REMINDER

Since one card in a Card Layout can be viewed at a time, the above diagram displays only the last card. To display each card in sequence, an interval period between the display of cards has to be specified.

The *Card* Applet features a card layout with five cards. Each card within the array of cards that Applet holds is assigned a static text message. This is done using an array of Labels, specifying some text to be displayed on each label and then binding a label to a card.

This is done by using the *add()* method.

The *show()* method is used to display each element from the card on to the applet.

Scanned by CamScanner

Chapter

28

SECTION VI: CLIENT SIDE PROGRAMMING

Swing

Swing, is a Java Foundation Classes [JFC] library and an extension of the Abstract Window Toolkit [AWT]. Swing has been integrated into Java 2. Swing offers much improved functionality over AWT, new components, expanded components features, excellent event handling with drag and drop support.

The Java Foundation Classes

Swing, has about four times the number of User Interface [UI] components as AWT and is part of the standard Java distribution. By today's application GUI requirements, AWT is a limited implementation, not quite capable of providing the components required for developing complex GUI's required in modern commercial applications. The AWT component set has a quite a few bugs and really does take up a lot of system resources when compared to equivalent Swing resources.

Scanned by CamScanner

Netscape introduced its Internet Foundation Classes [IFC] library for use with Java. It's classes became very popular with programmers creating GUI's for commercial applications. Sun decided to act and the joint effort between Sun and Netscape produced the original Swing set of components as part of the Java Foundation Classes [JFC].

Many programmers think that JFC and Swing is one and the same thing, but that is not so. JFC contains Swing and quite a number of other items. The JFC contain:

- **Swing:** A large UI component package
- **Cut and paste:** Clipboard support
- **Accessibility features:** Aimed at developing GUI's for users with disabilities
- **The Desktop Colors features:** First introduced in Java 1.1
- **Java 2D:** Improved color, image and text support

Additionally, under the hood Swing introduced significant advantages to programmers:

- It's UI components use few system resources and thus are lightweight components
- Offers a whole lot more sophisticated UI components with advanced fucntionality
  All of which helps to programmers deliver a consistant look and feel of program GUI's

Swing was first released as an extension API to JDK 1.1 that had to be downloaded separately and then merged with the JDK. With JDK 1.2, JFC and Swing are integrated directly into the JDK.

JFC Technologies

JFC contains two major technologies, one of which is Swing.

What JFC also contains is the AWT, Accessibility, Drag and Drop and 2D Graphics.

A brief description of the constituents of JFC follows:

- **Accessibility API:** These are libraries that empower programmers to create GUI's for use by disabled [dierently enabled] people who need additional help when using an applicatgion's GUI

  The assistive systems include screen readers, screen magnifiers and speech recognition sub-systems. The Accessibility API provides an interface that allows assistive technologies to interact and communicate with JFC and AWT components.

Scanned by CamScanner

- **Abstract Windowing Toolkit:** The AWT is not new to Java programmers. The AWT is the cornerstone of the JFC itself and is one of the core libraries launched with JDK 1.0. In fact, the AWT is the foundation on which Swing components were built. Even though user interfaces can be created using Swing components, layout managers and event models supported by the AWT have to be used for positioning these components on the GUI
- **2D Graphics:** The API in the AWT supports graphics to some extent. However, as Java technology started being used extensively, the demand for a more sophisticated graphics API raised. JDK 1.2 has arrived with 2D graphics that enhances the existing graphics support. The 2D Graphics API can support advanced 2D graphics and imaging
- **Drag and Drop:** With native platform capablilities of drag and drop, a user with an external non Java application working adjacent to a Java application, will be able to drag and drop information between the Java application and the external application

Swing Features

Model View Controller [MVC] architecture is normally used when creating GUI's using Swing components. MVC architecture allows Swing components to be replaced with different data models and views. The pluggable look and feel of commercial application GUI's is a direct result of MVC architecture.

Java is a platform independent language and runs on any client machine, the GUI look and feel, owned and delivered by a platform specific O/S, simply does not effect an application's GUI constructed using Swing components.

