INTEGRATED POLYTECHNIC REGIONAL CENTER (IPRC), KIGALI
KICUKIRO – CAMPUS

ICT 311
Advanced Java Programming

Year 3 ICT

IPRC/ KCT , Kigali

Prepared by:

Pascal BENIMANA

Academic Year: 2013-2014

[*P.O.Box: 6579 Kigali- Rwanda*
www.iprckigali.ac.rw

Contents

Chapter I: The Applet Class and Event Handling

1.1. Applet Basics

All applets are subclasses (either directly or indirectly) of **Applet**. Applets are not stand-alone programs. Instead, they run within either a web browser or an applet viewer.

There are two varieties of applets.

The first are those based directly on the **Applet** class. These applets use the Abstract Window Toolkit (AWT) to provide the graphic user interface (or use no GUI at all). This style of applet has been available since Java was first created.

The second type of applets are those based on the Swing class **JApplet**. Swing applets use the Swing classes to provide the GUI. Swing offers a richer and often easier-to-use user interface than does the AWT. Thus, Swing-based applets are now the most popular. However, traditional AWT-based applets are still used, especially when only a very simple user interface is required.

Thus, both AWT- and Swing-based applets are valid.

1.2. The Applet Class

Applet provides all necessary support for applet execution, such as starting and stopping. It also provides methods that load and display images, and methods that load and play audio clips. **Applet** extends the AWT class **Panel**. In turn, **Panel** extends **Container**, which extends **Component**. These classes provide support for Java's window-based, graphical interface. Thus, **Applet** provides all of the necessary support for window-based activities.

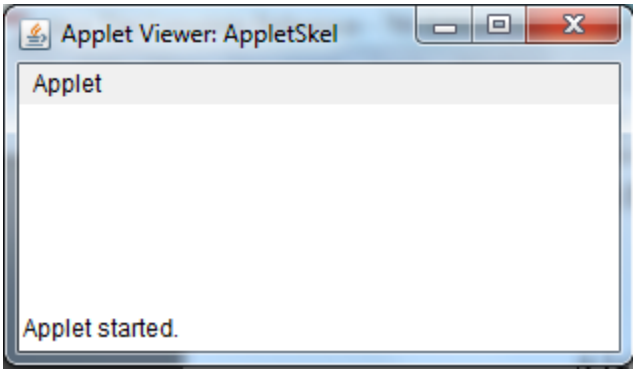
1.3 The Applet Skeleton

All but the most trivial applets override a set of methods that provides the basic mechanism by which the browser or applet viewer interfaces to the applet and controls its execution. Four of these methods, **init( )**, **start( )**, **stop( )**, and **destroy( )**, apply to all applets and are defined by **Applet**. Applets do not need to override those methods they do not use. However, only very simple applets will not need to define all of them.

AWT-based applets (such as those discussed in this chapter) will also override the **paint( )** method, which is defined by the AWT **Component** class. (Swing-based applets use a different mechanism to accomplish this task.) These five methods can be assembled into the skeleton shown here:

```
//An Applet skeleton.
import java.applet.Applet;
import java.awt.Graphics;

/*
 <applet code="AppletSkel" width=300 height=100>
 </applet>
 */
class AppletSkel extends Applet {

	// Called first.
	public void init() {
		// initialization
	}

	/*
	 * Called second, after init(). Also called whenever the applet is
	 * restarted.
	 */
	public void start() {
		// start or resume execution
	}

	// Called when the applet is stopped.
	public void stop() {
		// suspends execution
	}

	/*
	 * Called when applet is terminated. This is the last method executed.
	 */
	public void destroy() {
		// perform shutdown activities
	}

	// Called when an applet's window must be restored.
	public void paint(Graphics g) {
		// redisplay contents of window
	}

}
```

Although this skeleton does not do anything, it can be compiled and run. When run, it generates the following window when viewed with an applet viewer:



Applet Initialization and Termination

When an applet begins, the following methods are called, in this sequence:

1. **init( )**

2. **start( )**

3. **paint( )**

When an applet is terminated, the following sequence of method calls takes place:

1. **stop( )**

2. **destroy( )**

Let's look more closely at these methods.

init( )

The **init( )** method is the first method to be called. This is where you should initialize variables. This method is called only once during the run time of your applet.

start( )

The **start( )** method is called after **init( )**. It is also called to restart an applet after it has been stopped. Whereas **init( )** is called once during the applet's run time, **start( )** is called each time an applet's HTML document is displayed on screen. So, if a user leaves a web page and comes back, the applet resumes execution at **start( )**.

paint( )

The **paint( )** method is called each time your applet's output must be redrawn. The **paint( )** method has one parameter of type **Graphics**. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.

stop( )

The **stop( )** method is called when a web browser leaves the HTML document containing the applet; when it goes to another page, for example. When **stop( )** is called, the applet is probably running. You should use **stop( )** to suspend threads that don't need to run when the applet is not visible. You can restart them when **start( )** is called if the user returns to the page.

destroy( )

The **destroy( )** method is called when the environment determines that your applet needs to be removed completely from memory. The **stop( )** method is always called before **destroy( )**.

1.4 Simple Applet Display Methods

To output a string to an applet, use **drawString( )**, which is a member of the **Graphics** class. Typically, it is called from within either **update( )** or **paint( )**.

It has the following general form:

```
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.

To set the background color of an applet's window, 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)
```

The following example sets the background color to green and the text color to red:

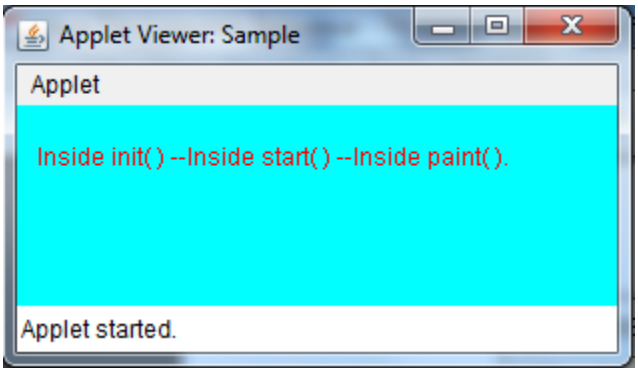
```
setBackground(Color.green);
setForeground(Color.red);
```

A good place to set the foreground and background colors is in the init( ) method.

Here is a very simple applet that sets the background color to cyan, the foreground color to red, and displays a message that illustrates the order in which the **init( )**, **start( )**, and **paint( )** methods are called when an applet starts up:

```
/* A simple applet that sets the foreground and
background colors and outputs a string. */
import java.applet.Applet;
import java.awt.Color;
import java.awt.Graphics;

/*
 <applet code="Sample" width=300 height=50>
 </applet>
 */
public class Sample extends Applet {
      String msg;

      // set foreground and background colors
      public void init() {
            setBackground(Color.cyan);
            setForeground(Color.red);
            msg = "Inside init( ) --";
            //repaint();
      }

      // initialize the string to be displayed.
      public void start() {
            msg += "Inside start( ) --";
            //repaint();
      }


      // Display message in the applet window.
      public void paint(Graphics g) {
            g.drawString(msg+"inside paint( ).", 10, 30);
            //repaint();
      }

}
```

This applet generates the window shown here:



The methods **stop( )** and **destroy( )** are not overridden, because they are not needed by this simple applet.

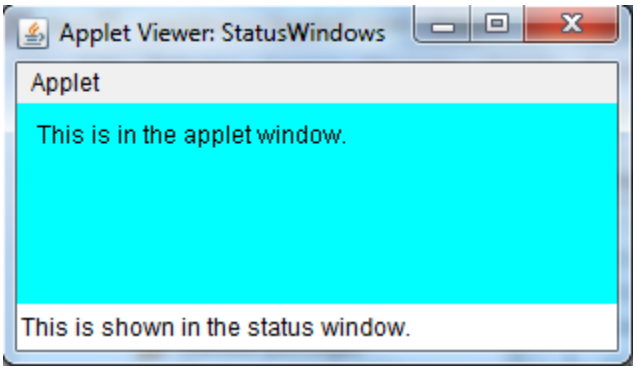
Applets with status window

In addition to displaying information in its window, an applet can also output a message to the status window of the browser or applet viewer on which it is running. To do so, call **`showStatus( )`** with the string that you want displayed. The status window is a good place to give the user feedback about what is occurring in the applet, suggest options, or possibly report some types of errors.

The following program demonstrates showstatus().

```
// Using the Status Window.
import java.applet.Applet;
import java.awt.Color;
import java.awt.Graphics;

/*
 <applet code="StatusWindow" width=300 height=50>
 </applet>
 */
public class StatusWindows extends Applet {
      public void init() {
            setBackground(Color.cyan);
      }

      // Display msg in applet window.
      public void paint(Graphics g) {
            g.drawString("This is in the applet window.", 10, 20);
            showStatus("This is shown in the status window.");
      }

}
```

Sample output from this program is shown here:



getDocumentBase( ) and getCodeBase( )

Java allows the applet to load data from the directory holding the HTML file that started the applet (the d*ocument base*) and the directory from which the applet's class file was loaded (the *code base*). These directories are returned as **URL** objects by **getDocumentBase( )** and **getCodeBase( )**.

The following applet illustrates these methods:

```
// Display code and document bases.
import java.applet.Applet;
import java.awt.Graphics;
import java.net.URL;
/*
<applet code="Bases" width=300 height=50>
</applet>
*/
public class Bases extends Applet {
      //Display code and document bases.
      public void paint(Graphics g){
            String msg;
            URL url=getCodeBase(); //get code base
            msg="Code base: "+url.toString();
            g.drawString(msg, 10, 20);

            url=getDocumentBase();//get document base
            msg="Document base: "+url.toString();
            g.drawString(msg, 10, 40);
      }

}
```

1.5 Event Handling

When an application or a program keeps on monitoring and quickly responds to any action that occurs at the GUI interface like mouse movement, selecting an item in a list or entering a keyboard input and so on then such a scenario is termed as *event handling*. In java the events from the event sources are captured and they are sent to event listeners for respective actions to be taken.

Events

an *event* is an object that describes a state change in a source. It can be generated as a consequence of a person interacting with the elements in a graphical user interface.

Most programs, to be useful, must respond to commands from the user. To do so, Java programs rely on events that describe user actions.

Some of the activities that cause events to be generated are pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse. Many other user operations could also be cited as examples.

Events may also occur that are not directly caused by interactions with a user interface. For example, an event may be generated when a timer expires, a counter exceeds a value, a software or hardware failure occurs, or an operation is completed. You are free to define events that are appropriate for your application.

Event Sources

An event source is an object that generates an event. Sources may generate more than one type of event. An event source must register listeners in order for the listeners to receive notifications about a specific type of event. Each type of event has its own registration method. Here is the general form:

```
public void addTypeListener(TypeListener el)
```

Here, *Type* is the name of the event, and *el* is a reference to the event listener. For example, the method that registers a keyboard event listener is called **addKeyListener( )**. The method that registers a mouse motion listener is called **addMouseMotionListener( )**.

Event Listeners

A *listener* is an object that is notified when an event occurs. It has two major requirements. First, it must have been registered with one or more sources to receive notifications about specific types of events. Second, it must implement methods to receive and process these notifications.

The methods that receive and process events are defined in a set of interfaces found in **java.awt.event**. For example, the **MouseMotionListener** interface defines two methods to receive notifications when the mouse is dragged or moved.

1.6 Event Classes

The classes that represent events are at the core of Java's event handling mechanism. Thus, a discussion of event handling must begin with the event classes.

At the root of the Java event class hierarchy is **EventObject**, which is in **java.util**. It is the superclass for all events. Its one constructor is shown here:

```
EventObject(Object src)
```

Here, *src* is the object that generates this event.

EventObject contains two methods: **getSource( )** and **toString( )**. The **getSource( )** method returns the source of the event. Its general form is shown here:

```
Object getSource( )
```

As expected, **toString( )** returns the string equivalent of the event.

The class **AWTEvent**, defined within the **java.awt** package, is a subclass of **EventObject**. It is the superclass (either directly or indirectly) of all AWT-based events.

Its **getID( )** method can be used to determine the type of the event. The signature of this method is shown here:

```
int getID( )
```

To summarize:

- **EventObject** is a superclass of all events.
- **AWTEvent** is a superclass of all AWT events.

The package **java.awt.event** defines many types of events that are generated by various user interface elements. Table 1 shows several commonly used event classes and provides a brief description of when they are generated.

Event	**Class Description**
ActionEvent	Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
AdjustmentEvent	Generated when a scroll bar is manipulated.
ComponentEvent	Generated when a component is hidden, moved, resized, or becomes visible.
ContainerEvent	Generated when a component is added to or removed from a container.
FocusEvent	Generated when a component gains or loses keyboard focus.
InputEvent	Abstract superclass for all component input event classes.
ItemEvent	Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected.
KeyEvent	Generated when input is received from the keyboard.
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.
MouseWheelEvent	Generated when the mouse wheel is moved.
TextEvent	Generated when the value of a text area or text field is changed.
WindowEvent	Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Table 1: Main Event Classes in java.awt.event

1.7 Event Listener Interfaces

Listeners are created by implementing one or more of the interfaces defined by the j**ava.awt.event** package. Table 2 lists commonly used listener interfaces and provides a brief description of the methods that they define.

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

Table 2: Commonly Used Event Listener Interfaces

1.7.1 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)
```

1.7.2 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)
```

1.7.3 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)
```

1.7.4 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)
```

1.7.5 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)
```

1.7.6 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)
```

1.7.7 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)
```

1.7.8 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)
```

1.7.9 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)
```

1.7.10 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)
```

1.7.11 The TextListener Interface

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

```
void textChanged(TextEvent te)
```

1.7.12 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)
```

1.7.13 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)
```

1.8 Events Handling Examples

1.8.1 Handling Mouse Events

To handle mouse events, you must implement the **MouseListener** and the **MouseMotionListener** interfaces.

The following applet demonstrates the process. It displays the current coordinates of the mouse in the applet's status window. Each time a button is pressed, the word "*Down*" is displayed at the location of the mouse pointer. Each time the button is released, the word "*Up*" is shown. If a button is clicked, the message "Mouse clicked" is displayed in the upperleft corner of the applet display area.

As the mouse enters or exits the applet window, a message is displayed in the upper-left corner of the applet display area. 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.

```
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;

public class MouseEvents extends Applet implements MouseListener,
MouseMotionListener{
	String msg="";
	int mouseX=0, mouseY=0;//coordinates of mouse
	public void init(){
		addMouseListener(this);
		addMouseMotionListener(this);
	}
	@Override
	public void mouseClicked(MouseEvent me) {// Handle mouse clicked.
		// Save coordinates
		mouseX=0;
		mouseY=10;
		msg="Mouse clicked";
		repaint();
	}
	@Override
	public void mouseEntered(MouseEvent me) {// Handle mouse entered.
		// Save coordinates
		mouseX=0;
		mouseY=10;
		msg="Mouse entered";
		repaint();
	}
	@Override
	public void mouseExited(MouseEvent me) {// Handle mouse
		// Save coordinates
		mouseX=10;
		mouseY=10;
		msg="Mouse exited";
		repaint();
	}
	@Override
	public void mousePressed(MouseEvent me) {// Handle mouse pressed
		// Save coordinates
		mouseX=me.getX();
		mouseY=me.getY();
		msg="Down";
		repaint();
	}
	@Override
	public void mouseReleased(MouseEvent me) {// handle mouse released
		// Save coordinates
		mouseX=me.getX();
```

```
            mouseY=me.getY();
            msg="Up";
            repaint();
        }

        @Override
        public void mouseDragged(MouseEvent e) {// Handle mouse dragged
            // Save coordinates
            mouseX=e.getX();
            mouseY=e.getY();
            msg="*";
            showStatus("Dragging mouse at: "+mouseX+" , "+mouseY);
            repaint();
        }
        @Override
        public void mouseMoved(MouseEvent e) { //Handle mouse moved
            // Save coordinates
            mouseX=e.getX();
            mouseY=e.getY();
            showStatus("Moving mouse at: "+mouseX+" , "+mouseY);
        }
        //Display msg in applet window at current X,Y location.
        public void paint(Graphics g){
            g.drawString(msg, mouseX, mouseY);
            //showStatus(arg0)
        }
}
```

Let's look closely at this example. The **MouseEvents** class extends **Applet** and implements both the **MouseListener** and **MouseMotionListener** interfaces. Notice that the applet is both the source and the listener for these events. This works because **Component**, which supplies the **addMouseListener( )** and **addMouseMotionListener( )** methods, is a superclass of **Applet**. Being both the source and the listener for events is a common situation for applets.

