NTSC (National Television Standards Committee) color-to-brightness conversion factor. It then simply returns a gray pixel that is the same brightness as the color source.

```
// Grayscale filter.
import java.awt.*;
import java.awt.image.*;

class Grayscale extends RGBImageFilter implements PlugInFilter {
  public Grayscale() {}

  public Image filter(Frame f, Image in) {
    return f.createImage(new FilteredImageSource(in.getSource(), this));
  }

  public int filterRGB(int x, int y, int rgb) {
    int r = (rgb >> 16) & 0xff;
    int g = (rgb >> 8) & 0xff;
    int b = rgb & 0xff;
    int k = (int) (.56 * g + .33 * r + .11 * b);
    return (0xff000000 | k << 16 | k << 8 | k);
  }
}
```

Invert.java

The **Invert** filter is also quite simple. It takes apart the red, green, and blue values and then inverts them by subtracting them from 255. These inverted values are packed back into a pixel value and returned.

```
// Invert colors filter.
import java.awt.*;
import java.awt.image.*;

class Invert extends RGBImageFilter implements PlugInFilter {
  public Invert() { }

  public Image filter(Frame f, Image in) {
    return f.createImage(new FilteredImageSource(in.getSource(), this));
  }

  public int filterRGB(int x, int y, int rgb) {
    int r = 0xff - (rgb >> 16) & 0xff;
    int g = 0xff - (rgb >> 8) & 0xff;
    int b = 0xff - rgb & 0xff;
    return (0xff000000 | r << 16 | g << 8 | b);
  }
}
```

Figure 27-7 shows the image after it has been run through the **Invert** filter.

Figure 27-7 Using the **Invert** filter with **ImageFilterDemo**

Contrast.java

The **Contrast** filter is very similar to **Grayscale**, except its override of **filterRGB( )** is slightly more complicated. The algorithm it uses for contrast enhancement takes the red, green, and blue values separately and boosts them by 1.2 times if they are already brighter than 128. If they are below 128, then they are divided by 1.2. The boosted values are properly clamped at 255 by the **multclamp( )** method.

```
// Contrast filter.
import java.awt.*;
import java.awt.image.*;
```

```
public class Contrast extends RGBImageFilter implements PlugInFilter {

  public Image filter(Frame f, Image in) {
    return f.createImage(new FilteredImageSource(in.getSource(), this));
  }

  private int multclamp(int in, double factor) {
    in = (int) (in * factor);
    return in > 255 ? 255 : in;
  }

  double gain = 1.2;
  private int cont(int in) {
    return (in < 128) ? (int)(in/gain) : multclamp(in, gain);
  }

  public int filterRGB(int x, int y, int rgb) {
    int r = cont((rgb >> 16) & 0xff);
    int g = cont((rgb >> 8) & 0xff);
    int b = cont(rgb & 0xff);
    return (0xff000000 | r << 16 | g << 8 | b);
  }
}
```

Figure 27-8 shows the image after **Contrast** is pressed.

Figure 27-8 Using the **Contrast** filter with **ImageFilterDemo**

Convolver.java

The abstract class **Convolver** handles the basics of a convolution filter by implementing the **ImageConsumer** interface to move the source pixels into an array called **imgpixels**. It also creates a second array called **newimgpixels** for the filtered data. Convolution filters sample a small rectangle of pixels around each pixel in an image, called the *convolution kernel*. This area, 3 × 3 pixels in this demo, is used to decide how to change the center pixel in the area.

NOTE The reason that the filter can't modify the **imgpixels** array in place is that the next pixel on a scan line would try to use the original value for the previous pixel, which would have just been filtered away.

The two concrete subclasses, shown in the next section, simply implement the **convolve( )** method, using **imgpixels** for source data and **newimgpixels** to store the result.

```
// Convolution filter.
import java.awt.*;
import java.awt.image.*;

abstract class Convolver implements ImageConsumer, PlugInFilter {
  int width, height;
  int imgpixels[], newimgpixels[];
  boolean imageReady = false;

  abstract void convolve();  // filter goes here...

  public Image filter(Frame f, Image in) {
    imageReady = false;
```

```
    in.getSource().startProduction(this);
    waitForImage();
    newimgpixels = new int[width*height];

    try {
      convolve();
    } catch (Exception e) {
      System.out.println("Convolver failed: " + e);
      e.printStackTrace();
    }

    return f.createImage(
      new MemoryImageSource(width, height, newimgpixels, 0, width));
  }

  synchronized void waitForImage() {
    try {
      while(!imageReady)
        wait();
    } catch (Exception e) {
      System.out.println("Interrupted");
    }
  }

  public void setProperties(java.util.Hashtable<?,?> dummy) { }
  public void setColorModel(ColorModel dummy) { }
  public void setHints(int dummy) { }

  public synchronized void imageComplete(int dummy) {
    imageReady = true;
    notifyAll();
  }

  public void setDimensions(int x, int y) {
    width = x;
    height = y;
    imgpixels = new int[x*y];
  }

  public void setPixels(int x1, int y1, int w, int h,
    ColorModel model, byte pixels[], int off, int scansize) {
    int pix, x, y, x2, y2, sx, sy;

    x2 = x1+w;
    y2 = y1+h;
    sy = off;
    for(y=y1; y<y2; y++) {
      sx = sy;
      for(x=x1; x<x2; x++) {
        pix = model.getRGB(pixels[sx++]);
        if((pix & 0xff000000) == 0)
            pix = 0x00ffffff;
```

```
            imgpixels[y*width+x] = pix;
          }
          sy += scansize;
        }
      }

      public void setPixels(int x1, int y1, int w, int h,
        ColorModel model, int pixels[], int off, int scansize) {
        int pix, x, y, x2, y2, sx, sy;

        x2 = x1+w;
        y2 = y1+h;
        sy = off;
        for(y=y1; y<y2; y++) {
          sx = sy;
          for(x=x1; x<x2; x++) {
            pix = model.getRGB(pixels[sx++]);
            if((pix & 0xff000000) == 0)
                pix = 0x00ffffff;
            imgpixels[y*width+x] = pix;
          }
          sy += scansize;
        }
      }
    }
```

NOTE A built-in convolution filter called **ConvolveOp** is provided by **java.awt.image**. You may want to explore its capabilities on your own.

Blur.java

The **Blur** filter is a subclass of **Convolver** and simply runs through every pixel in the source image array, **imgpixels**, and computes the average of the 3 × 3 box surrounding it. The corresponding output pixel in **newimgpixels** is that average value.

```
public class Blur extends Convolver {
  public void convolve() {
    for(int y=1; y<height-1; y++) {
      for(int x=1; x<width-1; x++) {
        int rs = 0;
        int gs = 0;
        int bs = 0;

        for(int k=-1; k<=1; k++) {
          for(int j=-1; j<=1; j++) {
            int rgb = imgpixels[(y+k)*width+x+j];
            int r = (rgb >> 16) & 0xff;
            int g = (rgb >> 8) & 0xff;
            int b = rgb & 0xff;
            rs += r;
            gs += g;
            bs += b;
          }
        }

        rs /= 9;
        gs /= 9;
        bs /= 9;

       newimgpixels[y*width+x] = (0xff000000 |
                                  rs << 16 | gs << 8 | bs);
      }
    }
  }
}
```

Figure 27-9 shows the image after **Blur**.

Figure 27-9 Using the **Blur** filter with **ImageFilterDemo**

Sharpen.java

The **Sharpen** filter is also a subclass of **Convolver** and is (more or less) the inverse of **Blur**. It runs through every pixel in the source image array, **imgpixels**, and computes the average of the 3 × 3 box surrounding it, not counting the center. The corresponding output pixel in **newimgpixels** has the difference between the center pixel and the surrounding average added to it. This basically says that if a pixel is 30 brighter than its surroundings, make it another 30 brighter. If, however, it is 10 darker, then make it another 10 darker. This tends to accentuate edges while leaving smooth areas unchanged.

```
public class Sharpen extends Convolver {

  private final int clamp(int c) {
    return (c > 255 ? 255 : (c < 0 ? 0 : c));
  }

  public void convolve() {
    int r0=0, g0=0, b0=0;

    for(int y=1; y<height-1; y++) {
      for(int x=1; x<width-1; x++) {
        int rs = 0;
        int gs = 0;
        int bs = 0;

        for(int k=-1; k<=1; k++) {
          for(int j=-1; j<=1; j++) {
            int rgb = imgpixels[(y+k)*width+x+j];
            int r = (rgb >> 16) & 0xff;
            int g = (rgb >> 8) & 0xff;
            int b = rgb & 0xff;
            if (j == 0 && k == 0) {
              r0 = r;
              g0 = g;
              b0 = b;
            } else {
              rs += r;
              gs += g;
              bs += b;
            }
          }
        }

        rs >>= 3;
        gs >>= 3;
        bs >>= 3;
        newimgpixels[y*width+x] = (0xff000000 |
                                   clamp(r0+r0-rs) << 16 |
                                   clamp(g0+g0-gs) << 8 |
                                   clamp(b0+b0-bs));
      }
    }
  }
}
```

Figure 27-10 shows the image after **Sharpen**.

Figure 27-10 Using the **Sharpen** filter with **ImageFilterDemo**

Additional Imaging Classes

In addition to the imaging classes described in this chapter, **java.awt.image** supplies several others that offer enhanced control over the imaging process and that support advanced imaging techniques. Also available is the imaging package called **javax.imageio**. It supports reading and writing images, and has plug-ins that handle various image formats. If sophisticated graphical output is of special interest to you, then you will want to explore the additional classes found in **java.awt.image** and **javax.imageio**.

CHAPTER

The Concurrency Utilities

From the start, Java has provided built-in support for multithreading and synchronization. For example, new threads can be created by implementing **Runnable** or by extending **Thread**; synchronization is available by use of the **synchronized** keyword; and interthread communication is supported by the **wait( )** and **notify( )** methods that are defined by **Object**. In general, this built-in support for multithreading was one of Java's most important innovations and is still one of its major strengths.

However, as conceptually pure as Java's original support for multithreading is, it is not ideal for all applications—especially those that make intensive use of multiple threads. For example, the original multithreading support does not provide several high-level features, such as semaphores, thread pools, and execution managers, that facilitate the creation of intensively concurrent programs.

It is important to explain at the outset that many Java programs make use of multithreading and are, therefore, "concurrent." However, as it is used in this chapter, the term *concurrent program* refers to a program that makes *extensive*, *integral* use of concurrently executing threads. An example of such a program is one that uses separate threads to simultaneously compute the partial results of a larger computation. Another example is a program that coordinates the activities of several threads, each of which seeks access to information in a database. In this case, read-only accesses might be handled differently from those that require read/write capabilities.

To begin to handle the needs of a concurrent program, JDK 5 added the *concurrency utilities,* also commonly referred to as the *concurrent API.* The original set of concurrency utilities supplied many features that had long been wanted by programmers who develop concurrent applications. For example, it offered synchronizers (such as the semaphore), thread pools, execution managers, locks, several concurrent collections, and a streamlined way to use threads to obtain computational results.

Although the original concurrent API was impressive in its own right, it was significantly expanded by JDK 7. The most important addition was the *Fork/Join Framework.* The Fork/Join Framework facilitates the creation of

programs that make use of multiple processors (such as those found in multicore systems). Thus, it streamlines the development of programs in which two or more pieces execute with true simultaneity (that is, true parallel execution), not just time-slicing. As you can easily imagine, parallel execution can dramatically increase the speed of certain operations. Because multicore systems are now commonplace, the inclusion of the Fork/Join Framework was as timely as it was powerful. With the release of JDK 8, the Fork/Join Framework was further enhanced.

Furthermore, both JDK 8 and JDK 9 have added features related to other parts of the concurrent API. Thus, the concurrent API continues to evolve and expand to meet the needs of the contemporary computing environment.

The original concurrent API was quite large, and the additions made over the years have increased its size substantially. As you might expect, many of the issues surrounding the concurrency utilities are quite complex. It is beyond the scope of this book to discuss all of its facets. The preceding notwithstanding, it is important for all programmers to have a general, working knowledge of key aspects of the concurrent API. Even in programs that are not intensively parallel, features such as synchronizers, callable threads, and executors, are applicable to a wide variety of situations. Perhaps most importantly, because of the rise of multicore computers, solutions involving the Fork/Join Framework are becoming more common. For these reasons, this chapter presents an overview of several core features defined by the concurrency utilities and shows a number of examples that demonstrate their use. It concludes with an introduction to the Fork/Join Framework.

The Concurrent API Packages

The concurrency utilities are contained in the **java.util.concurrent** package and in its two subpackages: **java.util.concurrent.atomic** and **java.util.concurrent.locks**. Beginning with JDK 9, all are in the **java.base** module. A brief overview of their contents is given here.

java.util.concurrent

java.util.concurrent defines the core features that support alternatives to the built-in approaches to synchronization and interthread communication. These include

- Synchronizers
- Executors
- Concurrent collections
- The Fork/Join Framework

Synchronizers offer high-level ways of synchronizing the interactions between multiple threads. The synchronizer classes defined by **java.util.concurrent** are

Semaphore	Implements the classic semaphore.
CountDownLatch	Waits until a specified number of events have occurred.
CyclicBarrier	Enables a group of threads to wait at a predefined execution point.
Exchanger	Exchanges data between two threads.
Phaser	Synchronizes threads that advance through multiple phases of an operation.

Notice that each synchronizer provides a solution to a specific type of synchronization problem. This enables each synchronizer to be optimized for its intended use. In the past, these types of synchronization objects had to be crafted by hand. The concurrent API standardizes them and makes them available to all Java programmers.

Executors manage thread execution. At the top of the executor hierarchy is the **Executor** interface, which is used to initiate a thread. **ExecutorService** extends **Executor** and provides methods that manage execution. There are three implementations of **ExecutorService**: **ThreadPoolExecutor**, **ScheduledThreadPoolExecutor**, and **ForkJoinPool**. **java.util.concurrent** also defines the **Executors** utility class, which includes a number of static methods that simplify the creation of various executors.

Related to executors are the **Future** and **Callable** interfaces. A **Future** contains a value that is returned by a thread after it executes. Thus, its value becomes defined "in the future," when the thread terminates. **Callable** defines a thread that returns a value.

java.util.concurrent defines several concurrent collection classes, including **ConcurrentHashMap**, **ConcurrentLinkedQueue**, and **CopyOnWriteArrayList**. These offer concurrent alternatives to their related classes defined by the Collections Framework.

The *Fork/Join Framework* supports parallel programming. Its main classes are **ForkJoinTask**, **ForkJoinPool**, **RecursiveTask**, and **RecursiveAction**.

To better handle thread timing, **java.util.concurrent** defines the **TimeUnit** enumeration.

Beginning with JDK 9, **java.util.concurrent** also includes a subsystem that offers a means by which the flow of data can be controlled. It is based on the **Flow** class and these nested interfaces: **Flow.Subscriber**, **Flow.Publisher**, **Flow.Processor**, and **Flow.Subscription**. Although a detailed discussion of the **Flow** subsystem is outside the focus of this chapter, here is a brief description. **Flow** and its nested interfaces support the *reactive streams* specification. This specification defines a means by which a consumer of data can prevent the producer of the data from overrunning the consumer's ability to process the data. In this approach, data is produced by a *publisher* and consumed by a *subscriber*. Control is achieved by implementing a form of *back pressure*.

java.util.concurrent.atomic

java.util.concurrent.atomic facilitates the use of variables in a concurrent environment. It provides a means of efficiently updating the value of a variable without the use of locks. This is accomplished through the use of classes, such as **AtomicInteger** and **AtomicLong**, and methods, such as **compareAndSet( )**, **decrementAndGet( )**, and **getAndSet( )**. These methods execute as a single, non-interruptible operation.

java.util.concurrent.locks

java.util.concurrent.locks provides an alternative to the use of synchronized methods. At the core of this alternative is the **Lock** interface, which defines the basic mechanism used to acquire and relinquish access to an object. The key methods are **lock( )**, **tryLock( )**, and **unlock( )**. The advantage to using these methods is greater control over synchronization.

The remainder of this chapter takes a closer look at the constituents of the concurrent API.

Using Synchronization Objects

Synchronization objects are supported by the **Semaphore**, **CountDownLatch**, **CyclicBarrier**, **Exchanger**, and **Phaser** classes. Collectively, they enable you to handle several formerly difficult synchronization situations with ease. They are also applicable to a wide range of programs—even those that contain only

limited concurrency. Because the synchronization objects will be of interest to nearly all Java programs, each is examined here in some detail.

Semaphore

The synchronization object that many readers will immediately recognize is **Semaphore**, which implements a classic semaphore. A semaphore controls access to a shared resource through the use of a counter. If the counter is greater than zero, then access is allowed. If it is zero, then access is denied. What the counter is counting are *permits* that allow access to the shared resource. Thus, to access the resource, a thread must be granted a permit from the semaphore.

In general, to use a semaphore, the thread that wants access to the shared resource tries to acquire a permit. If the semaphore's count is greater than zero, then the thread acquires a permit, which causes the semaphore's count to be decremented. Otherwise, the thread will be blocked until a permit can be acquired. When the thread no longer needs access to the shared resource, it releases the permit, which causes the semaphore's count to be incremented. If there is another thread waiting for a permit, then that thread will acquire a permit at that time. Java's **Semaphore** class implements this mechanism.

Semaphore has the two constructors shown here:

```
Semaphore(int num)
Semaphore(int num, boolean how)
```

Here, *num* specifies the initial permit count. Thus, *num* specifies the number of threads that can access a shared resource at any one time. If *num* is one, then only one thread can access the resource at any one time. By default, waiting threads are granted a permit in an undefined order. By setting *how* to **true**, you can ensure that waiting threads are granted a permit in the order in which they requested access.