A summary of Swing's key features:

- **Lightweight Components:** Starting with the JDK 1.1, its AWT supported lightweight component development. For a component to qualify as lightweight, it must not depend on any non Java [O/s based] system classes. Swing components have their own view supported by Java's look and feel classes
- **Pluggable Look and Feel:** This feature enables the user to switch the look and feel of Swing components without restarting an application. The Swing library supports a components look and feel that remains the same across all platforms wherever the program runs. The Swing library provides an API that gives real flexibility in determining the look and feel of the GUI of an application

Scanned by CamScanner

Swing Components

Swing is a package built on top of the AWT that provides a great number of pre built classes [over 250 classes and 40 UI components]. From a commercial application developer's point of view, Swing's UI components are probably the most interesting.

Swing Components	Features
JApplet	An extended version of **java.applet.Applet** that adds support for root panes and other panes
JButton	A command button
JCheckBox	A checkbox that can be user selected or deselected and displays its state visually on the GUI
JCheckMenuItem	A menu item that can selected or deselected and displays its state visually on the GUI
JComboBox	A combo box, which is a combination of a text field and drop down list of selectable items
JDialog	The base class for creating a dialog window
JFrame	An extended version of **java.awt.Frame** that adds support for root panes and other panes
JLabel	A display area that can hold a short text message, an image or both normally placed adjacent to a textbox as its prompt
JList	A component that allows the user to select one or more objects from a list
JMenu	A pop up menu containing JMenuItem that is displayed when the user selects it in the JMenuBar component
JMenuBar	An implementation of a menu bar
JMenuItem	An implementation of a menu item
JOptionPane	Make it easy to pop up a standard dialog box
JPanel	A generic lightweight container
JPasswordField	Allows editing of a single line of text where the view does not show the original characters
JPopupMenu	A pop up menu
JPopupMenuSeparator	A pop-up menu-specific separator
JProgressBar	A component that displays an integer value within an interval
JRadioButton	A radio button that can be selected or deselected, displaying its state visually
JRadioButtonMenuItem	A radio button menu item
JRootPane	The fundamental component in the container hierarchy
JScrollBar	An implementation of a scrollbar

Scanned by CamScanner

Swing Components	Features
JScrollPane	A container that manages a viewport, optional vertical and horizontal scrollbars and optional row and column heading viewports
JSlider	A component that lets the user select a value by sliding a knob within an interval
JTabbedPane	Lets the user switch between a group of components by clicking tabs
JTable	Presents data in a two-dimensional table format
JTextArea	A multi line area that accepts and displays user input as plain text
JTextField	A single line area that accepts and displays user input as plain text
JtextPane	A text component that can be marked up with attributes that are represented graphically
JToolBar	A toolbar, used for displaying commonly used controls as icons in a the GUI of a commercial application
JTree	Displays a set of hierarchical data as an tree outline

REMINDER

Since each GUI component name begins with J, many programmers mistakenly use the terms JFC and Swing interchangeably.

There is a Swing replacement for every AWT control and container except **JCanvas**. The reason is that the **JPanel** class already supports all that the **Canvas** component does, hence Sun did not find it necessary to add a separate **JCanvas** component.

Comparison between Swing and AWT

The biggest difference between the AWT and Swing components is that Swing components are implemented totally in Java. Swing components have more functionality than AWT components. Since Swing components do not borrow from no native code belonging to the O/s or other frameworks, they can be shipped as an add-on to JDK 1.1, in addition to being part of JDK 1.2.

The simplest Swing components have capabilities far beyond AWT components:

- Swing buttons and labels can display images instead of or in addition to text
- The borders around most Swing compoimnents can be changed easily. For example: it is easy to put a 1 pixel border around the outside of a Swing label
- Swing components do not have to be rectangular. Buttons, for example, can be round

Scanned by CamScanner

- Assertive technologies such as screen readers can easily get information from Swing components. For example: A screen reader tool can easily capture the text that is displayed on a Swing button or label

Swing allows specifying which **look and feel** the application's GUI has. By contrast, AWT components always have the look and feel of the O/s or other framework.

REMINDER

"**Look and Feel**" is an expression that is used often when describing application GUI programming. It referes to how an application's GUI looks and feels to the user of the application. Consistent *Look and Feel* features have became simple to implement in Java based application GUI's with the introduction of *Swing*.