Inside **init( )**, the applet registers itself as a listener for mouse events. This is done by using **addMouseListener( )** and **addMouseMotionListener( )**, which, as mentioned, are members of **Component**. They are shown here:

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

1.8.2 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.

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 keypress 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 applet window and shows the pressed/released status of each key in the status window.

```
// Demonstrate the key event handlers.
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
/*
<applet code="SimpleKey" width=300 height=100>
</applet>
*/
public class SimpleKey extends Applet implements KeyListener {
	String msg = "";
	int mouseX = 10, mouseY = 20;//Output coordinates

	public void init() {
		addKeyListener(this);
	}

	@Override
	public void keyPressed(KeyEvent ke) {
		// TODO Auto-generated method stub
		showStatus("Key Down");
	}

	@Override
	public void keyReleased(KeyEvent ke) {
		// TODO Auto-generated method stub
```

```
            showStatus("key Up");
        }

        @Override
        public void keyTyped(KeyEvent ke) {
            // TODO Auto-generated method stub
            msg += ke.getKeyChar();
            repaint();
        }
        // Display kestrokes.
        public void paint(Graphics g) {
            g.drawString(msg, mouseX, mouseY);
        }
}
```

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 applet outputs the name of a few of the special keys:

```
// Demonstrate some virtual key codes.
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
/*
<applet code="KeyEvents" width=300 height=100>
</applet>
*/
public class KeyEvents extends Applet implements KeyListener {
      String msg = "";
      int x = 10, y = 20;

      public void init() {
            addKeyListener(this);
      }

      @Override
      public void keyPressed(KeyEvent ke) {
            // TODO Auto-generated method stub
            showStatus("Key Down");
            int k = ke.getKeyCode();
            switch (k) {
            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+="<pg dn>";
                        break;
                case KeyEvent.VK_PAGE_UP:
                        msg+="<pg up>";
                        break;
                case KeyEvent.VK_LEFT:
                        msg+="<Left Arrow>";
                        break;
                case KeyEvent.VK_RIGHT:
                        msg += "<Right Arrow>";
                        break;
                default:
                        break;
                }
                repaint();

        }
        @Override
        public void keyReleased(KeyEvent ke) {
                // TODO Auto-generated method stub
                showStatus("Key Up");
        }
        @Override
        public void keyTyped(KeyEvent ke) {
                // TODO Auto-generated method stub
                msg += ke.getKeyChar();
                repaint();
        }
        // Display keystrokes.
        public void paint(Graphics g) {
        g.drawString(msg, x, y);
        }
}
```

Chapter II: Basics of AWT

The AWT contains numerous classes and methods that allow you to create and manage windows. It is also the foundation upon which Swing is built. 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 in one chapter.

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**.
Event	Encapsulates events.
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.

Table 3: A Sampling of AWT Classes

2.1 AWT Classes

The AWT classes are contained in the **java.awt** package. It is one of Java's largest packages. Table 3 and table 4 lists some of the many AWT classes.

Class	Description
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 4: A Sampling of AWT Classes (continued)

2.2 Window Fundamentals

The AWT defines windows according to a class hierarchy that adds functionality and specificity with each level. The two most common windows are those derived from **Panel**, which is used by applets, and those derived from **Frame**, which creates a standard application window.

2.2.1 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. 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.

2.2.2 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.

2.2.3 Panel

The Panel class is a concrete subclass of Container. It doesn't add any new methods. Panel is the superclass for Applet. When screen output is directed to an applet, it is drawn on the surface of a Panel object. In essence, a Panel is a window that does not contain a title bar, menu bar, or border. That is why you don't see these items when an applet is run inside a browser. When you run an applet using an applet viewer, the applet viewer provides the title and border.

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.

2.2.4 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.

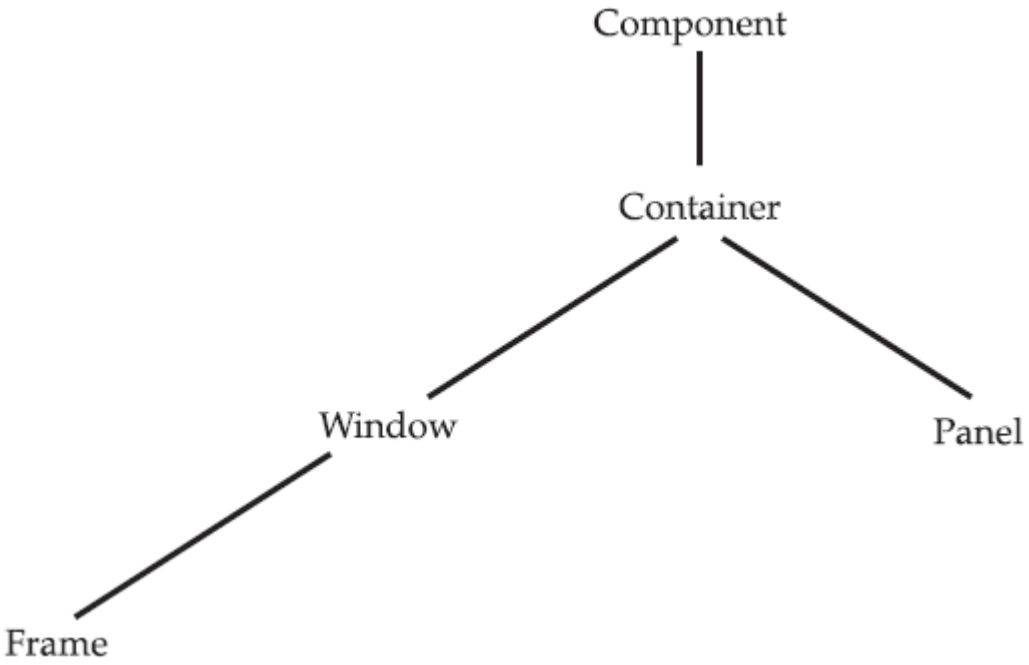


Figure 1: The class hierarchy for Panel and Frame

2.2.5 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. If you create a **Frame** object from within an applet, it will contain a warning message, such as "Java Applet Window," to the user that an applet window has been created. This message warns users that the window they see was started by an applet and not by software running on their computer.

When a Frame window is created by a stand-alone application rather than an applet, a normal window is created.

2.2.6 Canvas

Although it is not part of the hierarchy for applet or frame windows, there is one other type of window that you will find valuable: **Canvas**. Canvas encapsulates a blank window upon which you can draw.

2.3 Working with Frame Windows

After the applet, the type of window you will most often create is derived from **Frame**. Here are two of **Frame**'s constructors:

```
Frame( )
Frame(String title)
```

Setting the Window's Dimensions

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

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

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

```
Dimension getSize( )
```

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)
```

Closing a Frame Window

When using a frame window, your program must remove that window from the screen when it is closed, by calling **setVisible(false)**. To intercept a window-close event, you must implement the **windowClosing( )** method of the **WindowListener** interface. Inside **windowClosing( )**, you must remove the window from the screen.

2.4 AWT Controls, Layout Managers, and Menus

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. A menu bar is always 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.

2.4.1 Controls

The AWT supports the following types of controls:

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

These controls are subclasses of **Component**.

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**.

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**. It has this general form:

```
void remove(Component obj)
```

You can remove all controls by calling **`removeAll( )`**.

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( )
Label(String str)
Label(String str, int how)
```

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( )
```

The following example creates three labels and adds them to an applet window:

```
import java.applet.Applet;
import java.awt.Label;
/*
<applet code="LabelDemo" width=300 height=200>
</applet>
*/
public class LabelDemo extends Applet {
      public void init() {
            Label one = new Label("One");
            Label two = new Label("Two");
            Label three = new Label("Three");
            // add labels to applet window
            add(one);
            add(two);
            add(three);
      }

}
```

Buttons

A *push button* is a component that contains a label and that 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( )
```

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.

Here is an example that creates three buttons labeled “Yes”, “No”, and “May be”. 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.applet.Applet;
import java.awt.Button;
import java.awt.Graphics;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
/*
<applet code="ButtonDemo" width=250 height=150>
</applet>
*/

public class ButtonDemo extends Applet implements ActionListener {
	String msg;
	Button b1, b2, b3;
	public void init() {
		b1=new Button("Yes");
		b2=new Button();
		b3=new Button();
		b2.setLabel("No");
		b3.setLabel("May be");
		add(b1);
		add(b2);
		add(b3);
		b1.addActionListener(this);
		b2.addActionListener(this);
		b3.addActionListener(this);
	}

	@Override
	public void actionPerformed(ActionEvent ae) {
```

```
            // TODO Auto-generated method stub
            String s=ae.getActionCommand();
            if(s.equals("Yes"))
                  msg="You pressed yes";
            else if(s.equals("No"))
                  msg="You pressed No";
            else
                  msg="You pressed may be";
            repaint();
      }
      public void paint(Graphics g){
            g.drawString(msg, 6, 100);
      }
}
```

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( )`
- `Checkbox(String str)`
- `Checkbox(String str, boolean on)`
- `Checkbox(String str, boolean on, CheckboxGroup cbGroup)`
- `Checkbox(String str, CheckboxGroup cbGroup, boolean on)`

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.

```
import java.applet.Applet;
import java.awt.Checkbox;
import java.awt.Graphics;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
/*
 <applet code="CheckboxDemo" width=250 height=200>
 </applet>
 */
public class CheckboxDemo extends Applet implements ItemListener {
      String msg = "";
      Checkbox winXP, winVista, solaris, mac;

      public void init() {
            winXP = new Checkbox("Windows XP", null, true);
            winVista = new Checkbox("Windows Vista");
            solaris = new Checkbox("Solaris");
            mac = new Checkbox("Mac OS");
```

```
            add(winXP);
            add(winVista);
            add(solaris);
            add(mac);
            winXP.addItemListener(this);
            winVista.addItemListener(this);
            solaris.addItemListener(this);
            mac.addItemListener(this);
      }

      @Override
      public void itemStateChanged(ItemEvent arg0) {
            repaint();
      }
      // Display current state of the check boxes.
      public void paint(Graphics g) {
            msg = "Current state: ";
            g.drawString(msg, 6, 80);
            msg = " Windows XP: " + winXP.getState();
            g.drawString(msg, 6, 100);
            msg = " Windows Vista: " + winVista.getState();
            g.drawString(msg, 6, 120);
            msg = " Solaris: " + solaris.getState();
            g.drawString(msg, 6, 140);
            msg = " Mac OS: " + mac.getState();
            g.drawString(msg, 6, 160);
      }

}
```

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:

```
import java.applet.Applet;
import java.awt.Checkbox;
import java.awt.CheckboxGroup;
import java.awt.Graphics;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;

public class CBGroup extends Applet implements ItemListener {
	/*
	 * <applet code="CBGroup" width=250 height=200> </applet>
	 */
	String msg = "";
	Checkbox winXP, winVista, solaris, mac;
	CheckboxGroup cbg;

	public void init() {
		cbg = new CheckboxGroup();
		winXP = new Checkbox("Windows XP", cbg, true);
		winVista = new Checkbox("Windows Vista", cbg, false);
		solaris = new Checkbox("Solaris", cbg, false);
		mac = new Checkbox("Mac OS", cbg, false);
		add(winXP);
		add(winVista);
		add(solaris);
		add(mac);
		winXP.addItemListener(this);
		winVista.addItemListener(this);
		solaris.addItemListener(this);
		mac.addItemListener(this);
	}
	@Override
	public void itemStateChanged(ItemEvent e) {
		repaint();
	}
	// Display current state of the check boxes.
	public void paint(Graphics g) {
		msg = "Current selection: ";
		msg += cbg.getSelectedCheckbox().getLabel();
		g.drawString(msg, 6, 100);
	}

}
```

Choice Controls

The **Choice** class is used to create a *pop-up list* of items from which the user may choose. 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. **Choice** only defines 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

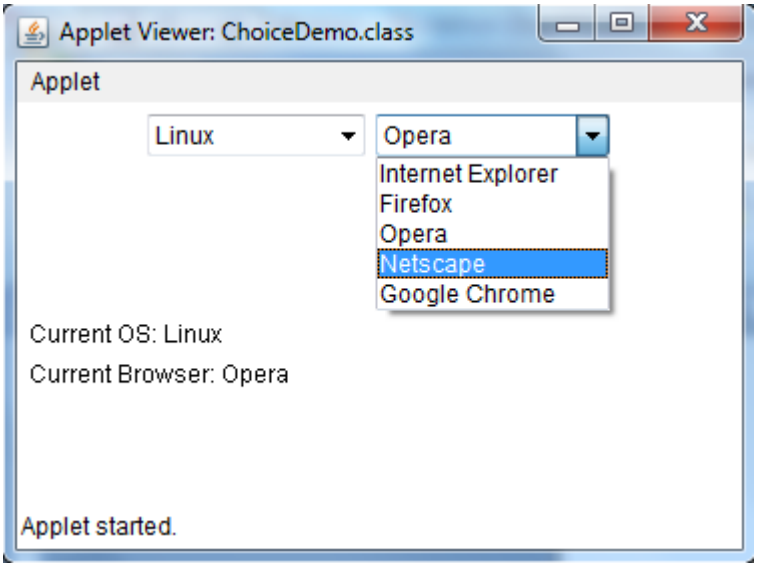
Each time a choice is selected, an item event is generated. This is sent to any listeners that previously registered an interest in receiving item event notifications from that component. Each listener implements the **ItemListener** interface. That interface defines the **itemStateChanged( )** method.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

```
import java.applet.Applet;
import java.awt.Choice;
import java.awt.Graphics;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
/*
<applet code="ChoiceDemo" width=300 height=180>
</applet>
*/
public class ChoiceDemo extends Applet implements ItemListener {
	Choice os, browser;
	String msg="";
	public void init(){
		os=new Choice();
		browser=new Choice();
		//add items to os list
		os.add("Windows XP");
		os.add("Windows Vista");
		os.add("Windows 7");
		os.add("Linux");
		os.add("Solaris");
		os.add("Mac OS");

		//add items to browser list
		browser.add("Internet Explorer");
		browser.add("Firefox");
		browser.add("Opera");
		browser.add("Netscape");
		browser.add("Google Chrome");

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

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

```
        //Display current selections.
        public void paint(Graphics g){
                msg="Current OS: ";
                msg+=os.getSelectedItem();
                g.drawString(msg, 6, 120);
                msg="Current Browser: ";
                msg+=browser.getSelectedItem();
                g.drawString(msg, 6, 140);
        }

}
```

Sample output:



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( )
List(int numRows)
List(int numRows, boolean multipleSelect)
```

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 *multipleSelec*t

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.

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( )
```

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( )**.