To acquire a permit, call the **acquire( )** method, which has these two forms:

void acquire( ) throws InterruptedException
void acquire(int *num*) throws InterruptedException

The first form acquires one permit. The second form acquires *num* permits. Most often, the first form is used. If the permit cannot be granted at the time of the call, then the invoking thread suspends until the permit is available.

To release a permit, call **release( )**, which has these two forms:

```
void release( )
void release(int num)
```

The first form releases one permit. The second form releases the number of permits specified by *num.*

To use a semaphore to control access to a resource, each thread that wants to use that resource must first call **acquire( )** before accessing the resource. When the thread is done with the resource, it must call **release( )**. Here is an example that illustrates the use of a semaphore:

```
// A simple semaphore example.

import java.util.concurrent.*;

class SemDemo {

  public static void main(String args[]) {
    Semaphore sem = new Semaphore(1);

    new Thread(new IncThread(sem, "A")).start();
    new Thread(new DecThread(sem, "B")).start();

  }
}

// A shared resource.
class Shared {
  static int count = 0;
}

// A thread of execution that increments count.
class IncThread implements Runnable {
  String name;
  Semaphore sem;

  IncThread(Semaphore s, String n) {
    sem = s;
    name = n;
  }

  public void run() {

    System.out.println("Starting " + name);

    try {
      // First, get a permit.
      System.out.println(name + " is waiting for a permit.");
      sem.acquire();
      System.out.println(name + " gets a permit.");

      // Now, access shared resource.
      for(int i=0; i < 5; i++) {
        Shared.count++;
        System.out.println(name + ": " + Shared.count);

        // Now, allow a context switch -- if possible.
        Thread.sleep(10);
      }
```

```
    } catch (InterruptedException exc) {
      System.out.println(exc);
    }

    // Release the permit.
    System.out.println(name + " releases the permit.");
    sem.release();
  }
}

// A thread of execution that decrements count.
class DecThread implements Runnable {
  String name;
  Semaphore sem;

  DecThread(Semaphore s, String n) {
    sem = s;
    name = n;
  }

  public void run() {

    System.out.println("Starting " + name);

    try {
      // First, get a permit.
      System.out.println(name + " is waiting for a permit.");
      sem.acquire();
      System.out.println(name + " gets a permit.");

      // Now, access shared resource.
      for(int i=0; i < 5; i++) {
        Shared.count--;
        System.out.println(name + ": " + Shared.count);

        // Now, allow a context switch -- if possible.
        Thread.sleep(10);
      }
    } catch (InterruptedException exc) {
      System.out.println(exc);
    }

    // Release the permit.
    System.out.println(name + " releases the permit.");
    sem.release();
  }
}
```

The output from the program is shown here. (The precise order in which the threads execute may vary.)

```
Starting A
A is waiting for a permit.
A gets a permit.
A: 1
Starting B
B is waiting for a permit.
A: 2
A: 3
A: 4
A: 5
A releases the permit.
B gets a permit.
B: 4
B: 3
B: 2
B: 1
B: 0
B releases the permit.
```

The program uses a semaphore to control access to the **count** variable, which is a static variable within the **Shared** class. **Shared.count** is incremented five times by the **run( )** method of **IncThread** and decremented five times by **DecThread**. To prevent these two threads from accessing **Shared.count** at the same time, access is allowed only after a permit is acquired from the controlling semaphore. After access is complete, the permit is released. In this way, only one thread at a time will access **Shared.count**, as the output shows.

In both **IncThread** and **DecThread**, notice the call to **sleep( )** within **run( )**. It is used to "prove" that accesses to **Shared.count** are synchronized by the semaphore. In **run( )**, the call to **sleep( )** causes the invoking thread to pause between each access to **Shared.count**. This would normally enable the second thread to run. However, because of the semaphore, the second thread must wait until the first has released the permit, which happens only after all accesses by the first thread are complete. Thus, **Shared.count** is incremented five times by **IncThread** and decremented five times by **DecThread**. The increments and decrements are *not* intermixed.

Without the use of the semaphore, accesses to **Shared.count** by both threads would have occurred simultaneously, and the increments and decrements would be intermixed. To confirm this, try commenting out the calls to **acquire( )** and **release( )**. When you run the program, you will see that access to **Shared.count**

is no longer synchronized, and each thread accesses it as soon as it gets a timeslice.

Although many uses of a semaphore are as straightforward as that shown in the preceding program, more intriguing uses are also possible. Here is an example. The following program reworks the producer/consumer program shown in Chapter 11 so that it uses two semaphores to regulate the producer and consumer threads, ensuring that each call to **put( )** is followed by a corresponding call to **get( )**:

```
// An implementation of a producer and consumer
// that use semaphores to control synchronization.

import java.util.concurrent.Semaphore;

class Q {
  int n;
```

```
  // Start with consumer semaphore unavailable.
  static Semaphore semCon = new Semaphore(0);
  static Semaphore semProd = new Semaphore(1);

  void get() {
    try {
      semCon.acquire();
    } catch(InterruptedException e) {
      System.out.println("InterruptedException caught");
    }

    System.out.println("Got: " + n);
    semProd.release();
  }

  void put(int n) {
    try {
      semProd.acquire();
    } catch(InterruptedException e) {
      System.out.println("InterruptedException caught");
    }

    this.n = n;
    System.out.println("Put: " + n);
    semCon.release();
  }
}

class Producer implements Runnable {
  Q q;

  Producer(Q q) {
    this.q = q;
  }

  public void run() {
    for(int i=0; i < 20; i++) q.put(i);
  }
}

class Consumer implements Runnable {
  Q q;

  Consumer(Q q) {
    this.q = q;
  }

  public void run() {
    for(int i=0; i < 20; i++) q.get();
  }
}
```

```
class ProdCon {
  public static void main(String args[]) {
    Q q = new Q();
    new Thread(new Consumer(q), "Consumer").start();
    new Thread(new Producer(q), "Producer").start();
  }
}
```

A portion of the output is shown here:

```
Put: 0
Got: 0
Put: 1
Got: 1
Put: 2
Got: 2
Put: 3
Got: 3
Put: 4
Got: 4
Put: 5
Got: 5
.
.
.
```

As you can see, the calls to **put( )** and **get( )** are synchronized. That is, each call to **put( )** is followed by a call to **get( )** and no values are missed. Without the semaphores, multiple calls to **put( )** would have occurred without matching calls to **get( )**, resulting in values being missed. (To prove this, remove the semaphore code and observe the results.)

The sequencing of **put( )** and **get( )** calls is handled by two semaphores: **semProd** and **semCon**. Before **put( )** can produce a value, it must acquire a permit from **semProd**. After it has set the value, it releases **semCon**. Before **get( )** can consume a value, it must acquire a permit from **semCon**. After it consumes the value, it releases **semProd**. This “give and take” mechanism ensures that each call to **put( )** must be followed by a call to **get( )**.

Notice that **semCon** is initialized with no available permits. This ensures that **put( )** executes first. The ability to set the initial synchronization state is one of the more powerful aspects of a semaphore.

CountDownLatch

Sometimes you will want a thread to wait until one or more events have occurred. To handle such a situation, the concurrent API supplies **CountDownLatch**. A **CountDownLatch** is initially created with a count of the number of events that must occur before the latch is released. Each time an event happens, the count is decremented. When the count reaches zero, the latch opens.

CountDownLatch has the following constructor:

CountDownLatch(int *num*)

Here, *num* specifies the number of events that must occur in order for the latch to open.

To wait on the latch, a thread calls **await( )**, which has the forms shown here:

void await( ) throws InterruptedException
boolean await(long *wait*, TimeUnit *tu*) throws InterruptedException

The first form waits until the count associated with the invoking **CountDownLatch** reaches zero. The second form waits only for the period of time specified by *wait*. The units represented by *wait* are specified by *tu*, which is an object the **TimeUnit** enumeration. (**TimeUnit** is described later in this chapter.) It returns **false** if the time limit is reached and **true** if the countdown reaches zero.

To signal an event, call the **countDown( )** method, shown next:

void countDown( )

Each call to **countDown( )** decrements the count associated with the invoking object.

The following program demonstrates **CountDownLatch**. It creates a latch that requires five events to occur before it opens.

```
// An example of CountDownLatch.

import java.util.concurrent.CountDownLatch;

class CDLDemo {
  public static void main(String args[]) {
    CountDownLatch cdl = new CountDownLatch(5);

    System.out.println("Starting");

    new Thread(new MyThread(cdl)).start();

    try {
      cdl.await();
    } catch (InterruptedException exc) {
      System.out.println(exc);
    }

    System.out.println("Done");
  }
}

class MyThread implements Runnable {
  CountDownLatch latch;

  MyThread(CountDownLatch c) {
    latch = c;
  }

  public void run() {
    for(int i = 0; i<5; i++) {
      System.out.println(i);
      latch.countDown(); // decrement count
    }
  }
}
```

The output produced by the program is shown here:

```
Starting
0
1
2
3
```

```
4
Done
```

Inside **main( )**, a **CountDownLatch** called **cdl** is created with an initial count of five. Next, an instance of **MyThread** is created, which begins execution of a new thread. Notice that **cdl** is passed as a parameter to **MyThread**'s constructor and stored in the **latch** instance variable. Then, the main thread calls **await( )** on **cdl**, which causes execution of the main thread to pause until **cdl**'s count has been decremented five times.

Inside the **run( )** method of **MyThread**, a loop is created that iterates five times. With each iteration, the **countDown( )** method is called on **latch**, which refers to **cdl** in **main( )**. After the fifth iteration, the latch opens, which allows the main thread to resume.

CountDownLatch is a powerful yet easy-to-use synchronization object that is appropriate whenever a thread must wait for one or more events to occur.

CyclicBarrier

A situation not uncommon in concurrent programming occurs when a set of two or more threads must wait at a predetermined execution point until all threads in the set have reached that point. To handle such a situation, the concurrent API supplies the **CyclicBarrier** class. It enables you to define a synchronization object that suspends until the specified number of threads has reached the barrier point.

CyclicBarrier has the following two constructors:

CyclicBarrier(int *numThreads*)
CyclicBarrier(int *numThreads*, Runnable *action*)

Here, *numThreads* specifies the number of threads that must reach the barrier before execution continues. In the second form, *action* specifies a thread that will be executed when the barrier is reached.

Here is the general procedure that you will follow to use **CyclicBarrier**. First, create a **CyclicBarrier** object, specifying the number of threads that you will be waiting for. Next, when each thread reaches the barrier, have it call **await( )** on that object. This will pause execution of the thread until all of the other threads also call **await( )**. Once the specified number of threads has reached the barrier, **await( )** will return and execution will resume. Also, if you have specified an action, then that thread is executed.

The **await( )** method has the following two forms:

```
int await( ) throws InterruptedException, BrokenBarrierException

int await(long wait, TimeUnit tu)
  throws InterruptedException, BrokenBarrierException, TimeoutException
```

The first form waits until all the threads have reached the barrier point. The second form waits only for the period of time specified by *wait.* The units represented by *wait* are specified by *tu.* Both forms return a value that indicates the order that the threads arrive at the barrier point. The first thread returns a value equal to the number of threads waited upon minus one. The last thread returns zero.

Here is an example that illustrates **CyclicBarrier**. It waits until a set of three threads has reached the barrier. When that occurs, the thread specified by **BarAction** executes.

```
// An example of CyclicBarrier.

import java.util.concurrent.*;

class BarDemo {
  public static void main(String args[]) {
    CyclicBarrier cb = new CyclicBarrier(3, new BarAction() );

    System.out.println("Starting");

    new Thread(new MyThread(cb, "A")).start();
    new Thread(new MyThread(cb, "B")).start();
    new Thread(new MyThread(cb, "C")).start();

  }
}

// A thread of execution that uses a CyclicBarrier.

class MyThread implements Runnable {
  CyclicBarrier cbar;
  String name;

  MyThread(CyclicBarrier c, String n) {
    cbar = c;
    name = n;
  }

  public void run() {

    System.out.println(name);

    try {
      cbar.await();
    } catch (BrokenBarrierException exc) {
      System.out.println(exc);
    } catch (InterruptedException exc) {
      System.out.println(exc);
    }
  }
}

// An object of this class is called when the
// CyclicBarrier ends.
```

```
class BarAction implements Runnable {
  public void run() {
    System.out.println("Barrier Reached!");
  }
}
```

The output is shown here. (The precise order in which the threads execute may vary.)

```
Starting
A
B
C
Barrier Reached!
```

A **CyclicBarrier** can be reused because it will release waiting threads each time the specified number of threads calls **await( )**. For example, if you change **main( )** in the preceding program so that it looks like this:

```
public static void main(String args[]) {
CyclicBarrier cb = new CyclicBarrier(3, new BarAction() );

  System.out.println("Starting");

  new Thread(new MyThread(cb, "A")).start();
  new Thread(new MyThread(cb, "B")).start();
  new Thread(new MyThread(cb, "C")).start();
  new Thread(new MyThread(cb, "X")).start();
  new Thread(new MyThread(cb, "Y")).start();
  new Thread(new MyThread(cb, "Z")).start();

}
```

the following output will be produced. (The precise order in which the threads execute may vary.)

```
Starting
A
B
C
Barrier Reached!
X
Y
```

```
Z
Barrier Reached!
```

As the preceding example shows, the **CyclicBarrier** offers a streamlined solution to what was previously a complicated problem.

Exchanger

Perhaps the most interesting of the synchronization classes is **Exchanger**. It is designed to simplify the exchange of data between two threads. The operation of an **Exchanger** is astoundingly simple: it simply waits until two separate threads call its **exchange( )** method. When that occurs, it exchanges the data supplied by the threads. This mechanism is both elegant and easy to use. Uses for **Exchanger** are easy to imagine. For example, one thread might prepare a buffer for receiving information over a network connection. Another thread might fill that buffer with the information from the connection. The two threads work together so that each time a new buffer is needed, an exchange is made.

Exchanger is a generic class that is declared as shown here:

Exchanger<V>

Here, **V** specifies the type of the data being exchanged.

The only method defined by **Exchanger** is **exchange( )**, which has the two forms shown here:

V exchange(V *objRef*) throws InterruptedException

V exchange(V *objRef*, long *wait*, TimeUnit *tu*)
  throws InterruptedException, TimeoutException

Here, *objRef* is a reference to the data to exchange. The data received from the other thread is returned. The second form of **exchange( )** allows a time-out period to be specified. The key point about **exchange( )** is that it won't succeed until it has been called on the same **Exchanger** object by two separate threads. Thus, **exchange( )** synchronizes the exchange of the data.

Here is an example that demonstrates **Exchanger**. It creates two threads. One thread creates an empty buffer that will receive the data put into it by the second thread. In this case, the data is a string. Thus, the first thread exchanges an empty string for a full one.

```
// An example of Exchanger.

import java.util.concurrent.Exchanger;

class ExgrDemo {
  public static void main(String args[]) {
    Exchanger<String> exgr = new Exchanger<String>();

    new Thread(new UseString(exgr)).start();
    new Thread(new MakeString(exgr)).start();
  }
}

// A Thread that constructs a string.
class MakeString implements Runnable {
  Exchanger<String> ex;
  String str;

  MakeString(Exchanger<String> c) {
    ex = c;
    str = new String();

  }
```

```
  public void run() {
    char ch = 'A';

    for(int i = 0; i < 3; i++) {

      // Fill Buffer
      for(int j = 0; j < 5; j++)
        str += ch++;

      try {
        // Exchange a full buffer for an empty one.
        str = ex.exchange(str);
      } catch(InterruptedException exc) {
        System.out.println(exc);
      }
    }
  }
}

// A Thread that uses a string.
class UseString implements Runnable {
  Exchanger<String> ex;
  String str;
  UseString(Exchanger<String> c) {
    ex = c;
  }

  public void run() {

    for(int i=0; i < 3; i++) {
      try {
        // Exchange an empty buffer for a full one.
        str = ex.exchange(new String());
        System.out.println("Got: " + str);
      } catch(InterruptedException exc) {
        System.out.println(exc);
      }
    }
  }
}
```

Here is the output produced by the program:

```
Got: ABCDE
Got: FGHIJ
Got: KLMNO
```

In the program, the **main( )** method creates an **Exchanger** for strings. This object is then used to synchronize the exchange of strings between the **MakeString** and **UseString** classes. The **MakeString** class fills a string with data. The **UseString** exchanges an empty string for a full one. It then displays the contents of the newly constructed string. The exchange of empty and full buffers is synchronized by the **exchange( )** method, which is called by both classes' **run( )** method.

Phaser

Another synchronization class is called **Phaser**. Its primary purpose is to enable the synchronization of threads that represent one or more phases of activity. For example, you might have a set of threads that implement three phases of an order-processing application. In the first phase, separate threads are used to validate customer information, check inventory, and confirm pricing. When that phase is complete, the second phase has two threads that compute shipping costs and all applicable tax. After that, a final phase confirms payment and determines estimated shipping time. In the past, to synchronize the multiple threads that comprise this scenario would require a bit of work on your part. With the inclusion of **Phaser**, the process is now much easier.

To begin, it helps to know that a **Phaser** works a bit like a **CyclicBarrier**, described earlier, except that it supports multiple phases. As a result, **Phaser** lets you define a synchronization object that waits until a specific phase has completed. It then advances to the next phase, again waiting until that phase concludes. It is important to understand that **Phaser** can also be used to synchronize only a single phase. In this regard, it acts much like a **CyclicBarrier**. However, its primary use is to synchronize multiple phases.