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:

```
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.List;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
/*
```

```
<applet code="ListDemo" width=300 height=180>
</applet>
*/
public class ListDemo extends Applet implements ActionListener {
	List os, browser;
	String msg="";
	public void init(){
		os=new List(4,true);
		browser=new List(4,false);
		//add items to os list
		os.add("Windows XP");
		os.add("Windows Vista");
		os.add("Windows 7");
		os.add("Linux");
		os.add("Solaris");
		os.add("Mac OS");

		//add items to browser list
		browser.add("Internet Explorer");
		browser.add("Firefox");
		browser.add("Opera");
		browser.add("Netscape");
		browser.add("Google Chrome");

		browser.select(1);
		//add choice lists to window
		add(os);
		add(browser);
		// register to receive action events
		os.addActionListener(this);
		browser.addActionListener(this);

	}
	@Override
	public void actionPerformed(ActionEvent arg0) {
		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);
	}

}
```

Scroll Bars

Scroll bars are used to select continuous values between a specified minimum and maximum. Scroll bars may be oriented horizontally or vertically. Scroll bars are encapsulated by the **Scrollbar** class.

Scrollbar defines the following constructors:

- `Scrollbar( )`
- `Scrollbar(int style)`
- `Scrollbar(int style, int initialValue, int thumbSize, int min, int max)`

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.

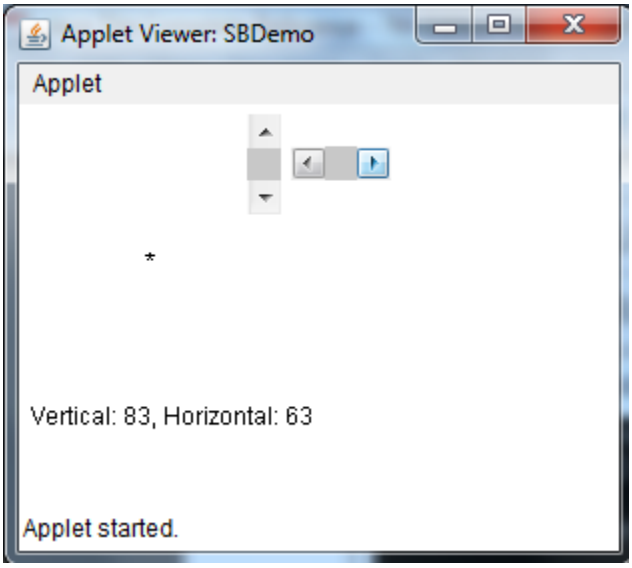
The following example creates both a vertical and a horizontal scroll bar. The current settings of the scroll bars are displayed. If you drag the mouse while inside the window, the coordinates of

each drag event are used to update the scroll bars. An asterisk is displayed at the current drag position.

```
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.Scrollbar;
import java.awt.event.AdjustmentEvent;
import java.awt.event.AdjustmentListener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseMotionListener;
/*
<applet code="SBDemo" width=300 height=200>
</applet>
*/
public class SBDemo extends Applet implements AdjustmentListener,
            MouseMotionListener {
      String msg = "";
      Scrollbar vertSB, horzSB;

      public void init() {
            int width = Integer.parseInt(getParameter("width"));
            int height = Integer.parseInt(getParameter("height"));
            vertSB = new Scrollbar(Scrollbar.VERTICAL, 0, 1, 0, height);
            horzSB = new Scrollbar(Scrollbar.HORIZONTAL, 0, 1, 0, width);
            add(vertSB);
            add(horzSB);
            // register to receive adjustment events
            vertSB.addAdjustmentListener(this);
            horzSB.addAdjustmentListener(this);
            addMouseMotionListener(this);
      }

      // Update scroll bars to reflect mouse dragging.
      @Override
      public void mouseDragged(MouseEvent e) {
            // TODO Auto-generated method stub
            int x = e.getX();
            int y = e.getY();
            vertSB.setValue(y);
            horzSB.setValue(x);
            repaint();
      }

      // Necessary for MouseMotionListener
      @Override
      public void mouseMoved(MouseEvent e) {
            // TODO Auto-generated method stub
      }

      @Override
      public void adjustmentValueChanged(AdjustmentEvent ae) {
            // TODO Auto-generated method stub
            repaint();
      }
```

```
        // Display current value of scroll bars.
        public void paint(Graphics g) {
                msg = "Vertical: " + vertSB.getValue();
                msg += ", Horizontal: " + horzSB.getValue();
                g.drawString(msg, 6, 160);
                // show current mouse drag position
                g.drawString("*", horzSB.getValue(), vertSB.getValue());
        }
}
```

Sample output:



TextField (Implements ActionListener)

The text field implements a single line text entry area, usually called an edit control.

TextField defines the following constructors:

```
TextField( )
TextField(int numChars)
TextField(String str)
TextField(String str, int numChars)
```

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.

To obtain the string currently contained in the text field, call getText() and to set the text, call setText().

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 username and password screen

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

/*
 <applet code="TextFieldDemo" width=380 height=150>
 </applet>
 */
public class TextFieldDemo extends Applet implements ActionListener
{
	TextField name, pass;

	public void init()
	{
		Label namep = new Label("Name: ", Label.LEFT);
		Label passp = new Label("Password: ", Label.LEFT);
		name = new TextField(12);
		pass = new TextField(8);
		pass.setEchoChar('?');
		add(namep);
		add(name);
		add(passp);
		add(pass);
		name.addActionListener(this);
		pass.addActionListener(this);
	}

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

	public void paint(Graphics g) {
		g.drawString("Name: " + name.getText(), 6, 70);
		g.drawString("Password: " + pass.getText(), 6, 110);
	}
}
```

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( )
TextArea(int numLines, int numChars)
TextArea(String str)
TextArea(String str, int numLines, int numChars)
TextArea(String str, int numLines, int numChars, int sBars)
```

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

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

```
import java.applet.Applet;
import java.awt.TextArea;

/*
 <applet code="TextAreaDemo" width=300 height=250>
 </applet>
 */
public class TextAreaDemo extends Applet {
	public void init() {
		String val = "Java 7 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);
	}
}
```

Chapter III: Basics of Swing

3.0 Introduction

Swing is a set of classes that provides more powerful and flexible GUI components than does the AWT. Swing provides the look and feel of the modern Java GUI.

It is important to understand that the number of classes and interfaces in the Swing packages is quite large, and they can't all be covered in this chapter. (In fact, full coverage of Swing requires an entire book of its own.)

***Note**: although Swing eliminates a number of the limitations inherent in the AWT, Swing does not replace it. Instead, Swing is built on the foundation of the AWT. This is why the AWT is still a crucial part of Java.*

Swing also uses the same event handling mechanism as the AWT. Therefore, a basic understanding of the AWT and of event handling is required to use Swing.

3.1 Components and Containers

A Swing GUI consists of two key items: *components* and *containers*. The difference between the two is found in their intended purpose: As the term is commonly used, a *component* is an independent visual control, such as a push button or slider. A container holds a group of components. Thus, a container is a special type of component that is designed to hold other components. Furthermore, in order for a component to be displayed, it must be held within a container. Thus, all Swing GUIs will have at least one container. Because containers are components, a container can also hold other containers. This enables Swing to define what is called a *containment hierarchy*, at the top of which must be a *top-level container*.

The following list shows the class names for Swing components (including those used as containers).

JApplet	JButton	JCheckBox	JCheckBoxMenuItem
JColorChooser	JComboBox	JComponent	JDesktopPane
JDialog	JEditorPane	JFileChooser	JFormattedTextField

JFrame	JInternalFrame	JLabel	JLayeredPane
JList	JMenu	JMenuBar	JMenuItem
JOptionPane	JPanel	JPasswordField	JPopupMenu
JProgressBar	JRadioButton	JRadioButtonMenuItem	JRootPane
JScrollBar	JScrollPane	JSeparator	Jslider
JSpinner	JSplitPane	JTabbedPane	Jtable
JTextArea	JTextField	JTextPane	Jtogglebutton
JToolBar	JToolTip	JTree	Jviewport
JWindow			

3.2 A Simple Swing Application

Swing programs differ from both the console-based programs and the AWT-based programs shown earlier in this syllabus. For example, they use a different set of components and a different container hierarchy than does the AWT.

There are two types of Java programs in which Swing is typically used. The first is a desktop application. The second is the applet.

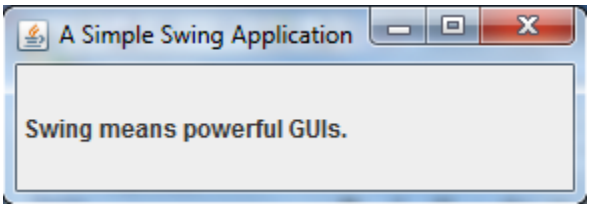
Although quite short, the following program shows one way to write a Swing application. It uses two Swing components: **JFrame** and **JLabel**.

JFrame is the top-level container that is commonly used for Swing applications. **JLabel** is the Swing component that creates a label, which is a component that displays information.

```
// A simple Swing application.
import javax.swing.*;

class SwingDemo {
      SwingDemo() {
            // Create a new JFrame container.
            JFrame jfrm = new JFrame("A Simple Swing Application");
            // Give the frame an initial size.
            jfrm.setSize(275, 100);
            // Terminate the program when the user closes the application.
            jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            // Create a text-based label.
            JLabel jlab=new JLabel(" Swing means powerful GUIs.");
            // Add the label to the content pane.
            jfrm.add(jlab);
            //Display the Frame
            jfrm.setVisible(true);
      }

}
```

```
public class SwingDemoTest {
	public static void main (String args[]){
		SwingDemo sd=new SwingDemo();
	}

}
```

Output:



3.3 Event Handling

The event handling mechanism used by Swing is the same as that used by the AWT. In many cases, Swing uses the same events as does the AWT, and these events are packaged in **java.awt.event**. Events specific to Swing are stored in **javax.swing.event**.

The following program handles the event generated by a Swing push button.

```
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
class EventDemo {
	JLabel jlab;
	EventDemo(){
		// Create a new JFrame container.
		JFrame jfrm = new JFrame("An Event Example");
		// Specify FlowLayout for the layout manager.
		jfrm.setLayout(new FlowLayout());
		// Give the frame an initial size.
		jfrm.setSize(220, 90);
		// Terminate the program when the user closes the application.
		jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		// Make two buttons.
		JButton jbtnAlpha, jbtnBeta;
		jbtnAlpha=new JButton("Alpha");
		jbtnBeta=new JButton("Beta");
		// Add action listener for Alpha
		jbtnAlpha.addActionListener(new ActionListener() {
			@Override
			public void actionPerformed(ActionEvent arg0) {
```

```
                        // TODO Auto-generated method stub
                        jlab.setText("Alpha was pressed.");
                  }
            });
            // Add action listener for Beta
            jbtnBeta.addActionListener(new ActionListener() {

                  @Override
                  public void actionPerformed(ActionEvent arg0) {
                        // TODO Auto-generated method stub
                        jlab.setText("Beta was pressed");
                  }
            });
            // Add the buttons to the content pane.
            jfrm.add(jbtnAlpha);
            jfrm.add(jbtnBeta);
            // Create a text-based label.
            jlab = new JLabel("Press a button.");
            // Add the label to the content pane.
            jfrm.add(jlab);
            // Display the frame.
            jfrm.setVisible(true);
      }

}

import javax.swing.SwingUtilities;

public class EventDemoTest {

      public static void main(String[] args) {
            // Create the frame on the event dispatching thread.
            SwingUtilities.invokeLater(new Runnable() {

                  @Override
                  public void run() {
                        // TODO Auto-generated method stub
                        new EventDemo();
                  }
            });
      }
}
```

In this program, notice that the program now imports both the **java.awt** and **java.awt.event** packages. The **EventDemo** constructor begins by creating a **JFrame** called **jfrm**. It then sets the layout manager for the content pane of **jfrm** to **FlowLayout**. When a push button is pressed, it generates an **ActionEvent**. Thus, **JButton** provides the **addActionListener( )** method, which is used to add an action listener. (**JButton** also provides **removeActionListener( )** to remove a listener, but this method is not used by this program.)

As explained in Chapter I, the **ActionListener** interface defines only one method: **actionPerformed( )**. It is shown again here for your convenience:

```
void actionPerformed(ActionEvent ae)
```

This method is called when a button is pressed.

One last point: Remember that all event handlers, such as **actionPerformed( )**, are called on the event dispatching thread. Therefore, an event handler must return quickly in order to avoid slowing down the application. If your application needs to do something time consuming as the result of an event, it must use a separate thread.

3.4 Create a Swing Applet

The second type of program that commonly uses Swing is the applet. Swing-based applets are similar to AWT-based applets, but with an important difference: A Swing applet extends **JApplet** rather than **Applet**. **JApplet** is derived from **Applet**. Thus, **JApplet** includes all of the functionality found in **Applet** and adds support for Swing. **JApplet** is a top-level Swing container, which means that it is *not* derived from **JComponent**. Because **JApplet** is a top-level container, it includes the various panes described earlier. This means that all components are added to **JApplet**'s content pane in the same way that components are added to **JFrame**'s content pane.

Swing applets use the same four lifecycle methods as described in Chapter I: **init( )**, **start( )**, **stop( )**, and **destroy( )**. Of course, you need override only those methods that are needed by your applet. Painting is accomplished differently in Swing than it is in the AWT, and a Swing applet will not normally override the paint( ) method.

Here is an example of a Swing applet. It provides the same functionality as the previous application, but does so in applet form.

```
// A simple Swing-based applet
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JApplet;
import javax.swing.JButton;
```

```
import javax.swing.JLabel;

public class MySwingApplet extends JApplet {
	JButton jbtnAlpha, jbtnBeta;
	JLabel jlab;

	public void init() {
		makeGUI();
	}

	// Set up and initialize the GUI.
	private void makeGUI() {
		// Set the applet to use flow layout.
		setLayout(new FlowLayout());
		// Make two buttons.
		jbtnAlpha = new JButton("Alpha");
		jbtnBeta = new JButton("Beta");
		// Add action listener for Alpha.
		jbtnAlpha.addActionListener(new ActionListener() {
			@Override
			public void actionPerformed(ActionEvent ae) {
				jlab.setText("Alpha was pressed.");
			}
		});
		// Add action listener for Beta.
		jbtnBeta.addActionListener(new ActionListener() {

			@Override
			public void actionPerformed(ActionEvent arg0) {
				jlab.setText("Beta was pressed.");
			}
		});
		// Add the buttons to the content pane.
		add(jbtnAlpha);
		add(jbtnBeta);
		// Create a text-based label.
		jlab = new JLabel("Press a button.");
		// Add the label to the content pane.
		add(jlab);
	}

}
```

3.5 Swing component classes

The Swing components provide rich functionality and allow a high level of customization. Because of space limitations, it is not possible to describe all of their features and attributes.

3.5.1 JLabel and ImageIcon

JLabel can be used to display text and/or an icon. It is a passive component in that it does not respond to user input. **JLabel** defines several constructors. Here are three of them:

```
JLabel(Icon icon)
JLabel(String str)
JLabel(String str, Icon icon, int align)
```

The easiest way to obtain an icon is to use the **ImageIcon** class. **ImageIcon** implements **Icon** and encapsulates an image. Thus, an object of type **ImageIcon** can be passed as an argument to the **Icon** parameter of **JLabel**'s constructor. There are several ways to provide the image, including reading it from a file or downloading it from a URL. Here is the **ImageIcon** constructor used by the example in this section:

```
ImageIcon(String filename)
```

It obtains the image in the file named *filename.*

The icon and text associated with the label can be obtained by the following methods:

```
Icon getIcon( )
String getText( )
```

The icon and text associated with a label can be set by these methods:

void setIcon(Icon *icon*)

void setText(String *str*)

The following applet illustrates how to create and display a label containing both an icon and a string. It begins by creating an **ImageIcon** object for the file **france.gif**, which depicts the flag for France. This is used as the second argument to the **JLabel** constructor.

The first and last arguments for the **JLabel** constructor are the label text and the alignment. Finally, the label is added to the content pane.

```
// Demonstrate JLabel and ImageIcon.
import javax.swing.ImageIcon;
import javax.swing.JApplet;
import javax.swing.JLabel;
import javax.swing.SwingUtilities;
/*
<applet code="JLabelDemo" width=800 height=300>
</applet>
*/
public class JLabelDemo extends JApplet {
      public void init() {
            try {
                  SwingUtilities.invokeAndWait(new Runnable() {
```

```
                        public void run() {
                              makeGUI();
                        }
                  });
            } catch (Exception exc) {
                  System.out.println("Can't create because of " + exc);
            }
      }

      void makeGUI() {
            // Create an icon.
            ImageIcon ii = new ImageIcon("france.gif");
            // create a label
            JLabel jl = new JLabel("France country", ii, JLabel.CENTER);
            // add label to the content pane.
            add(jl);
      }

}
```

3.5.2 JTextField

JTextField is the simplest Swing text component. It is also probably its most widely used text component. **JTextField** allows you to edit one line of text. It is derived from **JTextComponent**, which provides the basic functionality common to Swing text components. Three of **JTextField**'s constructors are shown here:

```
JTextField(int cols)
JTextField(String str, int cols)
JTextField(String str)
```

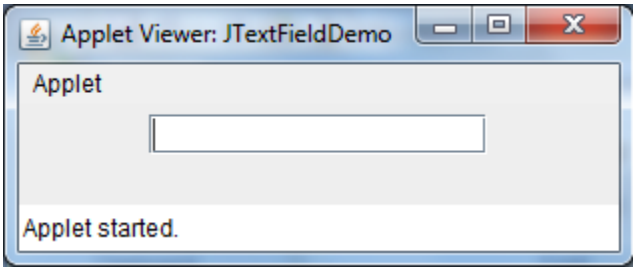
JTextField generates events in response to user interaction. For example, an **ActionEvent** is fired when the user presses ENTER. A **CaretEvent** is fired each time the caret (i.e., the cursor) changes position. (**CaretEvent** is packaged in **javax.swing.event**.) Other events are also possible. In many cases, your program will not need to handle these events. Instead, you will simply obtain the string currently in the text field when it is needed. To obtain the text currently in the text field, call **getText( )**.

The following example illustrates **JTextField**. It creates a **JTextField** and adds it to the content pane. When the user presses ENTER, an action event is generated. This is handled by displaying the text in the status window.

```
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JApplet;
import javax.swing.JTextField;
import javax.swing.SwingUtilities;

/*
<applet code="JTextFieldDemo" width=300 height=50>
</applet>
*/
public class JTextFieldDemo extends JApplet {
    JTextField jtf;
    public void init(){
        try{
            SwingUtilities.invokeAndWait(new Runnable() {

                @Override
                public void run() {
                    makeGUI();
                }
            });
        }catch(Exception exc){
            System.out.println("Can't create because of " + exc);
        }
    }
    private void makeGUI(){
        // Change to flow layout.
setLayout(new FlowLayout());
// Add text field to content pane.
jtf = new JTextField(15);
add(jtf);
jtf.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
               // Show text when user presses ENTER.
showStatus(jtf.getText());
            }
```

```
        });
    }
}
```

Output from the text field example is shown here:



3.5.3 Buttons

Swing defines four types of buttons: **JButton**, **JToggleButton**, **JCheckBox**, and **JRadioButton**. All are subclasses of the **AbstractButton** class, which extends **JComponent**. Thus, all buttons share a set of common traits.

JButton

The **JButton** class provides the functionality of a push button. Three of its constructors are shown here:

```
JButton(Icon icon)
JButton(String str)
JButton(String str, Icon icon)
```

When the button is pressed, an **ActionEvent** is generated. Using the **ActionEvent** object passed to the **actionPerformed( )** method of the registered **ActionListener**, you can obtain the *action command* string associated with the button. By default, this is the string displayed inside the button. However, you can set the action command by calling **setActionCommand( )** on the button. You can obtain the action command by calling **getActionCommand( )** on the event object. It is declared like this:

```
String getActionCommand( )
```

The following program displays four push buttons and a label. Each button displays an icon that represents the flag of a country. When a button is pressed, the name of that country is displayed in the label.

```
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.ImageIcon;
import javax.swing.JApplet;
import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.SwingUtilities;
/*
<applet code="JButtonDemo" width=250 height=450>
</applet>
 */

public class JButtonDemo extends JApplet implements ActionListener {

    JLabel jlab;

    public void init() {
        try {
            SwingUtilities.invokeAndWait(new Runnable() {

                @Override
                public void run() {
                    makeGUI();
                }
            });
        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
        }
    }

    private void makeGUI() {
        // Change to flow layout.
        setLayout(new FlowLayout());
        // Add buttons to content pane.
        ImageIcon france = new ImageIcon("france.gif");
        JButton jb = new JButton(france);
        jb.setActionCommand("France");
        jb.addActionListener(this);
        add(jb);
        ImageIcon germany = new ImageIcon("germany.gif");
        jb = new JButton(germany);
        jb.setActionCommand("Germany");
        jb.addActionListener(this);
        add(jb);
        ImageIcon italy = new ImageIcon("italy.gif");
        jb = new JButton(italy);
        jb.setActionCommand("Italy");
        jb.addActionListener(this);
    add(jb);

        ImageIcon japan = new ImageIcon("japan.gif");
```

```
        jb = new JButton(japan);
        jb.setActionCommand("Japan");
        jb.addActionListener(this);
        add(jb);
        // Create and add the label to content pane.
        jlab = new JLabel("Choose a Flag");
        add(jlab);
    }
    @Override
    public void actionPerformed(ActionEvent ae) {
        jlab.setText("You selected " + ae.getActionCommand());
      }
}
```

Output:

Check Boxes

The **JCheckBox** class provides the functionality of a check box. **JCheckBox** defines several constructors. The one used here is

```
JCheckBox(String str)
```

The following example illustrates check boxes. It displays four check boxes and a label. When the user clicks a check box, an **ItemEvent** is generated. Inside the **itemStateChanged( )** method, **getItem( )** is called to obtain a reference to the **JCheckBox** object that generated the event. Next,

a call to **isSelected( )** determines if the box was selected or cleared. The **getText( )** method gets the text for that check box and uses it to set the text inside the label.

```
import java.awt.FlowLayout;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;

import javax.swing.JApplet;
import javax.swing.JCheckBox;
import javax.swing.JLabel;
import javax.swing.SwingUtilities;
/*
<applet code="JCheckBoxDemo" width=270 height=50>
</applet>
*/
public class JCheckBoxDemo extends JApplet implements ItemListener {
	JLabel jlab;
	public void init(){
		try{
			SwingUtilities.invokeAndWait(new Runnable() {

				@Override
				public void run() {
					makeGUI();
				}
			});
		}catch (Exception exc) {
			System.out.println("Can't create because of " + exc);
		}
	}
	private void makeGUI(){
		//Change to flow layout
		setLayout(new FlowLayout());
		// Add check boxes to the content pane.
		JCheckBox cb=new JCheckBox("C");
		cb.addItemListener(this);
		add(cb);
		cb=new JCheckBox("C++");
		cb.addItemListener(this);
		add(cb);
		cb = new JCheckBox("Java");
		cb.addItemListener(this);
		add(cb);
		cb = new JCheckBox("Perl");
		cb.addItemListener(this);
		add(cb);
		// Create the label and add it to the content pane.
		jlab = new JLabel("Select languages");
		add(jlab);
	}
	// Handle item events for the check boxes.
	@Override
	public void itemStateChanged(ItemEvent ie) {
		JCheckBox cbx=(JCheckBox)ie.getItem();
		if (cbx.isSelected())
```

```
                jlab.setText(cbx.getText()+" is selected");
            else
                jlab.setText(cbx.getText()+" is cleared");
        }
}
```

Radio Buttons

Radio buttons are a group of mutually exclusive buttons, in which only one button can be selected at any one time. They are supported by the **JRadioButton** class. **JRadioButton** provides several constructors.The one used in the example is shown here:

```
JRadioButton(String str)
```

In order for their mutually exclusive nature to be activated, radio buttons must be configured into a group. Only one of the buttons in the group can be selected at any time. For example, if a user presses a radio button that is in a group, any previously selected button in that group is automatically deselected. Abutton group is created by the **ButtonGroup** class.

A**JRadioButton** generates action events, item events, and change events each time the button selection changes. Most often, it is the action event that is handled, which means that you will normally implement the **ActionListener** interface.

The following example illustrates how to use radio buttons. Three radio buttons are created. The buttons are then added to a button group. As explained, this is necessary to cause their mutually exclusive behavior. Pressing a radio button generates an action event, which is handled by **actionPerformed( )**.Within that handler, the **getActionCommand( )** method gets the text that is associated with the radio button and uses it to set the text within a label.

```
// Demonstrate JRadioButton
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.ButtonGroup;
import javax.swing.JApplet;
import javax.swing.JLabel;
import javax.swing.JRadioButton;
import javax.swing.SwingUtilities;
/*
<applet code="JRadioButtonDemo" width=300 height=50>
</applet>
*/
```

```
public class JRadioButtonDemo extends JApplet implements ActionListener {
	JLabel jlab;
	JRadioButton b1, b2, b3;
	public void init() {
		try{
			SwingUtilities.invokeAndWait(new Runnable() {

				@Override
				public void run() {
					makeGUI();
				}
			});
		}catch (Exception exc) {
			System.out.println("Can't create because of "+ exc);
		}
	}
	private void makeGUI(){
		//Change the flow layout
		setLayout(new FlowLayout());
		//create radio buttons
		b1=new JRadioButton("A");
		b2=new JRadioButton("B");
		b3=new JRadioButton("C");
		b1.addActionListener(this);
		b2.addActionListener(this);
		b3.addActionListener(this);
		add(b1);
		add(b2);
		add(b3);
		//define a button group
		ButtonGroup bg=new ButtonGroup();
		bg.add(b1);
		bg.add(b2);
		bg.add(b3);
		// Create a label and add it to the content pane.
		jlab = new JLabel("Select One");
		add(jlab);
	}
	// Handle button selection.
	@Override
	public void actionPerformed(ActionEvent ae) {
		jlab.setText("You selected: "+ae.getActionCommand());
	}

}
```

3.5.5 JTabbedPane

A tabbed pane is a componenet that appears as a group of folders in a file or cabinet. Each folder has a title. When a user selects a folder its contents become visible. Only one of the folders may be selected at a time. Tabbed panes are encapsulated by the JTabbedPane class. To add to the tab to the pane call, addTab().

```
import javax.swing.*;
```

```
public class JTabbedPaneDemo extends JApplet {
	public void init() {
		JTabbedPane jtp = new JTabbedPane();
		jtp.addTab("Cities", new CitiesPanel());
		jtp.addTab("Colors", new ColorsPanel());
		jtp.addTab("Flavors", new FlavorsPanel());
		getContentPane().add(jtp);
	}
}

class CitiesPanel extends JPanel {
	public CitiesPanel() {
		JButton b1 = new JButton("New York");
		add(b1);
		JButton b2 = new JButton("London");
		add(b2);
		JButton b3 = new JButton("Hong Kong");
		add(b3);
		JButton b4 = new JButton("Tokyo");
		add(b4);
	}
}

class ColorsPanel extends JPanel {
	public ColorsPanel() {
		JCheckBox cb1 = new JCheckBox("Red");
		add(cb1);
		JCheckBox cb2 = new JCheckBox("Green");
		add(cb2);
		JCheckBox cb3 = new JCheckBox("Blue");
		add(cb3);
	}
}

class FlavorsPanel extends JPanel {
	public FlavorsPanel() {
		JComboBox jcb = new JComboBox();
		jcb.addItem("Vanilla");
		jcb.addItem("Chocolate");
		jcb.addItem("Strawberry");
		add(jcb);
	}
}
```

3.5.5 JScrollPane

The scroll bars scroll the component through the viewport. In its default behavior, a **JScrollPane** will dynamically add or remove a scroll bar as needed. For example, if the component is taller than the viewport, a vertical scroll bar is added. If the component will completely fit within the viewport, the scroll bars are removed.

JScrollPane defines several constructors. The one used in this chapter is shown here:

```
JScrollPane(Component comp)
```

The component to be scrolled is specified by *comp*. Scroll bars are automatically displayed when the content of the pane exceeds the dimensions of the viewport.

Here are the steps to follow to use a scroll pane:

1. Create the component to be scrolled.
2. Create an instance of **JScrollPane**, passing to it the object to scroll.
3. Add the scroll pane to the content pane.

The following example illustrates a scroll pane. First, a **JPanel** object is created, and 400 buttons are added to it, arranged into 20 columns. This panel is then added to a scroll pane, and the scroll pane is added to the content pane. Because the panel is larger than the viewport, vertical and horizontal scroll bars appear automatically.

```
import java.awt.BorderLayout;
import java.awt.GridLayout;

import javax.swing.JApplet;
import javax.swing.JButton;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.SwingUtilities;
/*
<applet code="JScrollPaneDemo" width=300 height=250>
</applet>
*/
public class JScrollPaneDemo extends JApplet{
	JPanel jp;
	public void init(){
		try{
			SwingUtilities.invokeAndWait(new Runnable() {

				@Override
				public void run() {
					makeGUI();
				}
			});
		}catch (Exception e) {
			System.out.println("Can't create because of " + e);
		}
	}
	private void makeGUI(){
		//Add 400 Buttons to a panel.
		jp=new JPanel();
		jp.setLayout(new GridLayout(20,20));
```

```
            for(int i=0; i<20; i++)
                  for(int j=0;j<20;j++)
                        jp.add(new JButton("Button("+i+", "+j+")"));
            // Create the scroll pane.
            JScrollPane jsp=new JScrollPane(jp);
            // Add the scroll pane to the content pane.
            add(jsp, BorderLayout.CENTER);
      }
}
```

3.5.6 JComboBox

Swing provides a *combo box* (a combination of a text field and a drop-down list) through the **JComboBox** class. A combo box normally displays one entry, but it will also display a drop-down list that allows a user to select a different entry. You can also create a combo box that lets the user enter a selection into the text field. The **JComboBox** constructor used by the example is shown here:

```
JComboBox(Object[ ] items)
```

JComboBox generates an action event when the user selects an item from the list. **JComboBox** also generates an item event when the state of selection changes, which occurs when an item is selected or deselected.