Phaser defines four constructors. Here are the two used in this section:

Phaser( )

Phaser(int *numParties*)

The first creates a phaser that has a registration count of zero. The second sets the registration count to *numParties*. The term *party* is often applied to the objects that register with a phaser. Although typically there is a one-to-

correspondence between the number of registrants and the number of threads being synchronized, this is not required. In both cases, the current phase is zero. That is, when a **Phaser** is created, it is initially at phase zero.

In general, here is how you use **Phaser**. First, create a new instance of **Phaser**. Next, register one or more parties with the phaser, either by calling **register( )** or by specifying the number of parties in the constructor. For each registered party, have the phaser wait until all registered parties complete a phase. A party signals this by calling one of a variety of methods supplied by **Phaser**, such as **arrive( )** or **arriveAndAwaitAdvance( )**. After all parties have arrived, the phase is complete, and the phaser can move on to the next phase (if there is one), or terminate. The following sections explain the process in detail.

To register parties after a **Phaser** has been constructed, call **register( )**. It is shown here:

```
int register()
```

It returns the phase number of the phase to which it is registered.

To signal that a party has completed a phase, it must call **arrive( )** or some variation of **arrive( )**. When the number of arrivals equals the number of registered parties, the phase is completed and the **Phaser** moves on to the next phase (if there is one). The **arrive( )** method has this general form:

```
int arrive( )
```

This method signals that a party (normally a thread of execution) has completed some task (or portion of a task). It returns the current phase number. If the phaser has been terminated, then it returns a negative value. The **arrive( )** method does not suspend execution of the calling thread. This means that it does not wait for the phase to be completed. This method should be called only by a registered party.

If you want to indicate the completion of a phase and then wait until all other registrants have also completed that phase, use **arriveAndAwaitAdvance( )**. It is shown here:

```
int arriveAndAwaitAdvance( )
```

It waits until all parties have arrived. It returns the next phase number or a negative value if the phaser has been terminated. This method should be called only by a registered party.

A thread can arrive and then deregister itself by calling **arriveAndDeregister( )**. It is shown here:

```
int arriveAndDeregister( )
```

It returns the current phase number or a negative value if the phaser has been terminated. It does not wait until the phase is complete. This method should be called only by a registered party.

To obtain the current phase number, call **getPhase( )**, which is shown here:

```
final int getPhase( )
```

When a **Phaser** is created, the first phase will be 0, the second phase 1, the third phase 2, and so on. A negative value is returned if the invoking **Phaser** has been terminated.

Here is an example that shows **Phaser** in action. It creates three threads, each of which have three phases. It uses a **Phaser** to synchronize each phase.

```
// An example of Phaser.

import java.util.concurrent.*;

class PhaserDemo {
  public static void main(String args[]) {
    Phaser phsr = new Phaser(1);
    int curPhase;

    System.out.println("Starting");

    new Thread(new MyThread(phsr, "A")).start();
    new Thread(new MyThread(phsr, "B")).start();
    new Thread(new MyThread(phsr, "C")).start();

    // Wait for all threads to complete phase one.
    curPhase = phsr.getPhase();
    phsr.arriveAndAwaitAdvance();
    System.out.println("Phase " + curPhase + " Complete");

    // Wait for all threads to complete phase two.
    curPhase =  phsr.getPhase();
    phsr.arriveAndAwaitAdvance();
    System.out.println("Phase " + curPhase + " Complete");

    curPhase =  phsr.getPhase();
    phsr.arriveAndAwaitAdvance();
    System.out.println("Phase " + curPhase + " Complete");
```

```
    // Deregister the main thread.
    phsr.arriveAndDeregister();

    if(phsr.isTerminated())
      System.out.println("The Phaser is terminated");
  }
}

// A thread of execution that uses a Phaser.
class MyThread implements Runnable {
  Phaser phsr;
  String name;

  MyThread(Phaser p, String n) {
    phsr = p;
    name = n;
    phsr.register();
  }

  public void run() {

    System.out.println("Thread " + name + " Beginning Phase One");
    phsr.arriveAndAwaitAdvance(); // Signal arrival.

    // Pause a bit to prevent jumbled output. This is for illustration
    // only. It is not required for the proper operation of the phaser.
    try {
      Thread.sleep(100);
    } catch(InterruptedException e) {
      System.out.println(e);
    }

    System.out.println("Thread " + name + " Beginning Phase Two");
    phsr.arriveAndAwaitAdvance(); // Signal arrival.

    // Pause a bit to prevent jumbled output. This is for illustration
    // only. It is not required for the proper operation of the phaser.
    try {
      Thread.sleep(100);
    } catch(InterruptedException e) {
      System.out.println(e);
    }

    System.out.println("Thread " + name + " Beginning Phase Three");
    phsr.arriveAndDeregister(); // Signal arrival and deregister.
  }
}
```

Sample output is shown here. (Your output may vary.)

```
Starting
Thread A Beginning Phase One
Thread C Beginning Phase One
Thread B Beginning Phase One

Phase 0 Complete
Thread B Beginning Phase Two
Thread C Beginning Phase Two
Thread A Beginning Phase Two
Phase 1 Complete
Thread C Beginning Phase Three
Thread B Beginning Phase Three
Thread A Beginning Phase Three
Phase 2 Complete
The Phaser is terminated
```

Let's look closely at the key sections of the program. First, in **main( )**, a **Phaser** called **phsr** is created with an initial party count of 1 (which corresponds to the main thread). Then three threads are started by creating three **MyThread** objects. Notice that **MyThread** is passed a reference to **phsr** (the phaser). The **MyThread** objects use this phaser to synchronize their activities. Next, **main( )** calls **getPhase( )** to obtain the current phase number (which is initially zero) and then calls **arriveAndAwaitAdvance( )**. This causes **main( )** to suspend until phase zero has completed. This won't happen until all **MyThread**s also call **arriveAndAwaitAdvance( )**. When this occurs, **main( )** will resume execution, at which point it displays that phase zero has completed, and it moves on to the next phase. This process repeats until all three phases have finished. Then, **main( )** calls **arriveAndDeregister( )**. At that point, all three **MyThread**s have also deregistered. Since this results in there being no registered parties when the phaser advances to the next phase, the phaser is terminated.

Now look at **MyThread**. First, notice that the constructor is passed a reference to the phaser that it will use and then registers with the new thread as a party on that phaser. Thus, each new **MyThread** becomes a party registered with the passed-in phaser. Also notice that each thread has three phases. In this example, each phase consists of a placeholder that simply displays the name of the thread and what it is doing. Obviously, in real-world code, the thread would be performing more meaningful actions. Between the first two phases, the thread calls **arriveAndAwaitAdvance( )**. Thus, each thread waits until all threads have

completed the phase (and the main thread is ready). After all threads have arrived (including the main thread), the phaser moves on to the next phase. After the third phase, each thread deregisters itself with a call to **arriveAndDeregister( )**. As the comments in **MyThread** explain, the calls to **sleep( )** are used for the purposes of illustration to ensure that the output is not jumbled because of the multithreading. They are not needed to make the phaser work properly. If you remove them, the output may look a bit jumbled, but the phases will still be synchronized correctly.

One other point: Although the preceding example used three threads that were all of the same type, this is not a requirement. Each party that uses a phaser can be unique, with each performing some separate task.

It is possible to take control of precisely what happens when a phase advance occurs. To do this, you must override the **onAdvance( )** method. This method is called by the run time when a **Phaser** advances from one phase to the next. It is shown here:

protected boolean onAdvance(int *phase*, int *numParties*)

Here, *phase* will contain the current phase number prior to being incremented and *numParties* will contain the number of registered parties. To terminate the phaser, **onAdvance( )** must return **true**. To keep the phaser alive, **onAdvance( )** must return **false**. The default version of **onAdvance( )** returns **true** (thus terminating the phaser) when there are no registered parties. As a general rule, your override should also follow this practice.

One reason to override **onAdvance( )** is to enable a phaser to execute a specific number of phases and then stop. The following example gives you the flavor of this usage. It creates a class called **MyPhaser** that extends **Phaser** so that it will run a specified number of phases. It does this by overriding the **onAdvance( )** method. The **MyPhaser** constructor accepts one argument, which specifies the number of phases to execute. Notice that **MyPhaser** automatically registers one party. This behavior is useful in this example, but the needs of your own applications may differ.

```
// Extend Phaser and override onAdvance() so that only a specific
// number of phases are executed.

import java.util.concurrent.*;

// Extend MyPhaser to allow only a specific number of phases
// to be executed.
class MyPhaser extends Phaser {
  int numPhases;

  MyPhaser(int parties, int phaseCount) {
    super(parties);
    numPhases = phaseCount - 1;
  }

  // Override onAdvance() to execute the specified
  // number of phases.
  protected boolean onAdvance(int p, int regParties) {
    // This println() statement is for illustration only.
    // Normally, onAdvance() will not display output.
    System.out.println("Phase " + p + " completed.\n");

    // If all phases have completed, return true
    if(p == numPhases || regParties == 0) return true;

    // Otherwise, return false.
    return false;
  }
}

class PhaserDemo2 {
  public static void main(String args[]) {

    MyPhaser phsr = new MyPhaser(1, 4);

    System.out.println("Starting\n");

    new Thread(new MyThread(phsr, "A")).start();
    new Thread(new MyThread(phsr, "B")).start();
    new Thread(new MyThread(phsr, "C")).start();
```

```
    // Wait for the specified number of phases to complete.
    while(!phsr.isTerminated()) {
      phsr.arriveAndAwaitAdvance();
    }

    System.out.println("The Phaser is terminated");
  }
}

// A thread of execution that uses a Phaser.
class MyThread implements Runnable {
  Phaser phsr;
  String name;

  MyThread(Phaser p, String n) {
    phsr = p;
    name = n;
    phsr.register();
  }

  public void run() {

    while(!phsr.isTerminated()) {
      System.out.println("Thread " + name + " Beginning Phase " +
                         phsr.getPhase());

      phsr.arriveAndAwaitAdvance();

      // Pause a bit to prevent jumbled output. This is for illustration
      // only. It is not required for the proper operation of the phaser.
      try {
        Thread.sleep(100);
      } catch(InterruptedException e) {
        System.out.println(e);
      }
    }
  }
}
```

The output from the program is shown here:

```
Starting

Thread B Beginning Phase 0
Thread A Beginning Phase 0
Thread C Beginning Phase 0
Phase 0 completed.

Thread A Beginning Phase 1
Thread B Beginning Phase 1
Thread C Beginning Phase 1
Phase 1 completed.

Thread C Beginning Phase 2
Thread B Beginning Phase 2
Thread A Beginning Phase 2
Phase 2 completed.

Thread C Beginning Phase 3
Thread B Beginning Phase 3
Thread A Beginning Phase 3
Phase 3 completed.

The Phaser is terminated
```

Inside **main( )**, one instance of **Phaser** is created. It is passed 4 as an argument, which means that it will execute four phases and then stop. Next, three threads are created and then the following loop is entered:

```
// Wait for the specified number of phases to complete.
while(!phsr.isTerminated()) {
  phsr.arriveAndAwaitAdvance();
}
```

This loop simply calls **arriveAndAwaitAdvance( )** until the phaser is terminated. The phaser won't terminate until the specified number of phases have been executed. In this case, the loop continues to execute until four phases have run. Next, notice that the threads also call **arriveAndAwaitAdvance( )** within a loop that runs until the phaser is terminated. This means that they will execute until the specified number of phases has been completed.

Now, look closely at the code for **onAdvance( )**. Each time **onAdvance( )** is called, it is passed the current phase and the number of registered parties. If the current phase equals the specified phase, or if the number of registered parties is

zero, **onAdvance( )** returns **true**, thus stopping the phaser. This is accomplished with this line of code:

```
// If all phases have completed, return true
if(p == numPhases || regParties == 0) return true;
```

As you can see, very little code is needed to accommodate the desired outcome.

Before moving on, it is useful to point out that you don't necessarily need to explicitly extend **Phaser** as the previous example does to simply override **onAdvance( )**. In some cases, more compact code can be created by using an anonymous inner class to override **onAdvance( )**.

Phaser has additional capabilities that may be of use in your applications. You can wait for a specific phase by calling **awaitAdvance( )**, which is shown here:

int awaitAdvance(int *phase*)

Here, *phase* indicates the phase number on which **awaitAdvance( )** will wait until a transition to the next phase takes place. It will return immediately if the argument passed to *phase* is not equal to the current phase. It will also return immediately if the phaser is terminated. However, if *phase* is passed the current phase, then it will wait until the phase increments. This method should be called only by a registered party. There is also an interruptible version of this method called **awaitAdvanceInterruptibly( )**.

To register more than one party, call **bulkRegister( )**. To obtain the number of registered parties, call **getRegisteredParties( )**. You can also obtain the number of arrived parties and unarrived parties by calling **getArrivedParties( )** and **getUnarrivedParties( )**, respectively. To force the phaser to enter a terminated state, call **forceTermination( )**.

Phaser also lets you create a tree of phasers. This is supported by two additional constructors, which let you specify the parent, and the **getParent( )** method.

Using an Executor

The concurrent API supplies a feature called an *executor* that initiates and controls the execution of threads. As such, an executor offers an alternative to managing threads through the **Thread** class.

At the core of an executor is the **Executor** interface. It defines the following method:

void execute(Runnable *thread*)

The thread specified by *thread* is executed. Thus, **execute( )** starts the specified thread.

The **ExecutorService** interface extends **Executor** by adding methods that help manage and control the execution of threads. For example, **ExecutorService** defines **shutdown( )**, shown here, which stops the invoking **ExecutorService**.

void shutdown( )

ExecutorService also defines methods that execute threads that return results, that execute a set of threads, and that determine the shutdown status. We will look at several of these methods a little later.

Also defined is the interface **ScheduledExecutorService**, which extends **ExecutorService** to support the scheduling of threads.

The concurrent API defines three predefined executor classes: **ThreadPoolExecutor** and **ScheduledThreadPoolExecutor**, and **ForkJoinPool**. **ThreadPoolExecutor** implements the **Executor** and **ExecutorService** interfaces and provides support for a managed pool of threads. **ScheduledThreadPoolExecutor** also implements the **ScheduledExecutorService** interface to allow a pool of threads to be scheduled. **ForkJoinPool** implements the **Executor** and **ExecutorService** interfaces and is used by the Fork/Join Framework. It is described later in this chapter.

A thread pool provides a set of threads that is used to execute various tasks. Instead of each task using its own thread, the threads in the pool are used. This reduces the overhead associated with creating many separate threads. Although you can use **ThreadPoolExecutor** and **ScheduledThreadPoolExecutor** directly, most often you will want to obtain an executor by calling one of the static factory methods defined by the **Executors** utility class. Here are some examples:

static ExecutorService newCachedThreadPool( )
static ExecutorService newFixedThreadPool(int *numThreads*)
static ScheduledExecutorService newScheduledThreadPool(int *numThreads*)

newCachedThreadPool( ) creates a thread pool that adds threads as needed but reuses threads if possible. **newFixedThreadPool( )** creates a thread pool that consists of a specified number of threads. **newScheduledThreadPool( )** creates a thread pool that supports thread scheduling. Each returns a reference to an **ExecutorService** that can be used to manage the pool.

A Simple Executor Example

Before going any further, a simple example that uses an executor will be of value. The following program creates a fixed thread pool that contains two threads. It then uses that pool to execute four tasks. Thus, four tasks share the two threads that are in the pool. After the tasks finish, the pool is shut down and the program ends.

```
// A simple example that uses an Executor.

import java.util.concurrent.*;

class SimpExec {
  public static void main(String args[]) {
    CountDownLatch cdl = new CountDownLatch(5);
    CountDownLatch cdl2 = new CountDownLatch(5);
    CountDownLatch cdl3 = new CountDownLatch(5);
    CountDownLatch cdl4 = new CountDownLatch(5);
    ExecutorService es = Executors.newFixedThreadPool(2);

    System.out.println("Starting");

    // Start the threads.
    es.execute(new MyThread(cdl, "A"));
    es.execute(new MyThread(cdl2, "B"));
    es.execute(new MyThread(cdl3, "C"));
    es.execute(new MyThread(cdl4, "D"));

    try {
      cdl.await();
      cdl2.await();
      cdl3.await();
      cdl4.await();
    } catch (InterruptedException exc) {
      System.out.println(exc);
    }

    es.shutdown();
    System.out.println("Done");
  }
}

class MyThread implements Runnable {
  String name;
  CountDownLatch latch;

  MyThread(CountDownLatch c, String n) {
    latch = c;
    name = n;

  }
```

```
  public void run() {

    for(int i = 0; i < 5; i++) {
      System.out.println(name + ": " + i);
      latch.countDown();
    }
  }
}
```

The output from the program is shown here. (The precise order in which the threads execute may vary.)

```
Starting
A: 0
A: 1
A: 2
A: 3
A: 4
C: 0
C: 1
C: 2
C: 3
C: 4
D: 0
D: 1
D: 2
D: 3
D: 4
B: 0
B: 1
B: 2
B: 3
B: 4
Done
```

As the output shows, even though the thread pool contains only two threads, all four tasks are still executed. However, only two can run at the same time. The others must wait until one of the pooled threads is available for use.

The call to **shutdown( )** is important. If it were not present in the program, then the program would not terminate because the executor would remain active. To try this for yourself, simply comment out the call to **shutdown( )** and observe the result.