The following example demonstrates the combo box. The combo box contains entriesfor "France," "Germany," "Italy," and "Japan." When a country is selected, an icon-based label is updated to display the flag for that country. You can see how little code is required to use this powerful component.

```
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.ImageIcon;
import javax.swing.JApplet;
import javax.swing.JComboBox;
import javax.swing.JLabel;
import javax.swing.SwingUtilities;

/*
 <applet code="JComboBoxDemo" width=300 height=100>
 </applet>
 */
public class JComboBoxDemo extends JApplet {
      JLabel jlab;
      ImageIcon france, germany, italy, japan;
```

```
    JComboBox jcb;
    String flags[] = { "France", "Germany", "Italy", "Japan" };

    public void init() {
        try {
            SwingUtilities.invokeAndWait(new Runnable() {
                public void run() {
                    makeGUI();
                }
            });
        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
        }
    }

    private void makeGUI() {
        // Change to flow layout.
        setLayout(new FlowLayout());
        // Instantiate a combo box and add it to the content pane.
        jcb = new JComboBox(flags);
        add(jcb);
        // Handle selections.
        jcb.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ae) {
                String s = (String) jcb.getSelectedItem();
                jlab.setIcon(new ImageIcon(s + ".gif"));
            }
        });
        // Create a label and add it to the content pane.
        jlab = new JLabel(new ImageIcon("france.gif"));
        add(jlab);
    }

}
```

3.5.7 Trees

A*tree* is a component that presents a hierarchical view of data. Trees are implemented in Swing by the **JTree** class. A sampling of its constructors is shown here:

```
JTree(Object obj[ ])
JTree(Vector<?> v)
JTree(TreeNode tn)
```

In the first form, the tree is constructed from the elements in the array *obj*. The second form constructs the tree from the elements of vector *v*. In the third form, the tree whose root node is specified by *tn* specifies the tree.

Although **JTree** is packaged in **javax.swing**, its support classes and interfaces are packaged in **javax.swing.tree**. This is because the number of classes and interfaces needed to support **JTree** is quite large.

JTree relies on two models: **TreeModel** and **TreeSelectionModel**. A **JTree** generates a variety of events, but three relate specifically to trees: **TreeExpansionEvent**, **TreeSelectionEvent**, and **TreeModelEvent**. **TreeExpansionEvent** events occur when a node is expanded or collapsed. A **TreeSelectionEvent** is generated when the user selects or deselects a node within the tree. A**TreeModelEvent** is fired when the data or structure of the tree changes. The listeners for these events are **TreeExpansionListener**, **TreeSelectionListener**, and **TreeModelListener**, respectively. The tree event classes and listener interfaces are packaged in **javax.swing.event**.

JTree does not provide any scrolling capabilities of its own. Instead, a **JTree** is typically placed within a **JScrollPane**. This way, a large tree can be scrolled through a smaller viewport.

Here are the steps to follow to use a tree:

1. Create an instance of **JTree**.
2. Create a **JScrollPane** and specify the tree as the object to be scrolled.
3. Add the tree to the scroll pane.
4. Add the scroll pane to the content pane.

The following example illustrates how to create a tree and handle selections. The program creates a **DefaultMutableTreeNode** instance labeled "Options." This is the top node of the tree hierarchy. Additional tree nodes are then created, and the **add( )** method is called to connect these nodes to the tree. A reference to the top node in the tree is provided as the argument to the **JTree** constructor. The tree is then provided as the argument to the **JScrollPane** constructor. This scroll pane is then added to the content pane. Next, a label is created and added to the content pane. The tree selection is displayed in this label. To receive selection events from the tree, a **TreeSelectionListener** is registered for the tree.

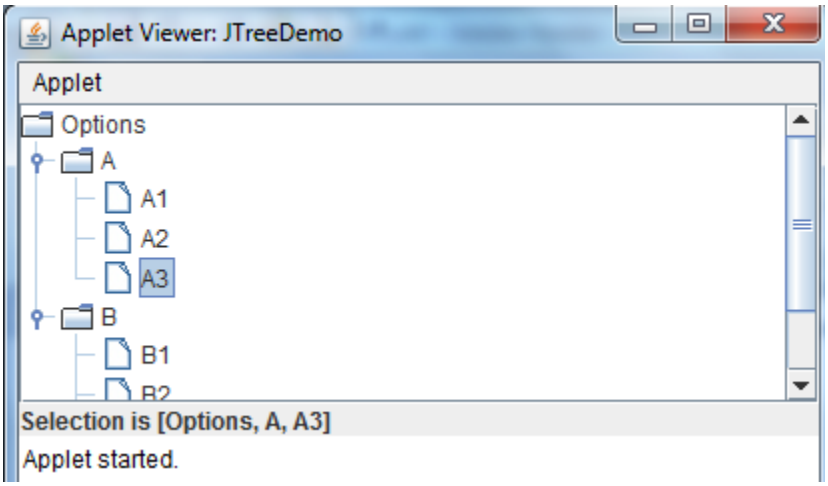
Inside the **valueChanged( )** method, the path to the current selection is obtained and displayed.

```
import java.awt.BorderLayout;

import javax.swing.JApplet;
import javax.swing.JLabel;
import javax.swing.JScrollPane;
import javax.swing.JTree;
import javax.swing.SwingUtilities;
import javax.swing.event.TreeSelectionEvent;
import javax.swing.event.TreeSelectionListener;
import javax.swing.tree.DefaultMutableTreeNode;
```

```
/*
 <applet code="JTreeDemo" width=400 height=200>
 </applet>
 */
public class JTreeDemo extends JApplet {
	JTree tree;
	JScrollPane jsp;
	JLabel lab;

	public void init() {
		try {
			SwingUtilities.invokeAndWait(new Runnable() {

				@Override
				public void run() {
					makeGUI();

				}
			});
		} catch (Exception e) {
			System.out.println("Can't create because of " + e);
		}
	}

	private void makeGUI() {
		// Create top node of tree.
		DefaultMutableTreeNode top = new
DefaultMutableTreeNode("Options");
		// Create subtree of "A".
		DefaultMutableTreeNode a, a1, a2, a3;
		a = new DefaultMutableTreeNode("A");
		a1 = new DefaultMutableTreeNode("A1");
		a2 = new DefaultMutableTreeNode("A2");
		a3 = new DefaultMutableTreeNode("A3");
		top.add(a);
		a.add(a1);
		a.add(a2);
		a.add(a3);
		//Create subtrees of B
		DefaultMutableTreeNode b,b1,b2,b3,b4;
		b=new DefaultMutableTreeNode("B");
		b1=new DefaultMutableTreeNode("B1");
		b2=new DefaultMutableTreeNode("B2");
		b3=new DefaultMutableTreeNode("B3");
		b4=new DefaultMutableTreeNode("B4");
		top.add(b);
		b.add(b1);
		b.add(b2);
		b.add(b3);
		b.add(b4);

		// Create the tree.
		tree = new JTree(top);
		// Add the tree to a scroll pane.
		jsp = new JScrollPane(tree);
		// Add the scroll pane to the content pane.
		add(jsp);
```

```
            // Add the label to the content pane.
            lab = new JLabel();
            add(lab, BorderLayout.SOUTH);
            // Handle tree selection events.
            tree.addTreeSelectionListener(new TreeSelectionListener() {

                  @Override
                  public void valueChanged(TreeSelectionEvent tse) {
                        lab.setText("Selection is " + tse.getPath());

                  }
            });
      }

}
```

Output from the tree example is shown here:



3.5.8 JTable

At its core, **JTable** is conceptually simple. It is a component that consists of one or more columns of information. At the top of each column is a heading. In addition to describing the data in a column, the heading also provides the mechanism by which the user can change the size of a column or change the location of a column within the table. **JTable** does not provide any scrolling capabilities of its own. Instead, you will normally wrap a **JTable** inside a **JScrollPane**. **JTable** supplies several constructors. The one used here is

```
JTable(Object data[ ][ ], Object colHeads[ ])
```

Here, *data* is a two-dimensional array of the information to be presented, and *colHeads* is a one-dimensional array with the column headings.

JTable relies on three models. The first is the table model, which is defined by the **TableModel** interface. This model defines those things related to displaying data in a two-dimensional format. The second is the table column model, which is represented by **TableColumnModel**. **JTable** is defined in terms of columns, and it is **TableColumnModel** that specifies the characteristics of a column. These two models are packaged in **javax.swing.table**.

The third model determines how items are selected, and it is specified by the **ListSelectionModel**.

Here are the steps required to set up a simple **JTable** that can be used to display data:

1. Create an instance of **JTable**.
2. Create a **JScrollPane** object, specifying the table as the object to scroll.
3. Add the table to the scroll pane.
4. Add the scroll pane to the content pane.

The following example illustrates how to create and use a simple table. A one-dimensional array of strings called **colHeads** is created for the column headings. Atwo-dimensional array of strings called **data** is created for the table cells. These arrays are passed to the **JTable** constructor. The table is added to a scroll pane, and then the scroll pane is added to the content pane. The table displays the data in the **data** array. The default table configuration also allows the contents of a cell to be edited. Changes affect the underlying array, which is **data** in this case.

```
import javax.swing.JApplet;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.SwingUtilities;
/*
<applet code="JTableDemo" width=400 height=200>
</applet>
*/
public class JTableDemo extends JApplet {
      JTable table;
      JScrollPane jsp;
      public void init() {
            try {
                  SwingUtilities.invokeAndWait(new Runnable() {

                        @Override
                        public void run() {
                              makeGUI();

                        }
                  });
```

```
            } catch (Exception exc) {
                  System.out.println("Can't create because of " + exc);
            }
      }

      private void makeGUI() {
            // Initialize column headings.
            String colHeads[]={"Student_ID", "First Name", "Last Name",
"Email_Address"};
            // Initialize data.
            Object data[][] = {
                        {"GS20110076", "Patrick", "CYUZUZO", "patrickcyuzuzo@gmail.com"},
                        {"GS20110079", "Lethitia", "DUSABEYEZU", "dusalae2020@yahoo.fr"},
                        {"GS20110078", "Thamar", "DUSABEYEZU", "dusath02@yahoo.com"},
                        {"GS20110083", "Jean de Dieu", "HABIMANA","habijedo@yahoo.fr"},
                        {"GS20110085", "Abdallahman", "HABYARIMANA","habdallah05@yahoo.fr"},
                        {"GS20110090", "Salathiel", "Kanani", "ksalath@gmail.com"},
                        {"GS20090039", "Jean Paul", "KAREMERA", "jipkare05@yahoo.fr"},
                        {"GS20110096", "J damascene", "MBITURIMANA", "madas01d@yahoo.fr"},
                        {"GS20110097", "Adrien", "MBONAMPEKA", "mbonadri@hotmail.com"},
                        {"GS20110101", "yves", "Mugisha", "filsmugisha@gmail.com"},
                        {"GS20110102", "Modeste", "MUHIRWA", "modehirwa@yahoo.fr"},
                        {"GS20090254", "Gaudelive", "MUKESHARUREMA", "gmukesharurema@yahoo.com"},
                        {"GS20110109", "Jean Claude", "MUTWARASIBO", "dclauso@yahoo.fr"},
                        {"GS20110113", "Jean damascene", "NDAMWIRINGIYE", "njean_damas@yahoo.com"},
                        {"GS20110114", "Gloria", "NIYOMUFASHA", "glospino@yahoo.fr"},
                        {"GS20110119", "J.CLAUDE", "NSABIMANA", "ngogajeanclaude@yahoo.com"},
                        {"GS20110120", "Valens", "NSANZIMFURA", "valens.nsanzimfura@yahoo.fr"},
                        {"GS20110126", "Jeannette", "NYAMPINGA", "jeadelle@yahoo.fr"},
                        {"GS20110127", "Pascaline", "NYIRAMINANI", "pascaline120@gmail.com"},
                        {"GS20110101", "Eric", "RWAMASUNZU", "rickwert1@yahoo.fr"},
                        {"GS20110132", "Elvis", "SHEMA", "elrnpsm@gmail.com"},
                        {"GS20110135", "JOSEPHINE", "TUJYIMBERE", "Josekelyne@gmail.com"},
                        {"PS20110081", "Eric", "UFITINEMA", "ufiteric05@yahoo.fr"},
                              };
            //Create the table
            table=new JTable(data,colHeads);
            //Add the table to a scroll pane
            jsp=new JScrollPane(table);
            //Add Scroll pane to the Content pane
            add(jsp);
      }
}
```

Chapter IV: Java Database Connectivity

4.0 Introduction

A **database** is an organized collection of data. There are many different strategies for organizing data to facilitate easy access and manipulation. A **database management system** (**DBMS**) provides mechanisms for storing, organizing, retrieving and modifying data for many users. Database management systems allow for the access and storage of data without concern for the internal representation of data.

Today's most popular database management systems are **relational database systems**. SQL is the international standard language used to query and manipulate relational data.

Programs connect to, and interact with, relational databases via an interface—software that facilitates communications between a database management system and a program. A **JDBC driver** enables Java applications to connect to a database in a particular DBMS and allows you to retrieve and manipulate database data.

Note:

- *Using the JDBC API enables developers to change the underlying DBMS (for example, from Java DB to MySQL, POSTGRESQL, ...) without modifying the Java code that accesses the database.*
- *Most major database vendors provide their own JDBC database drivers, and many thirdparty vendors provide JDBC drivers as well.*

4.0.1 Relational Databases

A relational database stores data in tables. Tables are composed of rows, and rows are composed of columns in which values are stored.

A table's primary key provides a unique value that cannot be duplicated among rows. Each column of a table represents a different attribute. The primary key can be composed of more than one column. Every column in a primary key must have a value, and the value of the primary key must be unique. This is known as the **Rule of Entity Integrity**.

A one-to-many relationship between tables indicates that a row in one table can have many related rows in a separate tables.

A foreign key is a column in a table that must match the primary-key column in another table. This is known as the **Rule of Referential Integrity**.

Foreign keys enable information from multiple tables to be joined together. There's a one-tomany relationship between a primary key and its corresponding foreign keys.

4.0.2 Example: the books database

The database consists of three tables: Authors, AuthorISBN and Titles.

The Authors table consists of three columns that maintain each author's unique ID number, first name and last name.

Column	Description
`AuthorID`	Author's ID number in the database. In the books database, this integer column is defined as autoincremented—for each row inserted in this table, the `AuthorID` value is increased by 1 automatically to ensure that each row has a unique `AuthorID`. This column represents the table's primary key.
`FirstName`	Author's first name (a string).
`LastName`	Author's last name (a string).

Figure 2: Authors table from the books database

AuthorID	FirstName	LastName
1	Paul	Deitel
2	Harvey	Deitel
3	Abbey	Deitel
4	Michael	Morgano
5	Eric	Kern

Figure 3: Sample data from the Authors table.

The AuthorISBN table consists of two columns that maintain each ISBN and the corresponding author's ID number. This table associates authors with their books. Both columns are foreign keys that represent the relationship between the tables Authors and Titles—one row in table Authors may be associated with many rows in table Titles, and vice versa. The combined columns of the AuthorISBN table represent the table's *primary key*—thus, each row in this table must be a *unique* combination of an AuthorID and an ISBN.

Column	Description
`AuthorID`	The author's ID number, a foreign key to the `Authors` table.
`ISBN`	The ISBN for a book, a foreign key to the `Titles` table.

Figure 4: AuthorISBN table from the books database.

AuthorID	ISBN	AuthorID	ISBN
1	013705842X	1	0132121360
2	013705842X	2	0132121360
3	013705842X	3	0132121360
4	013705842X	4	0132121360
5	013705842X		

Figure 5: Sample data from the AuthorISBN table of books.

The Titles table described consists of four columns that stand for the ISBN, the title, the edition number and the copyright year.

Column	Description
`ISBN`	ISBN of the book (a string). The table's primary key. ISBN is an abbreviation for "International Standard Book Number"—a numbering scheme that publishers use to give every book a unique identification number.
`Title`	Title of the book (a string).
`EditionNumber`	Edition number of the book (an integer).
`Copyright`	Copyright year of the book (a string).

Figure 6: Titles table from the books database.

ISBN	Title	EditionNumber	Copyright
0132152134	Visual Basic 2010 How to Program	5	2011
0132151421	Visual C# 2010 How to Program	4	2011
0132575663	Java How to Program	9	2012
0132662361	C++ How to Program	8	2012
0132404168	C How to Program	6	2010
013705842X	iPhone for Programmers: An App-Driven Approach	1	2010
0132121360	Android for Programmers: An App-Driven Approach	1	2012

Figure 7: Sample data from the Titles table of the books database

The diagram below shows the *database tables*and the *relationships* among them. The first compartment in each box contains the table's name and the remaining compartments contain the table's columns. The names in italic are primary keys. *A table's primary key uniquely identifies each row in the table.*

Every *row* must have a primary-key value, and that value must be unique in the table. This is known as the **Rule of Entity Integrity**. Again, for the AuthorISBN table, the primary key is the combination of both columns.

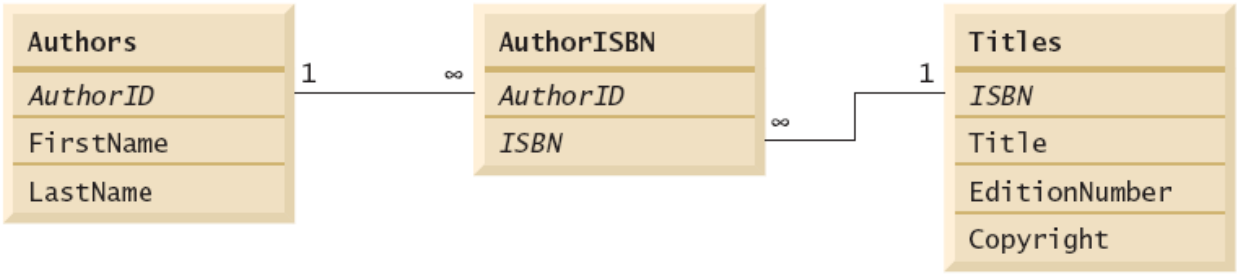


Figure 8: Table relationships in the books database.

Note:

- *Not providing a value for every column in a primary key breaks the Rule of Entity Integrity and causes the DBMS to report an error.*
- *Providing the same primary-key value in multiple rows causes the DBMS to report an error.*

The lines connecting the tables represent the relationships between them. Consider the line between the AuthorISBN and Authors tables.On the Authors end of the line is a 1, and on the AuthorISBN end is an infinity symbol (∞), indicating a **oneto-many relationship** in which every author in the Authors table can have an arbitrary number of books in the AuthorISBN table. The relationship line links the AuthorID column in Authors (i.e., its primary key) to the AuthorID column in AuthorISBN (i.e., its foreign key). The AuthorID column in the AuthorISBN table is a foreign key.

Note: *Providing a foreign-key value that does not appear as a primary-key value in another table breaks the Rule of Referential Integrity and causes the DBMS to report an error.*

The line between Titles and AuthorISBN illustrates another *one-to-many relationship*; a title can be written by any number of authors. In fact, the sole purpose of the AuthorISBN table is to provide a *many-to-many relationship* between Authors and Titles—an author can write many books and a book can have many authors.

4.0.3 SQL

SQL keyword	Description
SELECT	Retrieves data from one or more tables.
FROM	Tables involved in the query. Required in every SELECT.
WHERE	Criteria for selection that determine the rows to be retrieved, deleted or updated. Optional in a SQL query or a SQL statement.
GROUP BY	Criteria for grouping rows. Optional in a SELECT query.
ORDER BY	Criteria for ordering rows. Optional in a SELECT query.
INNER JOIN	Merge rows from multiple tables.
INSERT	Insert rows into a specified table.
UPDATE	Update rows in a specified table.
DELETE	Delete rows from a specified table.

Figure 9: SQL query keywords.

Basic SELECT Query

The basic form of a query is:

```
SELECT * FROM tableName
```

where the asterisk (*) indicates that all columns from *tableName* should be selected, and *tableName* specifies the table in the database from which rows will be retrieved. For example, to retrieve all the data in the Authors table, use:

```
SELECT * FROM Authors
```

To retrieve specific columns, replace the * with a comma-separated list of column names. For example, to retrieve only the columns AuthorID and LastName for all rows in the Authors table, use the query:

```
SELECT AuthorID, LastName FROM Authors
```

This query returns the data listed in the following figure:

AuthorID	LastName
1	Deitel
2	Deitel
3	Deitel
4	Morgano
5	Kern

Figure 10: Sample AuthorID and

WHERE Clause

The optional WHERE clause in a query specifies the selection criteria for the query. The basic form of a query with selection criteria is SELECT *columnName1*, *columnName2*, ... FROM *tableName* WHERE *criteri.*

The WHERE clause can contain operators <, >, <=, >=, =, <> and LIKE. LIKE is used for string pattern matching with wildcard characters percent (%) and underscore (_). A percent character in a pattern indicates that a string matching the pattern can have zero or more characters at the percent character's location in the pattern. An underscore in the pattern string indicates a single character at that position in the pattern.

For example, to select the Title, EditionNumber and Copyright columns from table Titles for which the Copyright date is greater than 2010, use the query

```
SELECT Title, EditionNumber, Copyright FROM Titles WHERE Copyright > '2010'
```

Strings in SQL are delimited by single (') rather than double (").

ORDER BY Clause

A query's result can be sorted with the ORDER BY clause. The simplest form of an ORDER BY clause is:

```
SELECT columnName1, columnName2, … FROM tableName ORDER BY column ASC
SELECT columnName1, columnName2, … FROM tableName ORDER BY column DESC
```

where ASC specifies ascending order, DESC specifies descending order and *column* specifies the column on which the sort is based. The default sorting order is ascending, so ASC is optional.

Multiple columns can be used for ordering purposes with an ORDER BY clause of the form ORDER BY *column1 sortingOrder*, *column2 sortingOrder*, *…*

The WHERE and ORDER BY clauses can be combined in one query. If used, ORDER BY must be the last clause in the query.

Merging Data from Multiple Tables: INNER JOIN

An INNER JOIN merges rows from two tables by matching values in columns that are common to the tables. The basic form for the INNER JOIN operator is:

```
SELECT columnName1, columnName2, … FROM table1 INNER JOIN table2 ON table1.columnName = table2.columnName
```

The ON clause specifies the columns from each table that are compared to determine which rows are joined. If a SQL statement uses columns with the same name from multiple tables, the column names must be fully qualified by prefixing them with their table names and a dot (.).

INSERT Statement

An INSERT statement inserts a new row into a table. The basic form of this statement is

`INSERT INTO tableName ( columnName1, columnName2, …, columnNameN ) VALUES ( value1, value2, …, valueN )`

where *tableName* is the table in which to insert the row. The *tableName* is followed by a comma-separated list of column names in parentheses. The list of column names is followed by the SQL keyword VALUES and a comma-separated list of values in parentheses.

UPDATE Statement

An UPDATE statement modifies data in a table. The basic form of an UPDATE statement is

`UPDATE tableName SET columnName1 = value1, columnName2 = value2, …, columnNameN = valueN WHERE criteria`

where *tableName* is the table to update. Keyword SET is followed by a comma-separated list of *columnName* = *value* pairs. The optional WHERE clause determines which rows to update.

DELETE Statement

A DELETE statement removes rows from a table. The simplest form for a DELETE statement

Is: `DELETE FROM tableName WHERE criteria`

where *tableName* is the table from which to delete a row (or rows). The optional WHERE *criteria* determines which rows to delete. If this clause is omitted, all the table's rows are deleted.

4.1 Java API & JDBC

API, an abbreviation of ***a****pplication* ***p****rogram* ***i****nterface*, is a set of routines, protocols, and tools for building software applications. A good API makes it easier to develop a program by providing all the building blocks. A programmer then puts the blocks together.

JDBC stands for **J**ava **D**ata**b**ase **C**onnectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.

The JDBC library includes APIs for each of the tasks commonly associated with database usage:

- Making a connection to a database
- Creating SQL statements
- Executing those SQL queries in the database
- Viewing & Modifying the resulting records

Fundamentally, JDBC is a specification that provides a complete set of interfaces that allows for portable access to an underlying database. Java can be used to write different types of executables, such as:

- Java Applications
- Java Applets
- Java Servlets
- Java ServerPages (JSPs)
- Enterprise JavaBeans (EJBs)
- Mobile Applications
- ...

All of these different executables are able to use a JDBC driver to access a database and take advantage of the stored data.

Two-Tier Database Access Models

When we use a two-tier database access model, our Java application accesses the database directly through the JDBC API. The results from the database are directly sent back to the application.

Three-Tier Database Access Models

The three-tier database access model is a bit complicated, but very secure and robust. In this model the JDBC driver sends commands to a middle tier, which in turn sends commands to the database. The results of these commands are then sent back to the middle tier, which communicates them back to the application.

4.2 JDBC Driver Types

There are four basic types of JDBC driver, they are as follows:

JDBC-ODBC Bridge Driver: These type of drivers use a bridge technology to connect a Java client to an ODBC database service.

Native-API Partly-Java Driver: These type of drivers wrap a thin layer of Java around database specific native code. Since these are implemented using native code, they may have better performance, but however there is a risk, because a defect in a driver's native codes section can crash the entire server.

Net-Protocol All-Java Driver: They communicate via a generic network protocol to a piece of custom middleware. The middleware component might use any type of driver to provide the actual database access. These drivers are all Java, which makes them useful for applet deployment and safe for servlet deployment.

Native-Protocol All-Java Driver: This type of driver is written in Java. These drivers understand database specific networking protocols and can access the database directly without any additional software.

4.3 Establishing a Database Connection

The first thing required is to establish a connection to the database. This is a two step process.

First the driver must be loaded and then the connection is established. Loading a JDBC driver is very simple. The following code fragment will make it more clear:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

When this method is called, it creates an instance of the driver and register it with the DriverManager.

Now the next step is to make a connection:

```
Connection con = DriverManager.getConnection("jdbc:odbc:DSNName","
UserName","Password")
```

returns a connection to a database.

The following code create a connection to a **PostgreSQL (**=DBMS like MySQL, MS Access or ORACLE**)** database. The database name is School.

```
import java.sql.Connection;
import java.sql.DriverManager;

public class Connect {
	public static void main(String args[]) {
		try {
			Class.forName("org.postgresql.Driver");
			System.out.println("Driver OK!");
			String url = "jdbc:postgresql://localhost:5432/School";
			String user = "user";
			String passwd = "postgres";
			Connection conn = DriverManager.getConnection(url, user,
passwd);
			System.out.println("Effective Connection!");
		} catch (Exception e) {
			e.printStackTrace();
		}
	}
}
```

Now compile above example.
The output should look like this:

```
Driver OK!
Effective Connection!
```

4.4 Searching in Database

The couple of object: **`Statement – ResultSet`** is frequently used.

Exapmle 1:

Here is a complete example that displays the content of table `class` of `School` database.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.Statement;

public class Connect2 {
	public static void main(String[] args) {
		try {
			Class.forName("org.postgresql.Driver");
			String url = "jdbc:postgresql://localhost:5432/School";
			String user = "user";
			String passwd = "password";
			Connection conn = DriverManager.getConnection(url, user,
passwd);
```

```
                    // Creation of the Statement object
                    Statement state = conn.createStatement();
                    // the object ResultSet contains the result of the SQL
request.
                    ResultSet result = state.executeQuery("SELECT * FROM
class");
                    // recuperation of MetaData
                    ResultSetMetaData resultMeta = result.getMetaData();
                    System.out.println("\n**********************************");
                    // Display the name of Fields
                    for (int i = 1; i <= resultMeta.getColumnCount(); i++)
                          System.out.print("\t"
                                      +
resultMeta.getColumnName(i).toUpperCase() + "\t *");
                    System.out.println("\n**********************************");
                    while (result.next()) {
                          for (int i = 1; i <= resultMeta.getColumnCount();
i++)
                                System.out.print("\t" +
result.getObject(i).toString()
                                            + "\t |");
                          System.out.println("\n-------------------------------
--");
                    }
                    result.close();
                    state.close();
              } catch (Exception e) {
                    e.printStackTrace();
              }
        }
}
```

The output:

```
**********************************
       CLS_ID   *      CLS_NOM   *
**********************************
       1        |      6 A       |
--------------------------------
       2        |      6 B       |
--------------------------------
       3        |      6 C       |
--------------------------------
       4        |      5 A       |
--------------------------------
       5        |      5 B       |
--------------------------------
       7        |      4 A       |
--------------------------------
       6        |      5 C       |
--------------------------------
       8        |      4 B       |
--------------------------------
       9        |      4 C       |
--------------------------------
       11       |      3 B       |
--------------------------------
       10       |      3 A       |
--------------------------------
       12       |      3 C       |
--------------------------------
```

If we don't consider the connection, the above program has four steps:

Step1: **`Statement`** object creation

Step2: SQL request execution

Step3: Recuperation and display of data via **`ResultSet`** object

Step4: Closing the used objects (Recommended even if not compulsory)

JDBC – Statements

Once a connection is obtained we can interact with the database. The JDBC *Statement interface* defines the methods and properties that enable you to send SQL commands and receive data from your database.

The Statement Objects

Before you can use a Statement object to execute a SQL statement, you need to create one using the Connection object's createStatement( ) method, as in the previous program in the statement:

```
Statement state = conn.createStatement();
```

Once you've created a Statement object, you can then use it to execute a SQL statement with one of its three execute methods.

1. **boolean execute(String SQL)** : Returns a boolean value of true if a ResultSet object can be retrieved; otherwise, it returns false. Use this method to execute SQL DDL statements or when you need to use truly dynamic SQL.
2. **int executeUpdate(String SQL)** : Returns the numbers of rows affected by the execution of the SQL statement. Use this method to execute SQL statements for which you expect to get a number of rows affected - for example, an INSERT, UPDATE, or DELETE statement.
3. **ResultSet executeQuery(String SQL)** : Returns a ResultSet object. Use this method when you expect to get a result set, as you would with a SELECT statement.

Closing Statement Object:

Just as you close a Connection objects to save database resources, for the same reason you should also close the Statement object.

A simple call to the close() method will do the job. If you close the Connection object first it will close the Statement object as well. However, you should always explicitly close the Statement object to ensure proper cleanup.

JDBC - Result Sets

The SQL statements that read data from a database query return the data in a result set. The SELECT statement is the standard way to select rows from a database and view them in a result set. The *java.sql.ResultSet* interface represents the result set of a database query.

A ResultSet object maintains a cursor that points to the current row in the result set. The term "result set" refers to the row and column data contained in a ResultSet object.

Example2:

The following example performs a simple query on the books database that retrieves the entire authors table and displays data.