Using Callable and Future

One of the most interesting features of the concurrent API is the **Callable** interface. This interface represents a thread that returns a value. An application can use **Callable** objects to compute results that are then returned to the invoking thread. This is a powerful mechanism because it facilitates the coding of many types of numerical computations in which partial results are computed simultaneously. It can also be used to run a thread that returns a status code that indicates the successful completion of the thread.

Callable is a generic interface that is defined like this:

```
interface Callable<V>
```

Here, **V** indicates the type of data returned by the task. **Callable** defines only one method, **call( )**, which is shown here:

```
V call( ) throws Exception
```

Inside **call( )**, you define the task that you want performed. After that task completes, you return the result. If the result cannot be computed, **call( )** must throw an exception.

A **Callable** task is executed by an **ExecutorService**, by calling its **submit( )** method. There are three forms of **submit( )**, but only one is used to execute a **Callable**. It is shown here:

```
<T> Future<T> submit(Callable<T> task)
```

Here, *task* is the **Callable** object that will be executed in its own thread. The result is returned through an object of type **Future**.

Future is a generic interface that represents the value that will be returned by a **Callable** object. Because this value is obtained at some future time, the name **Future** is appropriate. **Future** is defined like this:

```
interface Future<V>
```

Here, **V** specifies the type of the result.

To obtain the returned value, you will call **Future**'s **get( )** method, which has these two forms:

```
V get( )
  throws InterruptedException, ExecutionException
```

V get(long *wait*, TimeUnit *tu*)
  throws InterruptedException, ExecutionException, TimeoutException

The first form waits for the result indefinitely. The second form allows you to specify a timeout period in *wait.* The units of *wait* are passed in *tu,* which is an object of the **TimeUnit** enumeration, described later in this chapter.

The following program illustrates **Callable** and **Future** by creating three tasks that perform three different computations. The first returns the summation of a value, the second computes the length of the hypotenuse of a right triangle given the length of its sides, and the third computes the factorial of a value. All three computations occur simultaneously.

```
// An example that uses a Callable.

import java.util.concurrent.*;

class CallableDemo {
  public static void main(String args[]) {
    ExecutorService es = Executors.newFixedThreadPool(3);
    Future<Integer> f;
    Future<Double> f2;
    Future<Integer> f3;

    System.out.println("Starting");
```

```
    f = es.submit(new Sum(10));
    f2 = es.submit(new Hypot(3, 4));
    f3 = es.submit(new Factorial(5));

    try {
      System.out.println(f.get());
      System.out.println(f2.get());
      System.out.println(f3.get());
    } catch (InterruptedException exc) {
      System.out.println(exc);
    }
    catch (ExecutionException exc) {
      System.out.println(exc);
    }

    es.shutdown();
    System.out.println("Done");
  }
}

// Following are three computational threads.

class Sum implements Callable<Integer> {
  int stop;

  Sum(int v) { stop = v; }

  public Integer call() {
    int sum = 0;
    for(int i = 1; i <= stop; i++) {
      sum += i;
    }
    return sum;
  }
}

class Hypot implements Callable<Double> {
  double side1, side2;

  Hypot(double s1, double s2) {
    side1 = s1;
    side2 = s2;
  }

  public Double call() {
    return Math.sqrt((side1*side1) + (side2*side2));
  }
}

class Factorial implements Callable<Integer> {
  int stop;

  Factorial(int v) { stop = v; }
```

```
  public Integer call() {
    int fact = 1;
    for(int i = 2; i <= stop; i++) {
      fact *= i;
    }
    return fact;
  }
}
```

The output is shown here:

```
Starting
55
5.0
120
Done
```

The TimeUnit Enumeration

The concurrent API defines several methods that take an argument of type **TimeUnit**, which indicates a time-out period. **TimeUnit** is an enumeration that is used to specify the *granularity* (or resolution) of the timing. **TimeUnit** is defined within **java.util.concurrent**. It can be one of the following values:

DAYS
HOURS
MINUTES
SECONDS
MICROSECONDS
MILLISECONDS
NANOSECONDS

Although **TimeUnit** lets you specify any of these values in calls to methods that take a timing argument, there is no guarantee that the system is capable of the specified resolution.

Here is an example that uses **TimeUnit**. The **CallableDemo** class, shown in the previous section, is modified as shown next to use the second form of **get( )** that takes a **TimeUnit** argument.

```
try {
  System.out.println(f.get(10, TimeUnit.MILLISECONDS));
  System.out.println(f2.get(10, TimeUnit.MILLISECONDS));
  System.out.println(f3.get(10, TimeUnit.MILLISECONDS));
} catch (InterruptedException exc) {
  System.out.println(exc);
}
catch (ExecutionException exc) {
  System.out.println(exc);
} catch (TimeoutException exc) {
  System.out.println(exc);
}
```

In this version, no call to **get( )** will wait more than 10 milliseconds.

The **TimeUnit** enumeration defines various methods that convert between units. Those originally defined by **TimeUnit** are shown here:

long convert(long *tval*, TimeUnit *tu*)
long toMicros(long *tval*)
long toMillis(long *tval*)
long toNanos(long *tval*)
long toSeconds(long *tval*)
long toDays(long *tval*)
long toHours(long *tval*)
long toMinutes(long *tval*)

The **convert( )** method converts *tval* into the specified unit and returns the result. The **to** methods perform the indicated conversion and return the result. To these methods, JDK 9 added the methods **toChronoUnit( )** and **of( )**, which convert between **java.time.temporal.ChronoUnit**s and **TimeUnit**s. JDK 11 adds another version of **convert( )** that converts a **java.time.Duration** object into a **long**.

TimeUnit also defines the following timing methods:

void sleep(long *delay*) throws InterruptedExecution
void timedJoin(Thread *thrd*, long *delay*) throws InterruptedExecution
void timedWait(Object *obj*, long *delay*) throws InterruptedExecution

Here, **sleep( )** pauses execution for the specified delay period, which is specified in terms of the invoking enumeration constant. It translates into a call to

Thread.sleep( ). The **timedJoin( )** method is a specialized version of **Thread.join( )** in which *thrd* pauses for the time period specified by *delay*, which is described in terms of the invoking time unit. The **timedWait( )** method is a specialized version of **Object.wait( )** in which *obj* is waited on for the period of time specified by *delay*, which is described in terms of the invoking time unit.

The Concurrent Collections

As explained, the concurrent API defines several collection classes that have been engineered for concurrent operation. They include:

ArrayBlockingQueue
ConcurrentHashMap
ConcurrentLinkedDeque ConcurrentLinkedQueue
ConcurrentSkipListMap
ConcurrentSkipListSet
CopyOnWriteArrayList
CopyOnWriteArraySet
DelayQueue
LinkedBlockingDeque
LinkedBlockingQueue
LinkedTransferQueue PriorityBlockingQueue
SynchronousQueue

These offer concurrent alternatives to their related classes defined by the Collections Framework. These collections work much like the other collections except that they provide concurrency support. Programmers familiar with the Collections Framework will have no trouble using these concurrent collections.

Locks

The **java.util.concurrent.locks** package provides support for *locks*, which are objects that offer an alternative to using **synchronized** to control access to a shared resource. In general, here is how a lock works. Before accessing a shared resource, the lock that protects that resource is acquired. When access to the resource is complete, the lock is released. If a second thread attempts to acquire the lock when it is in use by another thread, the second thread will suspend until the lock is released. In this way, conflicting access to a shared resource is

prevented.

Locks are particularly useful when multiple threads need to access the value of shared data. For example, an inventory application might have a thread that first confirms that an item is in stock and then decreases the number of items on hand as each sale occurs. If two or more of these threads are running, then without some form of synchronization, it would be possible for one thread to be in the middle of a transaction when the second thread begins its transaction. The result could be that both threads would assume that adequate inventory exists, even if there is only sufficient inventory on hand to satisfy one sale. In this type of situation, a lock offers a convenient means of handling the needed synchronization.

The **Lock** interface defines a lock. The methods defined by **Lock** are shown in Table 28-1. In general, to acquire a lock, call **lock( )**. If the lock is unavailable, **lock( )** will wait. To release a lock, call **unlock( )**. To see if a lock is available, and to acquire it if it is, call **tryLock( )**. This method will not wait for the lock if it is unavailable. Instead, it returns **true** if the lock is acquired and **false** otherwise. The **newCondition( )** method returns a **Condition** object associated with the lock. Using a **Condition**, you gain detailed control of the lock through methods such as **await( )** and **signal( )**, which provide functionality similar to **Object.wait( )** and **Object.notify( )**.

Method	Description
void lock( )	Waits until the invoking lock can be acquired.
void lockInterruptibly( ) throws InterruptedException	Waits until the invoking lock can be acquired, unless interrupted.
Condition newCondition( )	Returns a **Condition** object that is associated with the invoking lock.
boolean tryLock( )	Attempts to acquire the lock. This method will not wait if the lock is unavailable. Instead, it returns **true** if the lock has been acquired and **false** if the lock is currently in use by another thread.
boolean tryLock(long *wait*, TimeUnit *tu*) throws InterruptedException	Attempts to acquire the lock. If the lock is unavailable, this method will wait no longer than the period specified by *wait*, which is in *tu* units. It returns **true** if the lock has been acquired and **false** if the lock cannot be acquired within the specified period.
void unlock( )	Releases the lock.

Table 28-1 The **Lock** Methods

java.util.concurrent.locks supplies an implementation of **Lock** called **ReentrantLock**. **ReentrantLock** implements a *reentrant lock,* which is a lock that can be repeatedly entered by the thread that currently holds the lock. Of course, in the case of a thread reentering a lock, all calls to **lock( )** must be offset by an equal number of calls to **unlock( )**. Otherwise, a thread seeking to acquire the lock will suspend until the lock is not in use.

The following program demonstrates the use of a lock. It creates two threads that access a shared resource called **Shared.count**. Before a thread can access **Shared.count**, it must obtain a lock. After obtaining the lock, **Shared.count** is incremented and then, before releasing the lock, the thread sleeps. This causes the second thread to attempt to obtain the lock. However, because the lock is still held by the first thread, the second thread must wait until the first thread stops sleeping and releases the lock. The output shows that access to **Shared.count** is, indeed, synchronized by the lock.

```
// A simple lock example.

import java.util.concurrent.locks.*;

class LockDemo {

  public static void main(String args[]) {
    ReentrantLock lock = new ReentrantLock();

    new Thread(new LockThread(lock, "A")).start();
    new Thread(new LockThread(lock, "B")).start();
  }
}

// A shared resource.
class Shared {
  static int count = 0;
}

// A thread of execution that increments count.
class LockThread implements Runnable {
  String name;
  ReentrantLock lock;

  LockThread(ReentrantLock lk, String n) {
    lock = lk;
    name = n;
  }

  public void run() {

    System.out.println("Starting " + name);
```

```
    try {
      // First, lock count.
      System.out.println(name + " is waiting to lock count.");
      lock.lock();
      System.out.println(name + " is locking count.");

      Shared.count++;
      System.out.println(name + ": " + Shared.count);

      // Now, allow a context switch -- if possible.
      System.out.println(name + " is sleeping.");
      Thread.sleep(1000);
    } catch (InterruptedException exc) {
      System.out.println(exc);
    } finally {
      // Unlock
      System.out.println(name + " is unlocking count.");
      lock.unlock();
    }
  }
}
```

The output is shown here. (The precise order in which the threads execute may vary.)

```
Starting A
A is waiting to lock count.
A is locking count.
A: 1
A is sleeping.
Starting B
B is waiting to lock count.
A is unlocking count.
B is locking count.
B: 2
B is sleeping.
B is unlocking count.
```

java.util.concurrent.locks also defines the **ReadWriteLock** interface. This interface specifies a lock that maintains separate locks for read and write access. This enables multiple locks to be granted for readers of a resource as long as the resource is not being written. **ReentrantReadWriteLock** provides an implementation of **ReadWriteLock**.

NOTE There is a specialized lock called **StampedLock**. It does not implement the **Lock** or **ReadWriteLock** interfaces. It does, however, provide a mechanism that enables aspects of it to be used like a **Lock** or **ReadWriteLock**.

Atomic Operations

java.util.concurrent.atomic offers an alternative to the other synchronization features when reading or writing the value of some types of variables. This package offers methods that get, set, or compare the value of a variable in one uninterruptible (that is, atomic) operation. This means that no lock or other synchronization mechanism is required.

Atomic operations are accomplished through the use of classes, such as **AtomicInteger** and **AtomicLong**, and methods such as **get( )**, **set( )**, **compareAndSet( )**, **decrementAndGet( )**, and **getAndSet( )**, which perform the action indicated by their names.

Here is an example that demonstrates how access to a shared integer can be synchronized by the use of **AtomicInteger**:

```
// A simple example of Atomic.

import java.util.concurrent.atomic.*;

class AtomicDemo {

  public static void main(String args[]) {
    new Thread(new AtomThread("A")).start();
    new Thread(new AtomThread("B")).start();
    new Thread(new AtomThread("C")).start();
  }
}

class Shared {
  static AtomicInteger ai = new AtomicInteger(0);
}

// A thread of execution that increments count.
class AtomThread implements Runnable {
  String name;

  AtomThread(String n) {
    name = n;
  }

public void run() {

  System.out.println("Starting " + name);

  for(int i=1; i <= 3; i++)
    System.out.println(name + " got: " +
           Shared.ai.getAndSet(i));
  }
}
```

In the program, a static **AtomicInteger** named **ai** is created by **Shared**. Then, three threads of type **AtomThread** are created. Inside **run( )**, **Shared.ai** is modified by calling **getAndSet( )**. This method returns the previous value and then sets the value to the one passed as an argument. The use of **AtomicInteger** prevents two threads from writing to **ai** at the same time.

In general, the atomic operations offer a convenient (and possibly more efficient) alternative to the other synchronization mechanisms when only a

single variable is involved. Among other features, **java.util.concurrent.atomic** also provides four classes that support lock-free cumulative operations. These are **DoubleAccumulator**, **DoubleAdder**, **LongAccumulator**, and **LongAdder**. The accumulator classes support a series of user-specified operations. The adder classes maintain a cumulative sum.

Parallel Programming via the Fork/Join Framework

In recent years, an important trend has emerged in software development: *parallel programming*. Parallel programming is the name commonly given to the techniques that take advantage of computers that contain two or more processors (multicore). As most readers will know, multicore computers have become commonplace. The advantage that multi-processor environments offer is the ability to significantly increase program performance. As a result, there had been a growing need for a mechanism that gives Java programmers a simple, yet effective way to make use of multiple processors in a clean, scalable manner. To answer this need, JDK 7 added several new classes and interfaces that support parallel programming. They are commonly referred to as the *Fork/Join Framework*. The Fork/Join Framework is defined in the **java.util.concurrent** package.

The Fork/Join Framework enhances multithreaded programming in two important ways. First, it simplifies the creation and use of multiple threads. Second, it automatically makes use of multiple processors. In other words, by using the Fork/Join Framework you enable your applications to automatically scale to make use of the number of available processors. These two features make the Fork/Join Framework the recommended approach to multithreading when parallel processing is desired.

Before continuing, it is important to point out the distinction between traditional multithreading and parallel programming. In the past, most computers had a single CPU and multithreading was primarily used to take advantage of idle time, such as when a program is waiting for user input. Using this approach, one thread can execute while another is waiting. In other words, on a single-CPU system, multithreading is used to allow two or more tasks to share the CPU. This type of multithreading is typically supported by an object of type **Thread** (as described in Chapter 11). Although this type of multithreading will always remain quite useful, it was not optimized for situations in which two or more

CPUs are available (multicore computers).

When multiple CPUs are present, a second type of multithreading capability that supports true parallel execution is required. With two or more CPUs, it is possible to execute portions of a program simultaneously, with each part executing on its own CPU. This can be used to significantly speed up the execution of some types of operations, such as sorting, transforming, or searching a large array. In many cases, these types of operations can be broken down into smaller pieces (each acting on a portion of the array), and each piece can be run on its own CPU. As you can imagine, the gain in efficiency can be enormous. Simply put: Parallel programming will be part of nearly every programmer's future because it offers a way to dramatically improve program performance.

The Main Fork/Join Classes

The Fork/Join Framework is packaged in **java.util.concurrent**. At the core of the Fork/Join Framework are the following four classes:

ForkJoinTask<V>	An abstract class that defines a task
ForkJoinPool	Manages the execution of **ForkJoinTask**s
RecursiveAction	A subclass of **ForkJoinTask<V>** for tasks that do not return values
RecursiveTask<V>	A subclass of **ForkJoinTask<V>** for tasks that return values

Here is how they relate. A **ForkJoinPool** manages the execution of **ForkJoinTask**s. **ForkJoinTask** is an abstract class that is extended by the abstract classes **RecursiveAction** and **RecursiveTask**. Typically, your code will extend these classes to create a task. Before looking at the process in detail, an overview of the key aspects of each class will be helpful.

NOTE The class **CountedCompleter** also extends **ForkJoinTask**. However, a discussion of **CountedCompleter** is beyond the scope of this book.

ForkJoinTask<V>

ForkJoinTask<V> is an abstract class that defines a task that can be managed by a **ForkJoinPool**. The type parameter **V** specifies the result type of the task. **ForkJoinTask** differs from **Thread** in that **ForkJoinTask** represents lightweight abstraction of a task, rather than a thread of execution. **ForkJoinTask**s are executed by threads managed by a thread pool of type **ForkJoinPool**. This

mechanism allows a large number of tasks to be managed by a small number of actual threads. Thus, **ForkJoinTask**s are very efficient when compared to threads.