```
// Displaying the contents of the authors table.
import java.sql.Connection;
import java.sql.Statement;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;

public class DisplayAuthors
{
   // JDBC driver name and database URL
   static final String DRIVER = "com.mysql.jdbc.Driver";
   static final String DATABASE_URL = "jdbc:mysql://localhost/books";

   // launch the application
   public static void main( String args[] )
   {
      Connection connection = null; // manages connection
      Statement statement = null; // query statement
      ResultSet resultSet = null; // manages results

      // connect to database books and query database
      try
      {
         // load the driver class
         Class.forName( DRIVER );

         // establish connection to database
         connection =
            DriverManager.getConnection( DATABASE_URL, "jhtp7", "jhtp7" );

         // create Statement for querying database
         statement = connection.createStatement();

         // query database
         resultSet = statement.executeQuery(
            "SELECT authorID, firstName, lastName FROM authors" );

         // process query results
         ResultSetMetaData metaData = resultSet.getMetaData();
```

```
            int numberOfColumns = metaData.getColumnCount();
            System.out.println( "Authors Table of Books Database:\n" );

            for ( int i = 1; i <= numberOfColumns; i++ )
               System.out.printf( "%-8s\t", metaData.getColumnName( i ) );
            System.out.println();

            while ( resultSet.next() )
            {
               for ( int i = 1; i <= numberOfColumns; i++ )
                  System.out.printf( "%-8s\t", resultSet.getObject( i ) );
               System.out.println();
            } // end while
         }  // end try
         catch ( SQLException sqlException )
         {
            sqlException.printStackTrace();
         } // end catch
         catch ( ClassNotFoundException classNotFound )
         {
            classNotFound.printStackTrace();
         } // end catch
         finally // ensure resultSet, statement and connection are closed
         {
            try
            {
               resultSet.close();
               statement.close();
               connection.close();
            } // end try
            catch ( Exception exception )
            {
               exception.printStackTrace();
            } // end catch
         } // end finally
      } // end main
} // end class DisplayAuthors
```

4.5 JDBC – Database examples

Create database

There are following steps required to create a new Database using JDBC application:

1. **Import the packages.** Requires that you include the packages containing the JDBC classes needed for database programming. Most often, using *import java.sql.** will suffice.
2. **Register the JDBC driver.** Requires that you initialize a driver so you can open a communications channel with the database.
3. **Open a connection.** Requires using the *DriverManager.getConnection()* method to create a Connection object, which represents a physical connection with datbase server.

To create a new database, you need not to give any database name while preparing database URL as mentioned in the below example.

4. **Execute a query.** Requires using an object of type Statement for building and submitting an SQL statement to the database.
5. **Clean up the environment.** Requires explicitly closing all database resources versus relying on the JVM's garbage collection.

Sample Code:

```
//Step 1: Import required package
import java.sql.*;
public class JDBC_CreateDatabase {
	public static void main(String args[])
	{
		String url, usr, pass;
		Connection conn=null;
		Statement state=null;
		try{
			//step 2: register JDBC Driver
			Class.forName("org.postgresql.Driver");
			System.out.println("Driver OK!");

			//step 3: open a connection
			System.out.println("Connecting to the database ...");
			url="jdbc:postgresql://localhost:5432/";
			usr="user";
			pass="password";
			conn=DriverManager.getConnection(url, usr, pass);

			//step 4: Execute a query
			System.out.println("Creating Database ...");
			state=conn.createStatement();

			String query;
			query="CREATE DATABASE students";
			state.executeUpdate(query);
			System.out.println("Database created successfully");
		}
		catch(SQLException se)
		{
			//handle error for JDBC
			se.printStackTrace();
		}
		catch(Exception e)
		{
			//handle error for Class.forName
			e.printStackTrace();
		}
	}
}
```

Drop Database

Sample codes

```
//STEP 1. Import required packages

import java.sql.*;
public class JDBCExample2 {
      public static void main(String args[])
      {
            String url, usr, pass, query;
            Statement state= null;
            Connection conn=null;

            try{
               //STEP 2: Register JDBC driver

                  Class.forName("org.postgresql.Driver");
                  System.out.println("Driver installed");

               //STEP 3: Open a connection

                  url="jdbc:postgresql://localhost:5432/";
                  usr="user";
                  pass="password";

                  conn=DriverManager.getConnection(url, usr, pass);
                  System.out.println("Connected successifully ...");

               //STEP 4: Execute a query

                  System.out.println("Deleting database ...");
                  state=conn.createStatement();
                  query="DROP DATABASE students";
                  state.executeUpdate(query);
                  System.out.println("Database deleted successfully ...");
            }
            catch(SQLException se)
            {
                  //handle error for JDBC
                  se.printStackTrace();
            }
            catch(Exception e)
            {
                  //handle error for Class.forName
                  e.printStackTrace();
            }
      }
}
```

Create Tables

Sample codes

```
//STEP 1. Import required packages
import java.sql.*;

public class JDBC_CreateTable {

	public static void main(String[] args) {
		String DB_URL = null;
		String USER = "user";
		String PASS = "password";
		Connection conn = null;
		Statement state = null;
		try {
			DB_URL = "jdbc:postgresql://localhost:5432/students";
			// STEP 2: Register JDBC driver
			Class.forName("org.postgresql.Driver");

			// STEP 3: Open a connection
			System.out.println("Connecting to a selected database...");
			conn = DriverManager.getConnection(DB_URL, USER, PASS);
			System.out.println("Connected database successfully...");

			// STEP 4: Execute a query
			System.out.println("Creating table in given database...");
			state = conn.createStatement();

			String sql = "CREATE TABLE registration "
					+ "(id INTEGER not NULL, " + " first
VARCHAR(255), "
					+ " last VARCHAR(255), " + " age INTEGER, "
					+ " PRIMARY KEY ( id ))";

			state.executeUpdate(sql);
			System.out.println("Created table in given database...");
		} catch (SQLException se) {
			// Handle errors for JDBC
			se.printStackTrace();
		} catch (Exception e) {
			// Handle errors for Class.forName
			e.printStackTrace();
		}
	}
}
```

Drop Tables

NOTE: This is a serious operation and you have to make a firm decision before proceeding to delete a table because everything you have in your table would be lost.

Codes:

```
//STEP 1. Import required packages
import java.sql.*;

public class JDBC_DropTable {

	public static void main(String[] args) {
		String usr, pass, url, sql;
		Connection conn = null;
		Statement stmt = null;
		try {
			// STEP 2: Register JDBC driver
			Class.forName("org.postgresql.Driver");

			// STEP 3: Open a connection
			System.out.println("Connecting to a selected database...");
			url = "jdbc:postgresql://localhost:5432/students";
			usr = "username";
			pass = "password";
			conn = DriverManager.getConnection(url, usr, pass);
			System.out.println("Connected database successfully...");

			// STEP 4: Execute a query
			System.out.println("Deleting table in given database...");
			stmt = conn.createStatement();

			sql = "DROP TABLE REGISTRATION ";

			stmt.executeUpdate(sql);
			System.out.println("Table  deleted in given database...");
		}// end try
		catch (SQLException se) {
			// Handle errors for JDBC
			se.printStackTrace();
		} catch (Exception e) {
			// Handle errors for Class.forName
			e.printStackTrace();
		}
	}


}// end JDBC_DropTable
```

Insert Records

```
import java.sql.*;

public class InsertDemo {

	public static void main(String[] args) {
		String usr, pass, url, sql;
		Connection conn = null;
		Statement stmt = null;

		try {
			// STEP 2: Register JDBC driver
			Class.forName("org.postgresql.Driver");

			// STEP 3: Open a connection
			System.out.println("Connecting to a selected database...");
			url = "jdbc:postgresql://localhost:5432/students";
			usr = "user";
			pass = "password";
			conn = DriverManager.getConnection(url, usr, pass);
			System.out.println("Connected database successfully...");

			// STEP 4: Execute a query
			System.out.println("Inserting records into the table...");
			stmt = conn.createStatement();

			sql = "INSERT INTO Registration "
					+ "VALUES (100, 'Zara', 'Ali', 18)";
			stmt.executeUpdate(sql);
			sql = "INSERT INTO Registration "
					+ "VALUES (101, 'Mahnaz', 'Fatma', 25)";
			stmt.executeUpdate(sql);
			sql = "INSERT INTO Registration "
					+ "VALUES (102, 'Zaid', 'Khan', 30)";
			stmt.executeUpdate(sql);
			sql = "INSERT INTO Registration "
					+ "VALUES(103, 'Sumit', 'Mittal', 28)";
			stmt.executeUpdate(sql);
			System.out.println("Inserted records into the table...");

		} catch (SQLException se) {
			// Handle errors for JDBC
			se.printStackTrace();
		} catch (Exception e) {
			// Handle errors for Class.forName
			e.printStackTrace();
		}
		System.out.println("Goodbye!");
	}// end main
}// end
```

Select Records

Codes

```
import java.sql.*;

public class SelectDemo {
	public static void main(String[] args) {
		String usr, pass, url, sql;
		Connection conn = null;
		Statement stmt = null;
		ResultSet result = null;

		try {
			// STEP 2: Register JDBC driver
			Class.forName("org.postgresql.Driver");

			// STEP 3: Open a connection
			System.out.println("Connecting to a selected database...");
			url = "jdbc:postgresql://localhost:5432/students";
			usr = "user";
			pass = "password";
			conn = DriverManager.getConnection(url, usr, pass);
			System.out.println("Connected database successfully...");

			// STEP 4: Execute a query
			System.out.println("Creating statement...");
			stmt = conn.createStatement();

			sql = "SELECT id, first, last, age FROM Registration";
			result = stmt.executeQuery(sql);
			// STEP 5: Extract data from result set
			while (result.next()) {
				// Retrieve by column name
				int id = result.getInt("id");
				int age = result.getInt("age");
				String first = result.getString("first");
				String last = result.getString("last");

				// Display values
				System.out.print("ID: " + id);
				System.out.print(", Age: " + age);
				System.out.print(", First: " + first);
				System.out.println(", Last: " + last);
			}
			result.close();
		} catch (SQLException se) {
			// Handle errors for JDBC
			se.printStackTrace();
		} catch (Exception e) {
			// Handle errors for Class.forName
			e.printStackTrace();
		}
		System.out.println("Goodbye!");
	}// end main
}
```

Update Records

Codes:

```
import java.sql.*;

public class UpdateDemo {

	public static void main(String[] args) {
		String usr, pass, url, sql;
		Connection conn = null;
		Statement stmt = null;

		try {
			// STEP 2: Register JDBC driver
			Class.forName("org.postgresql.Driver");

			// STEP 3: Open a connection
			System.out.println("Connecting to a selected database...");
			url = "jdbc:postgresql://localhost:5432/students";
			usr = "user";
			pass = "password";
			conn = DriverManager.getConnection(url, usr, pass);
			System.out.println("Connected database successfully...");

			// STEP 4: Execute a query
			System.out.println("Creating statement...");
			stmt = conn.createStatement();
			sql = "UPDATE Registration "
					+ "SET age = 30 WHERE id in (100, 101)";
			stmt.executeUpdate(sql);
			// Now you can extract all the records
			// to see the updated records
			sql = "SELECT id, first, last, age FROM Registration";
			ResultSet rs = stmt.executeQuery(sql);

			while (rs.next()) {
				// Retrieve by column name
				int id = rs.getInt("id");
				int age = rs.getInt("age");
				String first = rs.getString("first");
				String last = rs.getString("last");
				System.out.print("ID: " + id);
				System.out.print(", Age: " + age);
				System.out.print(", First: " + first);
				System.out.println(", Last: " + last);
			}
			rs.close();
		} catch (SQLException se) {
			se.printStackTrace();
		} catch (Exception e) {
			e.printStackTrace();
		}
		System.out.println("Goodbye!");
	}// end main
}
```

Delete Records

```
import java.sql.*;

public class DeleteDemo {

	public static void main(String[] args) {
		String usr, pass, url, sql;
		Connection conn = null;
		Statement stmt = null;

		try {
			// STEP 2: Register JDBC driver
			Class.forName("org.postgresql.Driver");

			// STEP 3: Open a connection
			System.out.println("Connecting to a selected database...");
			url = "jdbc:postgresql://localhost:5432/students";
			usr = "user";
			pass = "password";
			conn = DriverManager.getConnection(url, usr, pass);
			System.out.println("Connected database successfully...");

			// STEP 4: Execute a query
			System.out.println("Creating statement...");
			stmt = conn.createStatement();
			sql = "DELETE FROM Registration WHERE id = 101";
			stmt.executeUpdate(sql);

			// Now you can extract all the records
			// to see the remaining records
			sql = "SELECT id, first, last, age FROM Registration";
			ResultSet rs = stmt.executeQuery(sql);

			while (rs.next()) {
				// Retrieve by column name
				int id = rs.getInt("id");
				int age = rs.getInt("age");
				String first = rs.getString("first");
				String last = rs.getString("last");
				System.out.print("ID: " + id);
				System.out.print(", Age: " + age);
				System.out.print(", First: " + first);
				System.out.println(", Last: " + last);
			}
			rs.close();
		} catch (SQLException se) {
			// Handle errors for JDBC
			se.printStackTrace();
		} catch (Exception e) {
			// Handle errors for Class.forName
			e.printStackTrace();
		}
		System.out.println("Goodbye!");
	}// end main
}
```

WHERE Clause

This would add additional conditions using WHERE clause while selecting records from the table.

```
import java.sql.*;

public class WhereClauseDemo {

	public static void main(String[] args) {
		String usr, pass, url, sql;
		Connection conn = null;
		Statement stmt = null;

		try {
			// STEP 2: Register JDBC driver
			Class.forName("org.postgresql.Driver");

			// STEP 3: Open a connection
			System.out.println("Connecting to a selected database...");
			url = "jdbc:postgresql://localhost:5432/students";
			usr = "user";
			pass = "password";
			conn = DriverManager.getConnection(url, usr, pass);
			System.out.println("Connected database successfully...");

			// STEP 4: Execute a query
			System.out.println("Creating statement...");
			stmt = conn.createStatement();

			// Extract records without any condition.
			System.out.println("Fetching records without
condition...");
			sql = "SELECT id, first, last, age FROM Registration";
			ResultSet rs = stmt.executeQuery(sql);

			while (rs.next()) {
				// Retrieve by column name
				int id = rs.getInt("id");
				int age = rs.getInt("age");
				String first = rs.getString("first");
				String last = rs.getString("last");

				// Display values
				System.out.print("ID: " + id);
				System.out.print(", Age: " + age);
				System.out.print(", First: " + first);
				System.out.println(", Last: " + last);
			}

			// Select all records having ID equal or greater than 101
			System.out.println();
			System.out.println("Fetching records with condition...");
			sql = "SELECT id, first, last, age FROM Registration"
					+ " WHERE id >= 101 ";
			rs = stmt.executeQuery(sql);
```

```
                while (rs.next()) {
                        // Retrieve by column name
                        int id = rs.getInt("id");
                        int age = rs.getInt("age");
                        String first = rs.getString("first");
                        String last = rs.getString("last");

                        // Display values
                        System.out.print("ID: " + id);
                        System.out.print(", Age: " + age);
                        System.out.print(", First: " + first);
                        System.out.println(", Last: " + last);
                }
                rs.close();
            } catch (SQLException se) {
                // Handle errors for JDBC
                se.printStackTrace();
            } catch (Exception e) {
                // Handle errors for Class.forName
                e.printStackTrace();
            }
            System.out.println("Goodbye!");
        }// end main

}
```

LIKE Clause

This would add additional conditions using LIKE clause while selecting records from the table.