ForkJoinTask defines many methods. At the core are **fork( )** and **join( )**, shown here:

```
final ForkJoinTask<V> fork( )

final V join( )
```

The **fork( )** method submits the invoking task for asynchronous execution of the invoking task. This means that the thread that calls **fork( )** continues to run. The **fork( )** method returns **this** after the task is scheduled for execution. Prior to JDK 8, **fork( )** could be executed only from within the computational portion of another **ForkJoinTask**, which is running within a **ForkJoinPool**. (You will see how to create the computational portion of a task shortly.) However, with the advent of JDK 8, if **fork( )** is not called while executing within a **ForkJoinPool**, then a common pool is automatically used. The **join( )** method waits until the task on which it is called terminates. The result of the task is returned. Thus, through the use of **fork( )** and **join( )**, you can start one or more new tasks and then wait for them to finish.

Another important **ForkJoinTask** method is **invoke( )**. It combines the fork and join operations into a single call because it begins a task and then waits for it to end. It is shown here:

```
final V invoke( )
```

The result of the invoking task is returned.

You can invoke more than one task at a time by using **invokeAll( )**. Two of its forms are shown here:

```
static void invokeAll(ForkJoinTask<?> taskA, ForkJoinTask<?> taskB)

static void invokeAll(ForkJoinTask<?> ... taskList)
```

In the first case, *taskA* and *taskB* are executed. In the second case, all specified tasks are executed. In both cases, the calling thread waits until all of the specified tasks have terminated. Prior to JDK 8, the **invokeAll( )** method could be executed only from within the computational portion of another

ForkJoinTask, which is running within a **ForkJoinPool**. JDK 8's inclusion of the common pool relaxed this requirement.

RecursiveAction

A subclass of **ForkJoinTask** is **RecursiveAction**. This class encapsulates a task that does not return a result. Typically, your code will extend **RecursiveAction** to create a task that has a **void** return type. **RecursiveAction** specifies four methods, but only one is usually of interest: the abstract method called **compute( )**. When you extend **RecursiveAction** to create a concrete class, you will put the code that defines the task inside **compute( )**. The **compute( )** method represents the *computational* portion of the task.

The **compute( )** method is defined by **RecursiveAction** like this:

```
protected abstract void compute( )
```

Notice that **compute( )** is **protected** and **abstract**. This means that it must be implemented by a subclass (unless that subclass is also abstract).

In general, **RecursiveAction** is used to implement a recursive, divide-and-conquer strategy for tasks that don't return results. (See "The Divide-and-Conquer Strategy" later in this chapter.)

RecursiveTask<V>

Another subclass of **ForkJoinTask** is **RecursiveTask<V>**. This class encapsulates a task that returns a result. The result type is specified by **V**. Typically, your code will extend **RecursiveTask<V>** to create a task that returns a value. Like **RecursiveAction**, its abstract **compute( )** method is often of the greatest interest because it represents the computational portion of the task. When you extend **RecursiveTask<V>** to create a concrete class, put the code that represents the task inside **compute( )**. This code must also return the result of the task.

The **compute( )** method is defined by **RecursiveTask<V>** like this:

```
protected abstract V compute( )
```

Notice that **compute( )** is **protected** and **abstract**. This means that it must be implemented by a subclass. When implemented, it must return the result of the task.

In general, **RecursiveTask** is used to implement a recursive, divide-and-

conquer strategy for tasks that return results. (See “The Divide-and-Conquer Strategy” later in this chapter.)

ForkJoinPool

The execution of **ForkJoinTask**s takes place within a **ForkJoinPool**, which also manages the execution of the tasks. Therefore, in order to execute a **ForkJoinTask**, you must first have a **ForkJoinPool**. Beginning with JDK 8, there are two ways to acquire a **ForkJoinPool**. First, you can explicitly create one by using a **ForkJoinPool** constructor. Second, you can use what is referred to as the *common pool*. The common pool (which was added by JDK 8) is a static **ForkJoinPool** that is automatically available for your use. Each method is introduced here, beginning with manually constructing a pool.

ForkJoinPool defines several constructors. Here are two commonly used ones:

ForkJoinPool( )

ForkJoinPool(int *pLevel*)

The first creates a default pool that supports a level of parallelism equal to the number of processors available in the system. The second lets you specify the level of parallelism. Its value must be greater than zero and not more than the limits of the implementation. The level of parallelism determines the number of threads that can execute concurrently. As a result, the level of parallelism effectively determines the number of tasks that can be executed simultaneously. (Of course, the number of tasks that can execute simultaneously cannot exceed the number of processors.) It is important to understand that the level of parallelism *does not*, however, limit the number of tasks that can be managed by the pool. A **ForkJoinPool** can manage many more tasks than its level of parallelism. Also, the level of parallelism is only a target. It is not a guarantee.

After you have created an instance of **ForkJoinPool**, you can start a task in a number of different ways. The first task started is often thought of as the main task. Frequently, the main task begins subtasks that are also managed by the pool. One common way to begin a main task is to call **invoke( )** on the **ForkJoinPool**. It is shown here:

<T> T invoke(ForkJoinTask<T> *task*)

This method begins the task specified by *task*, and it returns the result of the

task. This means that the calling code waits until **invoke( )** returns.

To start a task without waiting for its completion, you can use **execute( )**. Here is one of its forms:

void execute(ForkJoinTask<?> *task*)

In this case, *task* is started, but the calling code does not wait for its completion. Rather, the calling code continues execution asynchronously.

Beginning with JDK 8, it is not necessary to explicitly construct a **ForkJoinPool** because a common pool is available for your use. In general, if you are not using a pool that you explicitly created, then the common pool will automatically be used. Although it won't always be necessary, you can obtain a reference to the common pool by calling **commonPool( )**, which is defined by **ForkJoinPool**. It is shown here:

static ForkJoinPool commonPool( )

A reference to the common pool is returned. The common pool provides a default level of parallelism. It can be set by use of a system property. (See the API documentation for details.) Typically, the default common pool is a good choice for many applications. Of course, you can always construct your own pool.

There are two basic ways to start a task using the common pool. First, you can obtain a reference to the pool by calling **commonPool( )** and then use that reference to call **invoke( )** or **execute( )**, as just described. Second, you can call **ForkJoinTask** methods such as **fork( )** or **invoke( )** on the task from outside its computational portion. In this case, the common pool will automatically be used. In other words, **fork( )** and **invoke( )** will start a task using the common pool if the task is not already running within a **ForkJoinPool**.

ForkJoinPool manages the execution of its threads using an approach called *work-stealing*. Each worker thread maintains a queue of tasks. If one worker thread's queue is empty, it will take a task from another worker thread. This adds to overall efficiency and helps maintain a balanced load. (Because of demands on CPU time by other processes in the system, even two worker threads with identical tasks in their respective queues may not complete at the same time.)

One other point: **ForkJoinPool** uses daemon threads. A daemon thread is automatically terminated when all user threads have terminated. Thus, there is no need to explicitly shut down a **ForkJoinPool**. However, with the exception of the common pool, it is possible to do so by calling **shutdown( )**. The **shutdown(**

) method has no effect on the common pool.

The Divide-and-Conquer Strategy

As a general rule, users of the Fork/Join Framework will employ a *divide-and-conquer* strategy that is based on recursion. This is why the two subclasses of **ForkJoinTask** are called **RecursiveAction** and **RecursiveTask**. It is anticipated that you will extend one of these classes when creating your own fork/join task.

The divide-and-conquer strategy is based on recursively dividing a task into smaller subtasks until the size of a subtask is small enough to be handled sequentially. For example, a task that applies a transform to each element in an array of *N* integers can be broken down into two subtasks in which each transforms half the elements in the array. That is, one subtask transforms the elements 0 to *N*/2, and the other transforms the elements *N*/2 to *N*. In turn, each subtask can be reduced to another set of subtasks, each transforming half of the remaining elements. This process of dividing the array will continue until a threshold is reached in which a sequential solution is faster than creating another division.

The advantage of the divide-and-conquer strategy is that the processing can occur in parallel. Therefore, instead of cycling through an entire array using a single thread, pieces of the array can be processed simultaneously. Of course, the divide-and-conquer approach works in many cases in which an array (or collection) is not present, but the most common uses involve some type of array, collection, or grouping of data.

One of the keys to best employing the divide-and-conquer strategy is correctly selecting the threshold at which sequential processing (rather than further division) is used. Typically, an optimal threshold is obtained through profiling the execution characteristics. However, very significant speed-ups will still occur even when a less-than-optimal threshold is used. It is, however, best to avoid overly large or overly small thresholds. At the time of this writing, the Java API documentation for **ForkJoinTask<T>** states that, as a rule-of-thumb, a task should perform somewhere between 100 and 10,000 computational steps.

It is also important to understand that the optimal threshold value is also affected by how much time the computation takes. If each computational step is fairly long, then smaller thresholds might be better. Conversely, if each computational step is quite short, then larger thresholds could yield better results. For applications that are to be run on a known system, with a known number of processors, you can use the number of processors to make informed decisions

about the threshold value. However, for applications that will be running on a variety of systems, the capabilities of which are not known in advance, you can make no assumptions about the execution environment.

One other point: Although multiple processors may be available on a system, other tasks (and the operating system, itself) will be competing with your application for CPU time. Thus, it is important not to assume that your program will have unrestricted access to all CPUs. Furthermore, different runs of the same program may display different run time characteristics because of varying task loads.

A Simple First Fork/Join Example

At this point, a simple example that demonstrates the Fork/Join Framework and the divide-and-conquer strategy will be helpful. Following is a program that transforms the elements in an array of **double** into their square roots. It does so via a subclass of **RecursiveAction**. Notice that it creates its own **ForkJoinPool**.

```
// A simple example of the basic divide-and-conquer strategy.
// In this case, RecursiveAction is used.
import java.util.concurrent.*;
import java.util.*;

// A ForkJoinTask (via RecursiveAction) that transforms
// the elements in an array of doubles into their square roots.
class SqrtTransform extends RecursiveAction {
  // The threshold value is arbitrarily set at 1,000 in this example.
  // In real-world code, its optimal value can be determined by
  // profiling and experimentation.
  final int seqThreshold = 1000;

  // Array to be accessed.
  double[] data;

  // Determines what part of data to process.
  int start, end;

  SqrtTransform(double[] vals, int s, int e ) {
    data = vals;
    start = s;
    end = e;
  }

  // This is the method in which parallel computation will occur.
  protected void compute() {

    // If number of elements is below the sequential threshold,
    // then process sequentially.
    if((end - start) < seqThreshold) {
      // Transform each element into its square root.
      for(int i = start; i < end; i++) {
         data[i] = Math.sqrt(data[i]);
      }
    }
    else {
      // Otherwise, continue to break the data into smaller pieces.

      // Find the midpoint.
      int middle = (start + end) / 2;
```

```
      // Invoke new tasks, using the subdivided data.
      invokeAll(new SqrtTransform(data, start, middle),
                new SqrtTransform(data, middle, end));
    }
  }
}

// Demonstrate parallel execution.
class ForkJoinDemo {
  public static void main(String args[]) {
    // Create a task pool.
    ForkJoinPool fjp = new ForkJoinPool();

    double[] nums = new double[100000];

    // Give nums some values.
    for(int i = 0; i < nums.length; i++)
      nums[i] = (double) i;

    System.out.println("A portion of the original sequence:");

    for(int i=0; i < 10; i++)
      System.out.print(nums[i] + " ");
    System.out.println("\n");

    SqrtTransform task = new SqrtTransform(nums, 0, nums.length);

    // Start the main ForkJoinTask.
    fjp.invoke(task);

    System.out.println("A portion of the transformed sequence" +
                       " (to four decimal places):");
    for(int i=0; i < 10; i++)
      System.out.format("%.4f ", nums[i]);
    System.out.println();
  }
}
```

The output from the program is shown here:

```
A portion of the original sequence:
0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0

A portion of the transformed sequence (to four decimal places):
0.0000 1.0000 1.4142 1.7321 2.0000 2.2361 2.4495 2.6458 2.8284 3.0000
```

As you can see, the values of the array elements have been transformed into their square roots.

Let's look closely at how this program works. First, notice that **SqrtTransform** is a class that extends **RecursiveAction**. As explained, **RecursiveAction** extends **ForkJoinTask** for tasks that do not return results. Next, notice the **final** variable **seqThreshold**. This is the value that determines when sequential processing will take place. This value is set (somewhat arbitrarily) to 1,000. Next, notice that a reference to the array to be processed is stored in **data** and that the fields **start** and **end** are used to indicate the boundaries of the elements to be accessed.

The main action of the program takes place in **compute( )**. It begins by checking if the number of elements to be processed is below the sequential processing threshold. If it is, then those elements are processed (by computing their square root in this example). If the sequential processing threshold has not been reached, then two new tasks are started by calling **invokeAll( )**. In this case, each subtask processes half the elements. As explained earlier, **invokeAll( )** waits until both tasks return. After all of the recursive calls unwind, each element in the array will have been modified, with much of the action taking place in parallel (if multiple processors are available).

As mentioned, beginning with JDK 8, it is not necessary to explicitly construct a **ForkJoinPool** because a common pool is available for your use. Furthermore, using the common pool is a simple matter. For example, you can obtain a reference to the common pool by calling the static **commonPool( )** method defined by **ForkJoinPool**. Therefore, the preceding program could be rewritten to use the common pool by replacing the call to the **ForkJoinPool** constructor with a call to **commonPool( )**, as shown here:

```
ForkJoinPool fjp = ForkJoinPool.commonPool();
```

Alternatively, there is no need to explicitly obtain a reference to the common pool because calling the **ForkJoinTask** methods **invoke( )** or **fork( )** on a task that is not already part of a pool will cause it to execute within the common pool automatically. For example, in the preceding program, you can eliminate the **fjp** variable entirely and start the task using this line:

```
task.invoke();
```

As this discussion shows, the common pool can be easier to use than creating your own pool. Furthermore, in many cases, the common pool is the preferable approach.

Understanding the Impact of the Level of Parallelism

Before moving on, it is important to understand the impact that the level of parallelism has on the performance of a fork/join task and how the parallelism and the threshold interact. The program shown in this section lets you experiment with different degrees of parallelism and threshold values. Assuming that you are using a multicore computer, you can interactively observe the effect of these values.

In the preceding example, the default level of parallelism was used. However, you can specify the level of parallelism that you want. One way is to specify it when you create a **ForkJoinPool** using this constructor:

ForkJoinPool(int *pLevel*)

Here, *pLevel* specifies the level of parallelism, which must be greater than zero and less than the implementation defined limit.

The following program creates a fork/join task that transforms an array of **double**s. The transformation is arbitrary, but it is designed to consume several CPU cycles. This was done to ensure that the effects of changing the threshold or the level of parallelism would be more clearly displayed. To use the program, specify the threshold value and the level of parallelism on the command line. The program then runs the tasks. It also displays the amount of time it takes the tasks to run. To do this, it uses **System.nanoTime( )**, which returns the value of the JVM's high-resolution timer.

```
// A simple program that lets you experiment with the effects of
// changing the threshold and parallelism of a ForkJoinTask.
import java.util.concurrent.*;

// A ForkJoinTask (via RecursiveAction) that performs a
// a transform on the elements of an array of doubles.
class Transform extends RecursiveAction {

  // Sequential threshold, which is set by the constructor.
  int seqThreshold;

  // Array to be accessed.
  double[] data;

  // Determines what part of data to process.
  int start, end;

  Transform(double[] vals, int s, int e, int t ) {
    data = vals;
    start = s;
    end = e;
    seqThreshold = t;
  }

  // This is the method in which parallel computation will occur.
  protected void compute() {

    // If number of elements is below the sequential threshold,
    // then process sequentially.
    if((end - start) < seqThreshold) {
      // The following code assigns an element at an even index the
      // square root of its original value. An element at an odd
      // index is assigned its cube root. This code is designed
      // to simply consume CPU time so that the effects of concurrent
      // execution are more readily observable.
      for(int i = start; i < end; i++) {
        if((data[i] % 2) == 0)
          data[i] = Math.sqrt(data[i]);
        else
          data[i] = Math.cbrt(data[i]);
      }
    }
    else {
      // Otherwise, continue to break the data into smaller pieces.
```

```
      // Find the midpoint.
      int middle = (start + end) / 2;

      // Invoke new tasks, using the subdivided data.
      invokeAll(new Transform(data, start, middle, seqThreshold),
                new Transform(data, middle, end, seqThreshold));
    }
  }
}

// Demonstrate parallel execution.
class FJExperiment {

  public static void main(String args[]) {
    int pLevel;
    int threshold;

    if(args.length !=  2) {
      System.out.println("Usage: FJExperiment parallelism threshold ");
      return;
    }

    pLevel = Integer.parseInt(args[0]);
    threshold = Integer.parseInt(args[1]);

    // These variables are used to time the task.
    long beginT, endT;

    // Create a task pool. Notice that the parallelism level is set.
    ForkJoinPool fjp = new ForkJoinPool(pLevel);

    double[] nums = new double[1000000];

    for(int i = 0; i < nums.length; i++)
      nums[i] = (double) i;

    Transform task = new Transform(nums, 0, nums.length, threshold);

    // Starting timing.
    beginT = System.nanoTime();

    // Start the main ForkJoinTask.
    fjp.invoke(task);

    // End timing.
    endT = System.nanoTime();

    System.out.println("Level of parallelism: " + pLevel);
    System.out.println("Sequential threshold: " + threshold);
    System.out.println("Elapsed time: " + (endT - beginT) + " ns");
    System.out.println();
  }
}
```

To use the program, specify the level of parallelism followed by the threshold limit. You should try experimenting with different values for each, observing the results. Remember, to be effective, you must run the code on a computer with at least two processors. Also, understand that two different runs may (almost certainly will) produce different results because of the effect of other processes in the system consuming CPU time.