```
import java.sql.*;

public class LikeClauseDemo {
      public static void main(String args[]) {
            String usr, pass, url, sql;
            Connection conn = null;
            Statement stmt = null;

            try {
                  // STEP 2: Register JDBC driver
                  Class.forName("org.postgresql.Driver");

                  // STEP 3: Open a connection
                  System.out.println("Connecting to a selected database...");
                  url = "jdbc:postgresql://localhost:5432/students";
                  usr = "user";
                  pass = "password";
                  conn = DriverManager.getConnection(url, usr, pass);
                  System.out.println("Connected database successfully...");

                  // STEP 4: Execute a query
                  System.out.println("Creating statement...");
                  stmt = conn.createStatement();

                  // Extract records without any condition.
```

```
                System.out.println("Fetching records without
condition...");
                sql = "SELECT id, first, last, age FROM Registration";
                ResultSet rs = stmt.executeQuery(sql);

                while (rs.next()) {
                      // Retrieve by column name
                      int id = rs.getInt("id");
                      int age = rs.getInt("age");
                      String first = rs.getString("first");
                      String last = rs.getString("last");

                      // Display values
                      System.out.print("ID: " + id);
                      System.out.print(", Age: " + age);
                      System.out.print(", First: " + first);
                      System.out.println(", Last: " + last);
                }

                // Select all records having ID equal or greater than 101
                System.out.println("Fetching records with condition...");
                sql = "SELECT id, first, last, age FROM Registration"
                            + " WHERE first LIKE '%r%' ";
                rs = stmt.executeQuery(sql);

                while (rs.next()) {
                      // Retrieve by column name
                      int id = rs.getInt("id");
                      int age = rs.getInt("age");
                      String first = rs.getString("first");
                      String last = rs.getString("last");

                      // Display values
                      System.out.print("ID: " + id);
                      System.out.print(", Age: " + age);
                      System.out.print(", First: " + first);
                      System.out.println(", Last: " + last);
                }
                rs.close();
          } catch (SQLException se) {
                // Handle errors for JDBC
                se.printStackTrace();
          } catch (Exception e) {
                // Handle errors for Class.forName
                e.printStackTrace();
          }
          System.out.println("Goodbye!");
     }
}
```

Sorting Data

This would use **asc** and **desc** keywords to sort records in ascending or descending order.

```
import java.sql.Connection;
import java.sql.DriverManager;
```

```
import java.sql.Statement;

import java.sql.*;

public class SortingDemo {

	public static void main(String args[]) {
		String usr, pass, url, sql;
		Connection conn = null;
		Statement stmt = null;

		try {
			// STEP 2: Register JDBC driver
			Class.forName("org.postgresql.Driver");

			// STEP 3: Open a connection
			System.out.println("Connecting to a selected database...");
			url = "jdbc:postgresql://localhost:5432/students";
			usr = "user";
			pass = "password";
			conn = DriverManager.getConnection(url, usr, pass);
			System.out.println("Connected database successfully...");
			// STEP 4: Execute a query
			System.out.println("Creating statement...");
			stmt = conn.createStatement();

			// Extract records in ascending order by first name.
			System.out.println();
			System.out.println("Fetching records in ascending
order...");

			System.out.println("--------------------------------------
");
			sql = "SELECT id, first, last, age FROM Registration"
					+ " ORDER BY first ASC";
			ResultSet rs = stmt.executeQuery(sql);

			while (rs.next()) {
				// Retrieve by column name
				int id = rs.getInt("id");
				int age = rs.getInt("age");
				String first = rs.getString("first");
				String last = rs.getString("last");

				// Display values
				System.out.print("ID: " + id);
				System.out.print(", Age: " + age);
				System.out.print(", First: " + first);
				System.out.println(", Last: " + last);
			}

			// Extract records in descending order by first name.
			System.out.println();
			System.out.println("Fetching records in descending
order...");
			System.out.println("--------------------------------------
");
```

```
                sql = "SELECT id, first, last, age FROM Registration"
                            + " ORDER BY first DESC";
                rs = stmt.executeQuery(sql);

                while (rs.next()) {
                      // Retrieve by column name
                      int id = rs.getInt("id");
                      int age = rs.getInt("age");
                      String first = rs.getString("first");
                      String last = rs.getString("last");

                      // Display values
                      System.out.print("ID: " + id+"\t|");
                      System.out.print(" Age: " + age+"\t|");
                      System.out.print(" First: " + first+"\t|");
                      System.out.println(" Last: " + last+"\t|");
                }
                rs.close();
          } catch (SQLException se) {
                // Handle errors for JDBC
                se.printStackTrace();
          } catch (Exception e) {
                // Handle errors for Class.forName
                e.printStackTrace();
          }
          System.out.println("Goodbye!");
    }// end main

}
```

4.5 More practical Examples:

The next example allows the user to enter any query into the program. The example displays the result of a query in a **JTable**, using a **TableModel** object to provide the ResultSet data to the JTable.

Class ResultSetTableModel performs the connection to the database via a TableModel and maintains the ResultSet. Class DisplayQueryResults creates the GUI and specifies an instance of class ResultSetTableModel to provide data for the JTable.

ResultSetTableModel Class

Class ResultSetTableModel extends class **AbstractTableModel** (package javax.swing.table), which implements interface TableModel. ResultSetTableModel overrides TableModel methods **getColumnClass**, **getColumnCount**, **getColumnName**, **getRowCount** and **getValueAt**. The default implementations of TableModel methods is CellEditable and setValueAt (provided by AbstractTableModel) are not overridden, because this example does not support editing the

JTable cells. The default implementations of TableModel methods **addTableModelListener** and **removeTableModelListener** (provided by AbstractTableModel) are not overridden, because the implementations of these methods in AbstractTableModel properly add and remove event listeners.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.sql.Statement;

import javax.swing.table.AbstractTableModel;


public class ResultSetTableModel extends AbstractTableModel {
      private Connection connection;
         private Statement statement;
         private ResultSet resultSet;
         private ResultSetMetaData metaData;
         private int numberOfRows;

         // keep track of database connection status
         private boolean connectedToDatabase = false;

         // constructor initializes resultSet and obtains its meta data
object;
         // determines number of rows
         public ResultSetTableModel( String driver,String url, String
username,
            String password, String query )
            throws SQLException, ClassNotFoundException
         {
            // connect to database
            Class.forName( driver );
            connection = DriverManager.getConnection( url, username, password
);

            // create Statement to query database
            statement = connection.createStatement(
               ResultSet.TYPE_SCROLL_INSENSITIVE,
               ResultSet.CONCUR_READ_ONLY );

            // update database connection status
            connectedToDatabase = true;

            // set query and execute it
            setQuery( query );
         } // end constructor ResultSetTableModel

         // get class that represents column type
         public Class getColumnClass( int column ) throws
IllegalStateException
         {
```

```
            // ensure database connection is available
            if ( !connectedToDatabase )
               throw new IllegalStateException( "Not Connected to Database"
);

            // determine Java class of column
            try
            {
               String className = metaData.getColumnClassName( column + 1 );

               // return Class object that represents className
               return Class.forName( className );
            } // end try
            catch ( Exception exception )
            {
               exception.printStackTrace();
            } // end catch

            return Object.class; // if problems occur above, assume type
Object
         } // end method getColumnClass

         // get number of columns in ResultSet
         public int getColumnCount() throws IllegalStateException
         {
            // ensure database connection is available
            if ( !connectedToDatabase )
               throw new IllegalStateException( "Not Connected to Database"
);

            // determine number of columns
            try
            {
               return metaData.getColumnCount();
            } // end try
            catch ( SQLException sqlException )
            {
               sqlException.printStackTrace();
            } // end catch

            return 0; // if problems occur above, return 0 for number of
columns
         } // end method getColumnCount

         // get name of a particular column in ResultSet
         public String getColumnName( int column ) throws
IllegalStateException
         {
            // ensure database connection is available
            if ( !connectedToDatabase )
               throw new IllegalStateException( "Not Connected to Database"
);

            // determine column name
            try
            {
               return metaData.getColumnName( column + 1 );
```

```
            } // end try
            catch ( SQLException sqlException )
            {
               sqlException.printStackTrace();
            } // end catch

            return ""; // if problems, return empty string for column name
         } // end method getColumnName

         // return number of rows in ResultSet
         public int getRowCount() throws IllegalStateException
         {
            // ensure database connection is available
            if ( !connectedToDatabase )
               throw new IllegalStateException( "Not Connected to Database"
);

            return numberOfRows;
         } // end method getRowCount

         // obtain value in particular row and column
         public Object getValueAt( int row, int column )
            throws IllegalStateException
         {
            // ensure database connection is available
            if ( !connectedToDatabase )
               throw new IllegalStateException( "Not Connected to Database"
);

            // obtain a value at specified ResultSet row and column
            try
            {
               resultSet.absolute( row + 1 );
               return resultSet.getObject( column + 1 );
            } // end try
            catch ( SQLException sqlException )
            {
               sqlException.printStackTrace();
            } // end catch

            return ""; // if problems, return empty string object
         } // end method getValueAt

         // set new database query string
         public void setQuery( String query )
            throws SQLException, IllegalStateException
         {
            // ensure database connection is available
            if ( !connectedToDatabase )
               throw new IllegalStateException( "Not Connected to Database"
);

            // specify query and execute it
            resultSet = statement.executeQuery( query );

            // obtain meta data for ResultSet
            metaData = resultSet.getMetaData();
```

```
            // determine number of rows in ResultSet
            resultSet.last();                   // move to last row
            numberOfRows = resultSet.getRow();  // get row number

            // notify JTable that model has changed
            fireTableStructureChanged();
         } // end method setQuery

         // close Statement and Connection
         public void disconnectFromDatabase()
         {
            if ( connectedToDatabase )
            {
               // close Statement and Connection
               try
               {
                  resultSet.close();
                  statement.close();
                  connection.close();
               } // end try
               catch ( SQLException sqlException )
               {
                  sqlException.printStackTrace();
               } // end catch
               finally  // update database connection status
               {
                  connectedToDatabase = false;
               } // end finally
            } // end if

         } // end method disconnectFromDatabase
```

DisplayQueryResults Class

Class DisplayQueryResults implements the application's GUI and interacts with the ResultSetTableModel via a JTable object. This application also demonstrates the JTable sorting and filtering capabilities.

```
// Display the contents of the database table in a jtable

import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.sql.SQLException;
import java.util.regex.PatternSyntaxException;

import javax.swing.Box;
import javax.swing.JButton;
import javax.swing.JFrame;
```

```
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.RowFilter;
import javax.swing.ScrollPaneConstants;
import javax.swing.table.TableModel;
import javax.swing.table.TableRowSorter;

public class DisplayQueryResults extends JFrame {
      // JDBC database URL, username and password
         static final String DRIVER = "com.mysql.jdbc.Driver";
         static final String DATABASE_URL = "jdbc:mysql://localhost/books";
         static final String USERNAME = "user";
         static final String PASSWORD = "password";

         // default query retrieves all data from authors table
         static final String DEFAULT_QUERY = "SELECT * FROM authors";

         private ResultSetTableModel tableModel;
         private JTextArea queryArea;

         // create ResultSetTableModel and GUI
         public DisplayQueryResults()
         {
            super( "Displaying Query Results" );

            // create ResultSetTableModel and display database table
            try
            {
               // create TableModel for results of query SELECT * FROM
authors
               tableModel = new ResultSetTableModel( DRIVER, DATABASE_URL,
                  USERNAME, PASSWORD, DEFAULT_QUERY );

               // set up JTextArea in which user types queries
               queryArea = new JTextArea( DEFAULT_QUERY, 3, 100 );
               queryArea.setWrapStyleWord( true );
               queryArea.setLineWrap( true );

               JScrollPane scrollPane = new JScrollPane( queryArea,
                  ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED,
                  ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER );

               // set up JButton for submitting queries
               JButton submitButton = new JButton( "Submit Query" );

               // create Box to manage placement of queryArea and
               // submitButton in GUI
               Box boxNorth = Box.createHorizontalBox();
               boxNorth.add( scrollPane );
               boxNorth.add( submitButton );

               // create JTable delegate for tableModel
               JTable resultTable = new JTable( tableModel );
```

```
                JLabel filterLabel = new JLabel( "Filter:" );
                final JTextField filterText = new JTextField();
                JButton filterButton = new JButton( "Apply Filter" );
                Box boxSouth = boxNorth.createHorizontalBox();

                boxSouth.add( filterLabel );
                boxSouth.add( filterText );
                boxSouth.add( filterButton );

                // place GUI components on content pane
                add( boxNorth, BorderLayout.NORTH );
                add( new JScrollPane( resultTable ), BorderLayout.CENTER );
                add( boxSouth, BorderLayout.SOUTH );

                // create event listener for submitButton
                submitButton.addActionListener(

                   new ActionListener()
                   {
                      // pass query to table model
                      public void actionPerformed( ActionEvent event )
                      {
                         // perform a new query
                         try
                         {
                            tableModel.setQuery( queryArea.getText() );
                         } // end try
                         catch ( SQLException sqlException )
                         {
                            JOptionPane.showMessageDialog( null,
                               sqlException.getMessage(), "Database error",
                               JOptionPane.ERROR_MESSAGE );

                            // try to recover from invalid user query
                            // by executing default query
                            try
                            {
                               tableModel.setQuery( DEFAULT_QUERY );
                               queryArea.setText( DEFAULT_QUERY );
                            } // end try
                            catch ( SQLException sqlException2 )
                            {
                               JOptionPane.showMessageDialog( null,
                                  sqlException2.getMessage(), "Database
error",
                                  JOptionPane.ERROR_MESSAGE );

                               // ensure database connection is closed
                               tableModel.disconnectFromDatabase();

                               System.exit( 1 ); // terminate application
                            } // end inner catch
                         } // end outer catch
                      } // end actionPerformed
                   }  // end ActionListener inner class
                ); // end call to addActionListener
```

```
            final TableRowSorter< TableModel > sorter =
               new TableRowSorter< TableModel >( tableModel );
            resultTable.setRowSorter( sorter );
            setSize( 500, 250 ); // set window size
            setVisible( true ); // display window

            // create listener for filterButton
            filterButton.addActionListener(
               new ActionListener()
               {
                  // pass filter text to listener
                  public void actionPerformed( ActionEvent e )
                  {
                     String text = filterText.getText();

                     if ( text.length() == 0 )
                        sorter.setRowFilter( null );
                     else
                     {
                        try
                        {
                           sorter.setRowFilter(
                              RowFilter.regexFilter( text ) );
                        } // end try
                        catch ( PatternSyntaxException pse )
                        {
                           JOptionPane.showMessageDialog( null,
                              "Bad regex pattern", "Bad regex pattern",
                              JOptionPane.ERROR_MESSAGE );
                        } // end catch
                     } // end else
                  } // end method actionPerfomed
               } // end annonymous inner class
            ); // end call to addActionLister
         } // end try
         catch ( ClassNotFoundException classNotFound )
         {
            JOptionPane.showMessageDialog( null,
               "Database Driver not found", "Driver not found",
               JOptionPane.ERROR_MESSAGE );

            System.exit( 1 ); // terminate application
         } // end catch
         catch ( SQLException sqlException )
         {
            JOptionPane.showMessageDialog( null,
sqlException.getMessage(),
               "Database error", JOptionPane.ERROR_MESSAGE );

            // ensure database connection is closed
            tableModel.disconnectFromDatabase();

            System.exit( 1 ); // terminate application
         } // end catch

         // dispose of window when user quits application (this overrides
```

```
            // the default of HIDE_ON_CLOSE)
            setDefaultCloseOperation( DISPOSE_ON_CLOSE );

            // ensure database connection is closed when user quits
application
            addWindowListener(

               new WindowAdapter()
               {
                  // disconnect from database and exit when window has closed
                  public void windowClosed( WindowEvent event )
                  {
                     tableModel.disconnectFromDatabase();
                     System.exit( 0 );
                  } // end method windowClosed
               } // end WindowAdapter inner class
            ); // end call to addWindowListener
         } // end DisplayQueryResults constructor

         // execute application
         public static void main( String args[] )
         {
            new DisplayQueryResults();
         } // end main

}
```