To give you an idea of the difference that parallelism makes, try this experiment. First, execute the program like this:

```
java FJExperiment 1 1000
```

This requests 1 level of parallelism (essentially sequential execution) with a threshold of 1,000. Here is a sample run produced on a dual-core computer:

```
Level of parallelism: 1
Sequential threshold: 1000
Elapsed time: 259677487 ns
```

Now, specify 2 levels of parallelism like this:

```
java FJExperiment 2 1000
```

Here is sample output from this run produced by the same dual-core computer:

```
Level of parallelism: 2
Sequential threshold: 1000
Elapsed time: 169254472 ns
```

As is evident, adding parallelism substantially decreases execution time, thus increasing the speed of the program. You should experiment with varying the threshold and parallelism on your own computer. The results may surprise you.

Here are two other methods that you might find useful when experimenting with the execution characteristics of a fork/join program. First, you can obtain the level of parallelism by calling **getParallelism( )**, which is defined by **ForkJoinPool**. It is shown here:

int getParallelism( )

It returns the parallelism level currently in effect. Recall that for pools that you create, by default, this value will equal the number of available processors. (To obtain the parallelism level for the common pool, you can also use

getCommonPoolParallelism( ). Second, you can obtain the number of processors available in the system by calling **availableProcessors( )**, which is defined by the **Runtime** class. It is shown here:

```
int availableProcessors( )
```

The value returned may change from one call to the next because of other system demands.

An Example that Uses RecursiveTask<V>

The two preceding examples are based on **RecursiveAction**, which means that they concurrently execute tasks that do not return results. To create a task that returns a result, use **RecursiveTask**. In general, solutions are designed in the same manner as just shown. The key difference is that the **compute( )** method returns a result. Thus, you must aggregate the results, so that when the first invocation finishes, it returns the overall result. Another difference is that you will typically start a subtask by calling **fork( )** and **join( )** explicitly (rather than implicitly by calling **invokeAll( )**, for example).

The following program demonstrates **RecursiveTask**. It creates a task called **Sum** that returns the summation of the values in an array of **double**. In this example, the array consists of 5,000 elements. However, every other value is negative. Thus, the first values in the array are 0, –1, 2, –3, 4, and so on. (Notice that this example creates its own pool. You might try changing it to use the common pool as an exercise.)

```
// A simple example that uses RecursiveTask<V>.
import java.util.concurrent.*;

// A RecursiveTask that computes the summation of an array of doubles.
class Sum extends RecursiveTask<Double> {

  // The sequential threshold value.
  final int seqThresHold = 500;

  // Array to be accessed.
  double[] data;

  // Determines what part of data to process.
  int start, end;

  Sum(double[] vals, int s, int e ) {
    data = vals;
    start = s;
    end = e;
  }

  // Find the summation of an array of doubles.
  protected Double compute() {
    double sum = 0;

    // If number of elements is below the sequential threshold,
    // then process sequentially.
    if((end - start) < seqThresHold) {
      // Sum the elements.
      for(int i = start; i < end; i++) sum += data[i];
    }
    else {
      // Otherwise, continue to break the data into smaller pieces.

      // Find the midpoint.
      int middle = (start + end) / 2;

      // Invoke new tasks, using the subdivided data.
      Sum subTaskA = new Sum(data, start, middle);
      Sum subTaskB = new Sum(data, middle, end);
```

```
      // Start each subtask by forking.
      subTaskA.fork();
      subTaskB.fork();

      // Wait for the subtasks to return, and aggregate the results.
      sum = subTaskA.join() + subTaskB.join();
    }
      // Return the final sum.
      return sum;
  }
}

// Demonstrate parallel execution.
class RecurTaskDemo {
  public static void main(String args[]) {
    // Create a task pool.
    ForkJoinPool fjp = new ForkJoinPool();

    double[] nums = new double[5000];

    // Initialize nums with values that alternate between
    // positive and negative.
    for(int i=0; i < nums.length; i++)
      nums[i] = (double) (((i%2) == 0) ? i : -i) ;

    Sum task = new Sum(nums, 0, nums.length);

    // Start the ForkJoinTasks.  Notice that, in this case,
    // invoke() returns a result.
    double summation = fjp.invoke(task);

    System.out.println("Summation " + summation);
  }
}
```

Here's the output from the program:

```
Summation -2500.0
```

There are a couple of interesting items in this program. First, notice that the two subtasks are executed by calling **fork( )**, as shown here:

```
subTaskA.fork();
```

```
subTaskB.fork();
```

In this case, **fork( )** is used because it starts a task but does not wait for it to finish. (Thus, it asynchronously runs the task.) The result of each task is obtained by calling **join( )**, as shown here:

```
sum = subTaskA.join() + subTaskB.join();
```

This statement waits until each task ends. It then adds the results of each and assigns the total to **sum**. Thus, the summation of each subtask is added to the running total. Finally, **compute( )** ends by returning **sum**, which will be the final total when the first invocation returns.

There are other ways to approach the handling of the asynchronous execution of the subtasks. For example, the following sequence uses **fork( )** to start **subTaskA** and uses **invoke( )** to start and wait for **subTaskB**:

```
subTaskA.fork();
sum = subTaskB.invoke() + subTaskA.join();
```

Another alternative is to have **subTaskB** call **compute( )** directly, as shown here:

```
subTaskA.fork();
sum = subTaskB.compute() + subTaskA.join();
```

Executing a Task Asynchronously

The preceding programs have called **invoke( )** on a **ForkJoinPool** to initiate a task. This approach is commonly used when the calling thread must wait until the task has completed (which is often the case) because **invoke( )** does not return until the task has terminated. However, you can start a task asynchronously. In this approach, the calling thread continues to execute. Thus, both the calling thread and the task execute simultaneously. To start a task asynchronously, use **execute( )**, which is also defined by **ForkJoinPool**. It has the two forms shown here:

void execute(ForkJoinTask<?> *task*)

void execute(Runnable *task*)

In both forms, *task* specifies the task to run. Notice that the second form lets you specify a **Runnable** rather than a **ForkJoinTask** task. Thus, it forms a bridge

between Java's traditional approach to multithreading and the Fork/Join Framework. It is important to remember that the threads used by a **ForkJoinPool** are daemon. Thus, they will end when the main thread ends. As a result, you may need to keep the main thread alive until the tasks have finished.

Cancelling a Task

A task can be cancelled by calling **cancel( )**, which is defined by **ForkJoinTask**. It has this general form:

```
boolean cancel(boolean interuptOK)
```

It returns **true** if the task on which it was called is cancelled. It returns **false** if the task has ended or can't be cancelled. At this time, the *interruptOK* parameter is not used by the default implementation. In general, **cancel( )** is intended to be called from code outside the task because a task can easily cancel itself by returning.

You can determine if a task has been cancelled by calling **isCancelled( )**, as shown here:

```
final boolean isCancelled( )
```

It returns **true** if the invoking task has been cancelled prior to completion and **false** otherwise.

Determining a Task's Completion Status

In addition to **isCancelled( )**, which was just described, **ForkJoinTask** includes two other methods that you can use to determine a task's completion status. The first is **isCompletedNormally( )**, which is shown here:

```
final boolean isCompletedNormally( )
```

It returns **true** if the invoking task completed normally, that is, if it did not throw an exception and it was not cancelled via a call to **cancel( )**. It returns **false** otherwise.

The second is **isCompletedAbnormally( )**, which is shown here:

```
final boolean isCompletedAbnormally( )
```

It returns **true** if the invoking task completed because it was cancelled or because it threw an exception. It returns **false** otherwise.

Restarting a Task

Normally, you cannot rerun a task. In other words, once a task completes, it cannot be restarted. However, you can reinitialize the state of the task (after it has completed) so it can be run again. This is done by calling **reinitialize( )**, as shown here:

```
void reinitialize( )
```

This method resets the state of the invoking task. However, any modification made to any persistent data that is operated upon by the task will not be undone. For example, if the task modifies an array, then those modifications are not undone by calling **reinitialize( )**.

Things to Explore

The preceding discussion presented the fundamentals of the Fork/Join Framework and described several commonly used methods. However, Fork/Join is a rich framework that includes additional capabilities that give you extended control over concurrency. Although it is far beyond the scope of this book to examine all of the issues and nuances surrounding parallel programming and the Fork/Join Framework, a sampling of the other features are mentioned here.

A Sampling of Other ForkJoinTask Features

In some cases, you will want to ensure that methods such as **invokeAll( )** and **fork( )** are called only from within a **ForkJoinTask**. This is usually a simple matter, but occasionally, you may have code that can be executed from either inside or outside a task. You can determine if your code is executing inside a task by calling **inForkJoinPool( )**.

You can convert a **Runnable** or **Callable** object into a **ForkJoinTask** by using the **adapt( )** method defined by **ForkJoinTask**. It has three forms, one for converting a **Callable**, one for a **Runnable** that does not return a result, and one for a **Runnable** that does return a result. In the case of a **Callable**, the **call( )** method is run. In the case of **Runnable**, the **run( )** method is run.

You can obtain an approximate count of the number of tasks that are in the

queue of the invoking thread by calling **getQueuedTaskCount( )**. You can obtain an approximate count of how many tasks the invoking thread has in its queue that are in excess of the number of other threads in the pool that might "steal" them, by calling **getSurplusQueuedTaskCount( )**. Remember, in the Fork/Join Framework, work-stealing is one way in which a high level of efficiency is obtained. Although this process is automatic, in some cases, the information may prove helpful in optimizing through-put.

ForkJoinTask defines the following variants of **join( )** and **invoke( )** that begin with the prefix **quietly**. They are shown here:

final void quietlyJoin( )	Joins a task, but does not return a result or throw an exception
final void quietlyInvoke( )	Invokes a task, but does not return a result or throw an exception.

In essence, these methods are similar to their non-quiet counterparts except they don't return values or throw exceptions.

You can attempt to "un-invoke" (in other words, unschedule) a task by calling **tryUnfork( )**.

Several methods, such as **getForkJoinTaskTag( )** and **setForkJoinTaskTag( )**, support tags. Tags are short integer values that are linked with a task. They may be useful in specialized applications.

ForkJoinTask implements **Serializable**. Thus, it can be serialized. However, serialization is not used during execution.

A Sampling of Other ForkJoinPool Features

One method that is quite useful when tuning fork/join applications is **ForkJoinPool**'s override of **toString( )**. It displays a "user-friendly" synopsis of the state of the pool. To see it in action, use this sequence to start and then wait for the task in the **FJExperiment** class of the task experimenter program shown earlier:

```
// Asynchronously start the main ForkJoinTask.
fjp.execute(task);

// Display the state of the pool while waiting.
while(!task.isDone()) {
  System.out.println(fjp);
}
```

When you run the program, you will see a series of messages on the screen that

describe the state of the pool. Here is an example of one. Of course, your output may vary, based on the number of processors, threshold values, task load, and so on.

```
java.util.concurrent.ForkJoinPool@141d683[Running, parallelism = 2,
size = 2, active = 0, running = 2, steals = 0, tasks = 0,
submissions = 1]
```

You can determine if a pool is currently idle by calling **isQuiescent( )**. It returns **true** if the pool has no active threads and **false** otherwise.

You can obtain the number of worker threads currently in the pool by calling **getPoolSize( )**. You can obtain an approximate count of the active threads in the pool by calling **getActiveThreadCount( )**.

To shut down a pool, call **shutdown( )**. Currently active tasks will still be executed, but no new tasks can be started. To stop a pool immediately, call **shutdownNow( )**. In this case, an attempt is made to cancel currently active tasks. (It is important to point out, however, that neither of these methods affects the common pool.) You can determine if a pool is shut down by calling **isShutdown( )**. It returns **true** if the pool has been shut down and **false** otherwise. To determine if the pool has been shut down and all tasks have been completed, call **isTerminated( )**.

Some Fork/Join Tips

Here are a few tips to help you avoid some of the more troublesome pitfalls associated with using the Fork/Join Framework. First, avoid using a sequential threshold that is too low. In general, erring on the high side is better than erring on the low side. If the threshold is too low, more time can be consumed generating and switching tasks than in processing the tasks. Second, usually it is best to use the default level of parallelism. If you specify a smaller number, it may significantly reduce the benefits of using the Fork/Join Framework.

In general, a **ForkJoinTask** should not use synchronized methods or synchronized blocks of code. Also, you will not normally want to have the **compute( )** method use other types of synchronization, such as a semaphore. (The **Phaser** can, however, be used when appropriate because it is compatible with the fork/join mechanism.) Remember, the main idea behind a **ForkJoinTask** is the divide-and-conquer strategy. Such an approach does not normally lend itself to situations in which outside synchronization is needed. Also, avoid situations in which substantial blocking will occur through I/O.

Therefore, in general, a **ForkJoinTask** will not perform I/O. Simply put, to best utilize the Fork/Join Framework, a task should perform a computation that can run without outside blocking or synchronization.

One last point: Except under unusual circumstances, do not make assumptions about the execution environment that your code will run in. This means you should not assume that some specific number of processors will be available, or that the execution characteristics of your program won't be affected by other processes running at the same time.

The Concurrency Utilities Versus Java's Traditional Approach

Given the power and flexibility found in the concurrency utilities, it is natural to ask the following question: Do they replace Java's traditional approach to multithreading and synchronization? The answer is a resounding no! The original support for multithreading and the built-in synchronization features are still the mechanism that should be employed for many, many Java programs. For example, **synchronized**, **wait( )**, and **notify( )** offer elegant solutions to a wide range of problems. However, when extra control is needed, the concurrency utilities are available to handle the chore. Furthermore, the Fork/Join Framework offers a powerful way to integrate parallel programming techniques into your more sophisticated applications.

CHAPTER

The Stream API

Of the many new features recently added to Java, two of the most important are lambda expressions and the stream API. Lambda expressions were described in Chapter 15. The stream API is described here. As you will see, the stream API is designed with lambda expressions in mind. Moreover, the stream API provides some of the most significant demonstrations of the power that lambdas bring to Java.

Although its design compatibility with lambda expressions is impressive, the key aspect of the stream API is its ability to perform very sophisticated operations that search, filter, map, or otherwise manipulate data. For example, using the stream API, you can construct sequences of actions that resemble, in concept, the type of database queries for which you might use SQL. Furthermore, in many cases, such actions can be performed in parallel, thus providing a high level of efficiency, especially when large data sets are involved. Put simply, the stream API provides a powerful means of handling data in an efficient, yet easy to use way.

Before continuing, an important point needs to be made: The stream API uses some of Java's most advanced features. To fully understand and utilize it requires a solid understanding of generics and lambda expressions. The basic concepts of parallel execution and a working knowledge of the Collections Framework are also needed. (See Chapters 14, 15, 19, and 28.)

Stream Basics

Let's begin by defining the term *stream* as it applies to the stream API: a stream is a conduit for data. Thus, a stream represents a sequence of objects. A stream operates on a data source, such as an array or a collection. A stream, itself, never provides storage for the data. It simply moves data, possibly filtering, sorting, or otherwise operating on that data in the process. As a general rule, however, a stream operation by itself does not modify the data source. For example, sorting a stream does not change the order of the source. Rather, sorting a stream results in the creation of a new stream that produces the sorted result.

NOTE It is necessary to state that the term *stream* as used here differs from the use of *stream* when the I/O classes were described earlier in this book. Although an I/O stream can act conceptually much like one of the streams defined by **java.util.stream**, they are not the same. Thus, throughout this chapter, when the term *stream* is used, it refers to objects based on one of the stream types described here.

Stream Interfaces

The stream API defines several stream interfaces, which are packaged in **java.util.stream** and contained in the **java.base** module. At the foundation is **BaseStream**, which defines the basic functionality available in all streams. **BaseStream** is a generic interface declared like this:

```
interface BaseStream<T, S extends BaseStream<T, S>>
```

Here, **T** specifies the type of the elements in the stream, and **S** specifies the type of stream that extends **BaseStream**. **BaseStream** extends the **AutoCloseable** interface; thus, a stream can be managed in a **try**-with-resources statement. In general, however, only those streams whose data source requires closing (such as those connected to a file) will need to be closed. In most cases, such as those in which the data source is a collection, there is no need to close the stream. The methods declared by **BaseStream** are shown in Table 29-1.

Method	Description
void close( )	Closes the invoking stream, calling any registered close handlers. (As explained in the text, few streams need to be closed.)
boolean isParallel( )	Returns **true** if the invoking stream is parallel. Returns **false** if the stream is sequential.
Iterator<T> iterator( )	Obtains an iterator to the stream and returns a reference to it. (Terminal operation.)
S onClose(Runnable *handler*)	Returns a new stream with the close handler specified by *handler*. This handler will be called when the stream is closed. (Intermediate operation.)
S parallel( )	Returns a parallel stream based on the invoking stream. If the invoking stream is already parallel, then that stream is returned. (Intermediate operation.)
S sequential( )	Returns a sequential stream based on the invoking stream. If the invoking stream is already sequential, then that stream is returned. (Intermediate operation.)
Spliterator<T> spliterator( )	Obtains a spliterator to the stream and returns a reference to it. (Terminal operation.)
S unordered( )	Returns an unordered stream based on the invoking stream. If the invoking stream is already unordered, then that stream is returned. (Intermediate operation.)

Table 29-1 The Methods Declared by **BaseStream**

From **BaseStream** are derived several types of stream interfaces. The most general of these is **Stream**. It is declared as shown here:

```
interface Stream<T>
```

Here, **T** specifies the type of the elements in the stream. Because it is generic, **Stream** is used for all reference types. In addition to the methods that it inherits from **BaseStream**, the **Stream** interface adds several of its own, a sampling of which is shown in Table 29-2.

Method	Description
<R, A> R collect(Collector<? super T, A, R> *collectorFunc*)	Collects elements into a container, which is changeable, and returns the container. This is called a mutable reduction operation. Here, **R** specifies the type of the resulting container and **T** specifies the element type of the invoking stream. **A** specifies the internal accumulated type. The *collectorFunc* specifies how the collection process works. (Terminal operation.)
long count( )	Counts the number of elements in the stream and returns the result. (Terminal operation.)
Stream<T> filter(Predicate<? super T> *pred*)	Produces a stream that contains those elements from the invoking stream that satisfy the predicate specified by *pred*. (Intermediate operation.)
void forEach(Consumer<? super T> *action*)	For each element in the invoking stream, the code specified by *action* is executed. (Terminal operation.)
<R> Stream<R> map(Function<? super T, ? extends R> *mapFunc*)	Applies *mapFunc* to the elements from the invoking stream, yielding a new stream that contains those elements. (Intermediate operation.)
DoubleStream mapToDouble(ToDoubleFunction<? super T> *mapFunc*)	Applies *mapFunc* to the elements from the invoking stream, yielding a new **DoubleStream** that contains those elements. (Intermediate operation.)
IntStream mapToInt(ToIntFunction<? super T> *mapFunc*)	Applies *mapFunc* to the elements from the invoking stream, yielding a new **IntStream** that contains those elements. (Intermediate operation.)
LongStream mapToLong(ToLongFunction<? super T> *mapFunc*)	Applies *mapFunc* to the elements from the invoking stream, yielding a new **LongStream** that contains those elements. (Intermediate operation.)
Optional<T> max(Comparator<? super T> *comp*)	Using the ordering specified by *comp*, finds and returns the maximum element in the invoking stream. (Terminal operation.)
Optional<T> min(Comparator<? super T> *comp*)	Using the ordering specified by *comp*, finds and returns the minimum element in the invoking stream. (Terminal operation.)
T reduce(T *identityVal*, BinaryOperator<T> *accumulator*)	Returns a result based on the elements in the invoking stream. This is called a reduction operation. (Terminal operation.)
Stream<T> sorted( )	Produces a new stream that contains the elements of the invoking stream sorted in natural order. (Intermediate operation.)
Object[] toArray( )	Creates an array from the elements in the invoking stream. (Terminal operation.)

Table 29-2 A Sampling of Methods Declared by **Stream**

In both tables, notice that many of the methods are notated as being either *terminal* or *intermediate*. The difference between the two is very important. A *terminal* operation consumes the stream. It is used to produce a result, such as finding the minimum value in the stream, or to execute some action, as is the case with the **forEach( )** method. Once a stream has been consumed, it cannot be reused. *Intermediate* operations produce another stream. Thus, intermediate operations can be used to create a *pipeline* that performs a sequence of actions. One other point: intermediate operations do not take place immediately. Instead, the specified action is performed when a terminal operation is executed on the new stream created by an intermediate operation. This mechanism is referred to as *lazy behavior,* and the intermediate operations are referred to as *lazy.* The use of lazy behavior enables the stream API to perform more efficiently.

Another key aspect of streams is that some intermediate operations are *stateless* and some are *stateful*. In a stateless operation, each element is processed independently of the others. In a stateful operation, the processing of an element may depend on aspects of the other elements. For example, sorting is a stateful operation because an element's order depends on the values of the other elements. Thus, the **sorted( )** method is stateful. However, filtering elements based on a stateless predicate is stateless because each element is handled individually. Thus, **filter( )** can (and should be) stateless. The difference between stateless and stateful operations is especially important when parallel processing of a stream is desired because a stateful operation may require more than one pass to complete.

Because **Stream** operates on object references, it can't operate directly on primitive types. To handle primitive type streams, the stream API defines the following interfaces:

DoubleStream

IntStream

LongStream

These streams all extend **BaseStream** and have capabilities similar to **Stream** except that they operate on primitive types rather than reference types. They also provide some convenience methods, such as **boxed( )**, that facilitate their use. Because streams of objects are the most common, **Stream** is the primary focus

of this chapter, but the primitive type streams can be used in much the same way.

How to Obtain a Stream

You can obtain a stream in a number of ways. Perhaps the most common is when a stream is obtained for a collection. Beginning with JDK 8, the **Collection** interface was expanded to include two methods that obtain a stream from a collection. The first is **stream( )**, shown here:

default Stream<E> stream( )

Its default implementation returns a sequential stream. The second method is **parallelStream( )**, shown next:

default Stream<E> parallelStream( )

Its default implementation returns a parallel stream, if possible. (If a parallel stream can not be obtained, a sequential stream may be returned instead.) Parallel streams support parallel execution of stream operations. Because **Collection** is implemented by every collection, these methods can be used to obtain a stream from any collection class, such as **ArrayList** or **HashSet**.

A stream can also be obtained from an array by use of the static **stream( )** method, which was added to the **Arrays** class. One of its forms is shown here:

static <T> Stream<T> stream(T[ ] *array*)

This method returns a sequential stream to the elements in *array*. For example, given an array called **addresses** of type **Address**, the following obtains a stream to it:

```
Stream<Address> addrStrm = Arrays.stream(addresses);
```

Several overloads of the **stream( )** method are also defined, such as those that handle arrays of the primitive types. They return a stream of type **IntStream**, **DoubleStream**, or **LongStream**.

Streams can be obtained in a variety of other ways. For example, many stream operations return a new stream, and a stream to an I/O source can be obtained by calling **lines( )** on a **BufferedReader**. However a stream is obtained, it can be used in the same way as any other stream.

A Simple Stream Example

Before going any further, let’s work through an example that uses streams. The following program creates an **ArrayList** called **myList** that holds a collection of integers (which are automatically boxed into the **Integer** reference type). Next, it obtains a stream that uses **myList** as a source. It then demonstrates various stream operations.

```
// Demonstrate several stream operations.

import java.util.*;
import java.util.stream.*;
```

```
class StreamDemo {

  public static void main(String[] args) {

    // Create a list of Integer values.
    ArrayList<Integer> myList = new ArrayList<>( );
    myList.add(7);
    myList.add(18);
    myList.add(10);
    myList.add(24);
    myList.add(17);
    myList.add(5);

    System.out.println("Original list: " + myList);

    // Obtain a Stream to the array list.
    Stream<Integer> myStream = myList.stream();

    // Obtain the minimum and maximum value by use of min(),
    // max(), isPresent(), and get().
    Optional<Integer> minVal = myStream.min(Integer::compare);
    if(minVal.isPresent()) System.out.println("Minimum value: " +
                                              minVal.get());

    // Must obtain a new stream because previous call to min()
    // is a terminal operation that consumed the stream.
    myStream = myList.stream();
    Optional<Integer> maxVal = myStream.max(Integer::compare);
    if(maxVal.isPresent()) System.out.println("Maximum value: " +
                                              maxVal.get());

    // Sort the stream by use of sorted().
    Stream<Integer> sortedStream = myList.stream().sorted();

    // Display the sorted stream by use of forEach().
    System.out.print("Sorted stream: ");
    sortedStream.forEach((n) -> System.out.print(n + " "));
    System.out.println();

    // Display only the odd values by use of filter().
    Stream<Integer> oddVals =
           myList.stream().sorted().filter((n) -> (n % 2) == 1);
    System.out.print("Odd values: ");
    oddVals.forEach((n) -> System.out.print(n + " "));
    System.out.println();

    // Display only the odd values that are greater than 5. Notice that
    // two filter operations are pipelined.
    oddVals = myList.stream().filter( (n) -> (n % 2) == 1)
                             .filter((n) -> n > 5);
    System.out.print("Odd values greater than 5: ");
```

```
    oddVals.forEach((n) -> System.out.print(n + " ") );
    System.out.println();
  }
}
```

The output is shown here:

```
Original list: [7, 18, 10, 24, 17, 5]
Minimum value: 5
Maximum value: 24
Sorted stream: 5 7 10 17 18 24
Odd values: 5 7 17
Odd values greater than 5: 7 17
```

Let's look closely at each stream operation. After creating an **ArrayList**, the program obtains a stream for the list by calling **stream( )**, as shown here:

```
Stream<Integer> myStream = myList.stream();
```

As explained, the **Collection** interface defines the **stream( )** method, which obtains a stream from the invoking collection. Because **Collection** is implemented by every collection class, **stream( )** can be used to obtain a stream for any type of collection, including the **ArrayList** used here. In this case, a reference to the stream is assigned to **myStream**.

Next, the program obtains the minimum value in the stream (which is, of course, also the minimum value in the data source) and displays it, as shown here:

```
Optional<Integer> minVal = myStream.min(Integer::compare);
if(minVal.isPresent()) System.out.println("Minimum value: " +
                                          minVal.get());
```

Recall from Table 29-2 that **min( )** is declared like this:

Optional<T> min(Comparator<? super T> *comp*)

First, notice that the type of **min( )'s** parameter is a **Comparator**. This comparator is used to compare two elements in the stream. In the example, **min( )** is passed a method reference to **Integer**'s **compare( )** method, which is used to implement a **Comparator** capable of comparing two **Integer**s. Next, notice that the return type of **min( )** is **Optional**. The **Optional** class is described in Chapter

20, but briefly, here is how it works. **Optional** is a generic class packaged in **java.util** and declared like this:

class Optional<T>

Here, **T** specifies the element type. An **Optional** instance can either contain a value of type **T** or be empty. You can use **isPresent( )** to determine if a value is present. Assuming that a value is available, it can be obtained by calling **get( )**, or if you are using JDK 10 or later, **orElseThrow( )**. Here, **get( )** is used. In this example, the object returned will hold the minimum value of the stream as an **Integer** object.

One other point about the preceding line: **min( )** is a terminal operation that consumes the stream. Thus, **myStream** cannot be used again after **min( )** executes.

The next lines obtain and display the maximum value in the stream:

```
myStream = myList.stream();
Optional<Integer> maxVal = myStream.max(Integer::compare);
if(maxVal.isPresent()) System.out.println("Maximum value: " +
                                          maxVal.get());
```

First, **myStream** is once again assigned the stream returned by **myList.stream( )**. As just explained, this is necessary because the previous call to **min( )** consumed the previous stream. Thus, a new one is needed. Next, the **max( )** method is called to obtain the maximum value. Like **min( )**, **max( )** returns an **Optional** object. Its value is obtained by calling **get( )**.

The program then obtains a sorted stream through the use of this line:

```
Stream<Integer> sortedStream = myList.stream().sorted();
```

Here, the **sorted( )** method is called on the stream returned by **myList.stream( )**. Because **sorted( )** is an intermediate operation, its result is a new stream, and this is the stream assigned to **sortedStream**. The contents of the sorted stream are displayed by use of **forEach( )**:

```
sortedStream.forEach((n) -> System.out.print(n + " "));
```

Here, the **forEach( )** method executes an operation on each element in the stream. In this case, it simply calls **System.out.print( )** for each element in **sortedStream**. This is accomplished by use of a lambda expression. The

forEach( ) method has this general form:

void forEach(Consumer<? super T> *action*)

Consumer is a generic functional interface declared in **java.util.function**. Its abstract method is **accept( )**, shown here:

void accept(T *objRef*)

The lambda expression in the call to **forEach( )** provides the implementation of **accept( )**. The **forEach( )** method is a terminal operation. Thus, after it completes, the stream has been consumed.

Next, a sorted stream is filtered by **filter( )** so that it contains only odd values:

```
Stream<Integer> oddVals =
       myList.stream().sorted().filter((n) -> (n % 2) == 1);
```

The **filter( )** method filters a stream based on a predicate. It returns a new stream that contains only those elements that satisfy the predicate. It is shown here:

Stream<T> filter(Predicate<? super T> *pred*)

Predicate is a generic functional interface defined in **java.util.function**. Its abstract method is **test( )**, which is shown here:

boolean test(T *objRef*)

It returns **true** if the object referred to by *objRef* satisfies the predicate, and **false** otherwise. The lambda expression passed to **filter( )** implements this method. Because **filter( )** is an intermediate operation, it returns a new stream that contains filtered values, which, in this case, are the odd numbers. These elements are then displayed via **forEach( )** as before.

Because **filter( )**, or any other intermediate operation, returns a new stream, it is possible to filter a filtered stream a second time. This is demonstrated by the following line, which produces a stream that contains only those odd values greater than 5:

```
oddVals = myList.stream().filter((n) -> (n % 2) == 1)
                         .filter((n) -> n > 5);
```

Notice that lambda expressions are passed to both filters.

Reduction Operations

Consider the **min( )** and **max( )** methods in the preceding example program. Both are terminal operations that return a result based on the elements in the stream. In the language of the stream API, they represent *reduction operations* because each reduces a stream to a single value—in this case, the minimum and maximum. The stream API refers to these as *special case* reductions because they perform a specific function. In addition to **min( )** and **max( )**, other special case reductions are also available, such as **count( )**, which counts the number of elements in a stream. However, the stream API generalizes this concept by providing the **reduce( )** method. By using **reduce( )**, you can return a value from a stream based on any arbitrary criteria. By definition, all reduction operations are terminal operations.

Stream defines three versions of **reduce( )**. The two we will use first are shown here:

Optional<T> reduce(BinaryOperator<T> *accumulator*)

T reduce(T *identityVal*, BinaryOperator<T> *accumulator*)

The first form returns an object of type **Optional**, which contains the result. The second form returns an object of type **T** (which is the element type of the stream). In both forms, *accumulator* is a function that operates on two values and produces a result. In the second form, *identityVal* is a value such that an accumulator operation involving *identityVal* and any element of the stream yields that element, unchanged. For example, if the operation is addition, then the identity value will be 0 because 0 + x is x. For multiplication, the value will be 1, because 1 * x is x.

BinaryOperator is a functional interface declared in **java.util.function** that extends the **BiFunction** functional interface. **BiFunction** defines this abstract method:

R apply(T *val*, U *val2*)

Here, **R** specifies the result type, **T** is the type of the first operand, and **U** is the type of second operand. Thus, **apply( )** applies a function to its two operands (*val* and *val2*) and returns the result. When **BinaryOperator** extends **BiFunction**, it specifies the same type for all the type parameters. Thus, as it relates to **BinaryOperator**, **apply( )** looks like this:

T apply(T *val*, T *val2*)

Furthermore, as it relates to **reduce( )**, *val* will contain the previous result and *val2* will contain the next element. In its first invocation, *val* will contain either the identity value or the first element, depending on which version of **reduce( )** is used.

It is important to understand that the accumulator operation must satisfy three constraints. It must be

- Stateless
- Non-interfering
- Associative

As explained earlier, *stateless* means that the operation does not rely on any state information. Thus, each element is processed independently. *Non-interfering* means that the data source is not modified by the operation. Finally, the operation must be *associative*. Here, the term *associative* is used in its normal, arithmetic sense, which means that, given an associative operator used in a sequence of operations, it does not matter which pair of operands are processed first. For example,

(10 * 2) * 7

yields the same result as

10 * (2 * 7)

Associativity is of particular importance to the use of reduction operations on parallel streams, discussed in the next section.

The following program demonstrates the versions of **reduce( )** just described:

```
// Demonstrate the reduce() method.

import java.util.*;
import java.util.stream.*;

class StreamDemo2 {

  public static void main(String[] args) {

    // Create a list of Integer values.
    ArrayList<Integer> myList = new ArrayList<>( );

    myList.add(7);
    myList.add(18);
    myList.add(10);
    myList.add(24);
    myList.add(17);
    myList.add(5);

    // Two ways to obtain the integer product of the elements
    // in myList by use of reduce().
    Optional<Integer> productObj = myList.stream().reduce((a,b) -> a*b);
    if(productObj.isPresent())
      System.out.println("Product as Optional: " + productObj.get());

    int product = myList.stream().reduce(1, (a,b) -> a*b);
    System.out.println("Product as int: " + product);
  }
}
```

As the output here shows, both uses of **reduce( )** produce the same result:

```
Product as Optional: 2570400
Product as int: 2570400
```

In the program, the first version of **reduce( )** uses the lambda expression to produce a product of two values. In this case, because the stream contains **Integer** values, the **Integer** objects are automatically unboxed for the multiplication and reboxed to return the result. The two values represent the current value of the running result and the next element in the stream. The final result is returned in an object of type **Optional**. The value is obtained by calling **get( )** on the returned object.

In the second version, the identity value is explicitly specified, which for

multiplication is 1. Notice that the result is returned as an object of the element type, which is **Integer** in this case.

Although simple reduction operations such as multiplication are useful for examples, reductions are not limited in this regard. For example, assuming the preceding program, the following obtains the product of only the even values:

```
int evenProduct = myList.stream().reduce(1, (a,b) -> {
                    if(b%2 == 0) return a*b; else return a;
                  });
```

Pay special attention to the lambda expression. If **b** is even, then **a** * **b** is returned. Otherwise, **a** is returned. This works because **a** holds the current result and **b** holds the next element, as explained earlier.

Using Parallel Streams

Before exploring any more of the stream API, it will be helpful to discuss parallel streams. As has been pointed out previously in this book, the parallel execution of code via multicore processors can result in a substantial increase in performance. Because of this, parallel programming has become an important part of the modern programmer's job. However, parallel programming can be complex and error-prone. One of the benefits that the stream library offers is the ability to easily—and reliably—parallel process certain operations.

Parallel processing of a stream is quite simple to request: just use a parallel stream. As mentioned earlier, one way to obtain a parallel stream is to use the **parallelStream( )** method defined by **Collection**. Another way to obtain a parallel stream is to call the **parallel( )** method on a sequential stream. The **parallel( )** method is defined by **BaseStream**, as shown here:

S parallel()

It returns a parallel stream based on the sequential stream that invokes it. (If it is called on a stream that is already parallel, then the invoking stream is returned.) Understand, of course, that even with a parallel stream, parallelism will be achieved only if the environment supports it.

Once a parallel stream has been obtained, operations on the stream can occur in parallel, assuming that parallelism is supported by the environment. For example, the first **reduce( )** operation in the preceding program can be parallelized by substituting **parallelStream( )** for the call to **stream( )**:

```
Optional<Integer> productObj = myList.parallelStream().reduce((a,b)
-> a*b);
```

The results will be the same, but the multiplications can occur in different threads.

As a general rule, any operation applied to a parallel stream must be stateless. It should also be non-interfering and associative. This ensures that the results obtained by executing operations on a parallel stream are the same as those obtained from executing the same operations on a sequential stream.

When using parallel streams, you might find the following version of **reduce( )** especially helpful. It gives you a way to specify how partial results are combined:

```
<U> U reduce(U identityVal, BiFunction<U, ? super T, U> accumulator
             BinaryOperator<U> combiner)
```

In this version, *combiner* defines the function that combines two values that have been produced by the *accumulator* function. Assuming the preceding program, the following statement computes the product of the elements in **myList** by use of a parallel stream:

```
int parallelProduct = myList.parallelStream().reduce(1, (a,b) -> a*b,
                                                     (a,b) -> a*b);
```

As you can see, in this example, both the accumulator and combiner perform the same function. However, there are cases in which the actions of the accumulator must differ from those of the combiner. For example, consider the following program. Here, **myList** contains a list of **double** values. It then uses the combiner version of **reduce( )** to compute the product of the *square roots* of each element in the list.

```
// Demonstrate the use of a combiner with reduce()

import java.util.*;
import java.util.stream.*;

class StreamDemo3 {

  public static void main(String[] args) {

    // This is now a list of double values.
    ArrayList<Double> myList = new ArrayList<>( );

    myList.add(7.0);
    myList.add(18.0);
    myList.add(10.0);
    myList.add(24.0);
    myList.add(17.0);
    myList.add(5.0);

    double productOfSqrRoots = myList.parallelStream().reduce(
                                    1.0,
                                    (a,b) -> a * Math.sqrt(b),
                                    (a,b) -> a * b
                                );

    System.out.println("Product of square roots: " + productOfSqrRoots);
  }
}
```

Notice that the accumulator function multiplies the square roots of two elements, but the combiner multiplies the partial results. Thus, the two functions differ. Moreover, for this computation to work correctly, they *must* differ. For example, if you tried to obtain the product of the square roots of the elements by using the following statement, an error would result:

```
// This won't work.
double productOfSqrRoots2 = myList.parallelStream().reduce(
                                    1.0,
                                    (a,b) -> a * Math.sqrt(b));
```

In this version of **reduce( )**, the accumulator and the combiner function are one and the same. This results in an error because when two partial results are combined, their square roots are multiplied together rather than the partial

results, themselves.

As a point of interest, if the stream in the preceding call to **reduce( )** had been changed to a sequential stream, then the operation would yield the correct answer because there would have been no need to combine two partial results. The problem occurs when a parallel stream is used.

You can switch a parallel stream to sequential by calling the **sequential( )** method, which is specified by **BaseStream**. It is shown here:

```
S sequential( )
```

In general, a stream can be switched between parallel and sequential on an as-needed basis.

There is one other aspect of a stream to keep in mind when using parallel execution: the order of the elements. Streams can be either ordered or unordered. In general, if the data source is ordered, then the stream will also be ordered. However, when using a parallel stream, a performance boost can sometimes be obtained by allowing a stream to be unordered. When a parallel stream is unordered, each partition of the stream can be operated on independently, without having to coordinate with the others. In cases in which the order of the operations does not matter, it is possible to specify unordered behavior by calling the **unordered( )** method, shown here:

```
S unordered( )
```

One other point: the **forEach( )** method may not preserve the ordering of a parallel stream. If you want to perform an operation on each element in a parallel stream while preserving the order, consider using **forEachOrdered( )**. It is used just like **forEach( )**.

Mapping

Often it is useful to map the elements of one stream to another. For example, a stream that contains a database of name, telephone, and e-mail address information might map only the name and e-mail address portions to another stream. As another example, you might want to apply some transformation to the elements in a stream. To do this, you could map the transformed elements to a new stream. Because mapping operations are quite common, the stream API provides built-in support for them. The most general mapping method is **map( )**.

It is shown here:

<R> Stream<R> map(Function<? super T, ? extends R> *mapFunc*)

Here, **R** specifies the type of elements of the new stream; **T** is the type of elements of the invoking stream; and *mapFunc* is an instance of **Function**, which does the mapping. The map function must be stateless and non-interfering. Since a new stream is returned, **map( )** is an intermediate method.

Function is a functional interface declared in **java.util.function**. It is declared as shown here:

Function<T, R>

As it relates to **map( )**, **T** is the element type and **R** is the result of the mapping. **Function** has the abstract method shown here:

R apply(T *val*)

Here, *val* is a reference to the object being mapped. The mapped result is returned.

The following is a simple example of **map( )**. It provides a variation on the previous example program. As before, the program computes the product of the square roots of the values in an **ArrayList**. In this version, however, the square roots of the elements are first mapped to a new stream. Then, **reduce( )** is employed to compute the product.

```
// Map one stream to another.

import java.util.*;
import java.util.stream.*;

class StreamDemo4 {

  public static void main(String[] args) {

    // A list of double values.
    ArrayList<Double> myList = new ArrayList<>( );

    myList.add(7.0);
    myList.add(18.0);
    myList.add(10.0);
    myList.add(24.0);
    myList.add(17.0);
    myList.add(5.0);

    // Map the square root of the elements in myList to a new stream.
    Stream<Double> sqrtRootStrm = myList.stream().map((a) -> Math.sqrt(a));

    // Find the product of the square roots.
    double productOfSqrRoots = sqrtRootStrm.reduce(1.0, (a,b) -> a*b);

    System.out.println("Product of square roots is " + productOfSqrRoots);
  }
}
```

The output is the same as before. The difference between this version and the previous is simply that the transformation (i.e., the computation of the square roots) occurs during mapping, rather than during the reduction. Because of this, it is possible to use the two-parameter form of **reduce( )** to compute the product because it is no longer necessary to provide a separate combiner function.

Here is an example that uses **map( )** to create a new stream that contains only selected fields from the original stream. In this case, the original stream contains objects of type **NamePhoneEmail**, which contains names, phone numbers, and e-mail addresses. The program then maps only the names and phone numbers to a new stream of **NamePhone** objects. The e-mail addresses are discarded.

```
// Use map() to create a new stream that contains only
// selected aspects of the original stream.

import java.util.*;
import java.util.stream.*;

class NamePhoneEmail {
  String name;
  String phonenum;
  String email;

  NamePhoneEmail(String n, String p, String e) {
    name = n;
    phonenum = p;
    email = e;
  }
}

class NamePhone {
  String name;
  String phonenum;

  NamePhone(String n, String p) {
    name = n;
    phonenum = p;
  }
}
```

```
class StreamDemo5 {

  public static void main(String[] args) {

    // A list of names, phone numbers, and e-mail addresses.
    ArrayList<NamePhoneEmail> myList = new ArrayList<>( );

    myList.add(new NamePhoneEmail("Larry", "555-5555",
                                  "Larry@HerbSchildt.com"));
    myList.add(new NamePhoneEmail("James", "555-4444",
                                  "James@HerbSchildt.com"));
    myList.add(new NamePhoneEmail("Mary", "555-3333",
                                  "Mary@HerbSchildt.com"));

    System.out.println("Original values in myList: ");
    myList.stream().forEach( (a) -> {
      System.out.println(a.name + " " + a.phonenum + " " + a.email);
    });
    System.out.println();

    // Map just the names and phone numbers to a new stream.
    Stream<NamePhone> nameAndPhone = myList.stream().map(
                                      (a) -> new NamePhone(a.name,a.phonenum)
                                     );

    System.out.println("List of names and phone numbers: ");
    nameAndPhone.forEach( (a) -> {
      System.out.println(a.name + " " + a.phonenum);
    });
  }
}
```

The output, shown here, verifies the mapping:

```
Original values in myList:
Larry 555-5555 Larry@HerbSchildt.com
James 555-4444 James@HerbSchildt.com
Mary 555-3333 Mary@HerbSchildt.com

List of names and phone numbers:
Larry 555-5555
James 555-4444
Mary 555-3333
```

Because you can pipeline more than one intermediate operation together, you

can easily create very powerful actions. For example, the following statement uses **filter( )** and then **map( )** to produce a new stream that contains only the name and phone number of the elements with the name "James":

```
Stream<NamePhone> nameAndPhone = myList.stream().
                                 filter((a) -> a.name.equals("James")).
                                 map((a) -> new NamePhone(a.name,a.phonenum));
```

This type of filter operation is very common when creating database-style queries. As you gain experience with the stream API, you will find that such chains of operations can be used to create very sophisticated queries, merges, and selections on a data stream.

In addition to the version just described, three other versions of **map( )** are provided. They return a primitive stream, as shown here:

IntStream mapToInt(ToIntFunction<? super T> *mapFunc*)

LongStream mapToLong(ToLongFunction<? super T> *mapFunc*)

DoubleStream mapToDouble(ToDoubleFunction<? super T> *mapFunc*)

Each *mapFunc* must implement the abstract method defined by the specified interface, returning a value of the indicated type. For example, **ToDoubleFunction** specifies the **applyAsDouble(T *val*)** method, which must return the value of its parameter as a **double**.

Here is an example that uses a primitive stream. It first creates an **ArrayList** of **Double** values. It then uses **stream( )** followed by **mapToInt( )** to create an **IntStream** that contains the ceiling of each value.

```
// Map a Stream to an IntStream.

import java.util.*;
import java.util.stream.*;

class StreamDemo6 {

  public static void main(String[] args) {

    // A list of double values.
    ArrayList<Double> myList = new ArrayList<>( );

    myList.add(1.1);
    myList.add(3.6);
    myList.add(9.2);
    myList.add(4.7);
    myList.add(12.1);
    myList.add(5.0);

    System.out.print("Original values in myList: ");
    myList.stream().forEach( (a) -> {
      System.out.print(a + " ");
    });
    System.out.println();

    // Map the ceiling of the elements in myList to an IntStream.
    IntStream cStrm = myList.stream().mapToInt((a) -> (int) Math.ceil(a));

    System.out.print("The ceilings of the values in myList: ");
    cStrm.forEach( (a) -> {
      System.out.print(a + " ");
    });

  }
}
```

The output is shown here:

```
Original values in myList: 1.1 3.6 9.2 4.7 12.1 5.0
The ceilings of the values in myList: 2 4 10 5 13 5
```

The stream produced by **mapToInt( )** contains the ceiling values of the original elements in **myList**.

Before leaving the topic of mapping, it is necessary to point out that the

stream API also provides methods that support *flat maps*. These are **flatMap( )**, **flatMapToInt( )**, **flatMapToLong( )**, and **flatMapToDouble( )**. The flat map methods are designed to handle situations in which each element in the original stream is mapped to more than one element in the resulting stream.

Collecting

As the preceding examples have shown, it is possible (indeed, common) to obtain a stream from a collection. Sometimes it is desirable to obtain the opposite: to obtain a collection from a stream. To perform such an action, the stream API provides the **collect( )** method. It has two forms. The one we will use first is shown here:

<R, A> R collect(Collector<? super T, A, R> *collectorFunc*)

Here, **R** specifies the type of the result, and **T** specifies the element type of the invoking stream. The internal accumulated type is specified by **A**. The *collectorFunc* specifies how the collection process works. The **collect( )** method is a terminal operation.

The **Collector** interface is declared in **java.util.stream**, as shown here:

interface Collector<T, A, R>

T, **A**, and **R** have the same meanings as just described. **Collector** specifies several methods, but for the purposes of this chapter, we won't need to implement them. Instead, we will use two of the predefined collectors that are provided by the **Collectors** class, which is packaged in **java.util.stream**.

The **Collectors** class defines a number of static collector methods that you can use as-is. The two we will use are **toList( )** and **toSet( )**, shown here:

static <T> Collector<T, ?, List<T>> toList( )

static <T> Collector<T, ?, Set<T>> toSet( )

The **toList( )** method returns a collector that can be used to collect elements into a **List**. The **toSet( )** method returns a collector that can be used to collect elements into a **Set**. For example, to collect elements into a **List**, you can call **collect( )** like this:

```
collect(Collectors.toList())
```

The following program puts the preceding discussion into action. It reworks the example in the previous section so that it collects the names and phone numbers into a **List** and a **Set**.

```
// Use collect() to create a List and a Set from a stream.

import java.util.*;
import java.util.stream.*;

class NamePhoneEmail {
  String name;
  String phonenum;
  String email;

  NamePhoneEmail(String n, String p, String e) {
    name = n;
    phonenum = p;
    email = e;
  }
}

class NamePhone {
  String name;
  String phonenum;

  NamePhone(String n, String p) {
    name = n;
    phonenum = p;
  }
}

class StreamDemo7 {

  public static void main(String[] args) {

    // A list of names, phone numbers, and e-mail addresses.
    ArrayList<NamePhoneEmail> myList = new ArrayList<>( );

    myList.add(new NamePhoneEmail("Larry", "555-5555",
                                  "Larry@HerbSchildt.com"));
    myList.add(new NamePhoneEmail("James", "555-4444",
                                  "James@HerbSchildt.com"));
    myList.add(new NamePhoneEmail("Mary", "555-3333",
                                  "Mary@HerbSchildt.com"));

    // Map just the names and phone numbers to a new stream.
    Stream<NamePhone> nameAndPhone = myList.stream().map(
                                       (a) -> new NamePhone(a.name,a.phonenum)
                                     );

    // Use collect to create a List of the names and phone numbers.
    List<NamePhone> npList = nameAndPhone.collect(Collectors.toList());
```

```
    System.out.println("Names and phone numbers in a List:");
    for(NamePhone e : npList)
      System.out.println(e.name + ": " + e.phonenum);

    // Obtain another mapping of the names and phone numbers.
    nameAndPhone = myList.stream().map(
                                    (a) -> new NamePhone(a.name,a.phonenum)
                                   );

    // Now, create a Set by use of collect().
    Set<NamePhone> npSet = nameAndPhone.collect(Collectors.toSet());

    System.out.println("\nNames and phone numbers in a Set:");
    for(NamePhone e : npSet)
      System.out.println(e.name + ": " + e.phonenum);
  }
}
```

The output is shown here:

```
Names and phone numbers in a List:
Larry: 555-5555
James: 555-4444
Mary: 555-3333

Names and phone numbers in a Set:
James: 555-4444
Larry: 555-5555
Mary: 555-3333
```

In the program, the following line collects the name and phone numbers into a **List** by using **toList( )**:

```
List<NamePhone> npList = nameAndPhone.collect(Collectors.toList());
```

After this line executes, the collection referred to by **npList** can be used like any other **List** collection. For example, it can be cycled through by using a for-each **for** loop, as shown in the next line:

```
for(NamePhone e : npList)
  System.out.println(e.name + ": " + e.phonenum);
```

The creation of a **Set** via **collect(Collectors.toSet( ))** works in the same way.

The ability to move data from a collection to a stream, and then back to a collection again is a very powerful attribute of the stream API. It gives you the ability to operate on a collection through a stream, but then repackage it as a collection. Furthermore, the stream operations can, if appropriate, occur in parallel.

The version of **collect( )** used by the previous example is quite convenient, and often the one you want, but there is a second version that gives you more control over the collection process. It is shown here:

```
<R> R collect(Supplier<R> target, BiConsumer<R, ? super T> accumulator,
              BiConsumer <R, R> combiner)
```

Here, *target* specifies how the object that holds the result is created. For example, to use a **LinkedList** as the result collection, you would specify its constructor. The *accumulator* function adds an element to the result and *combiner* combines two partial results. Thus, these functions work similarly to the way they do in **reduce( )**. For both, they must be stateless and non-interfering. They must also be associative.

Note that the *target* parameter is of type **Supplier**. It is a functional interface declared in **java.util.function**. It specifies only the **get( )** method, which has no parameters and, in this case, returns an object of type **R**. Thus, as it relates to **collect( )**, **get( )** returns a reference to a mutable storage object, such as a collection.

Note also that the types of *accumulator* and *combiner* are **BiConsumer**. This is a functional interface defined in **java.util.function**. It specifies the abstract method **accept( )** that is shown here:

```
void accept(T obj, U obj2)
```

This method performs some type of operation on *obj* and *obj2*. As it relates to *accumulator*, *obj* specifies the target collection, and *obj2* specifies the element to add to that collection. As it relates to *combiner*, *obj* and *obj2* specify two collections that will be combined.

Using the version of **collect( )** just described, you could use a **LinkedList** as the target in the preceding program, as shown here:

```
LinkedList<NamePhone> npList = nameAndPhone.collect(
                                () -> new LinkedList<>(),
                                (list, element) -> list.add(element),
                                (listA,listB ) -> listA.addAll(listB));
```