All the components of *Swing* are within the *javax.swing* package. To use a *Swing* class, either use an *import* statement for a *specific class* to be imported from within Swing or *import* all classes in the *Swing* package as follows:

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

The Swing API is powerful, flexible and large. The JFC 1.1 release has 15 public packages such as javax.accessibility, javax.swing.colorchooser, javax.swing.filechooser, javax.swing.plaf.metal, javax.swing.tree and so on.

The following are Java Swing packages. Each class has its own specific fucntionality:

Swing Packages	Features
javax.swing	This high level swing package primarily contains all the components, adapters, default components models and interfaces for Swing
javax.swing.plaf.basic	This *basic* swing package contains the User Interface [UI] classes, which implement the default look and feel of swing components
javax.swing.border	The *border* package declares the *Border* interface and classes, which define specific border rendering styles that can be applied to Swing GUI componens
javax.swing.event	The *event* package is for the Swing-specific event types and listeners. Swing components can generate their own event types, in addition to the events specified in the *java.awt.event* package

Scanned by CamScanner

javax.swing.plaf.multi	The *multi* package contains the multiplexing of user interface classes, which permits the creation of components accessed from different user interfaces
javax.swing.plaf	The *plaf* package contains the Pluggable Look-And-Feel API, used to define custom interfaces
javax.swing.table	The *table* package contains the interfaces and classes that support the Swing *JTable* component
Swing Packages	**Features**
javax.swing.text	The *text* package contains the support classes for the Swing document framework
javax.swing.text.html	The *text.html* package contains the support classes for the rendering on content contained within HTML tags
javax.swing.undo	The *undo* package provides the support classes for implementing undo/redo capabilities in a GUI

Each AWT component has its own, operating system specific window and therefore ends up looking like a standard control belonging to that operating system. In commercial applications that have GUI's with a large number of interactive and control components having a O/s window bound to each GUI component slows performance and uses up a great deal of memory. Hence AWT components are called **heavyweight**.

Swing controls, on the other hand, are simply drawn as images in their containers and do not have a O/s window of their own bound to them, so they use far fewer system resources. Therefore, they are called **lightweight components**.

All Swing components are derived from the **JComponent** class. This class is, in turn, derived from the AWT **Container** class, which uses no heavyweight O/S based window, so **JComponent** is a lightweight class.

JComponent adds a tremendous amount of programming support to the AWT Container class. All Swing components are derived from **JComponent** and **JComponent** is derived from the AWT **Container** class, hence Swing components are also AWT components hence AWT controls can be mixed with Swing controls if required when creating an application's

Scanned by CamScanner

GUI. However, because Swing controls are merely drawn in their container, strange results can be expected because the AWT control will appear on top of Swing components.

Not all Swing components are lightweight. To display anything in a windowed environment, some O/s windows and other GUI component display objects are required. These O/s windows are normally used borrowed and lightweight Swing controls drawn within them. Hence, Swing supports heavyweight classes like **JFrame**, **JDialog**, **JWindow** and **JApplet**.

Swing applets are built on the JApplet class and Swing applications are built using the **JFrame** class. The fact that Swing is built on top of AWT is by no means seamless to the programmer. Swing heavyweight containers have a complex structure that takes some getting used to. To paint components on an applications GUI, the paint() method need not be overridden anymore because Swing needs to do that to draw its component borders and so on. Having said that, there is direct access to the parts of an application's GUI in which menus and dialog boxes must be drawn and used. All in all, it takes some additional programming effort to move from AWT to Swing.

Working With Swing

Using Swing components is similar to using AWT components. Swing components are created by creating an object of the component class and adding the component object to a container.

JFrame

A Swing frame is a container that functions as the main window for applications that use Swing components. A Swing frame has a title, a border and buttons for iconifying, maximizing and closing the frame.

Swing provides the *JFrame* class that is a part of the *javax.swing* package. The *JFrame* class is an abstract class. Swing's JFrame class extends the AWT Frame class. Hence the Swing frame is a heavyweight container, as is it is obtained from the AWT Frame class. An instance of **JFrame** can make use of features that belong to both the **JFrame** and **Frame** classes.

The *JFrame* class is subdivided into several different panes. The main pane is the *content pane*, which represents the full area of a frame into which components can be added.

Scanned by CamScanner

The *content pane* was introduced in JFrame to deal with the complexities which are involved in making lightweight and heavyweight components work together. The content pane basically manages the interior of a Swing frame. This means that GUI components be added to the JFrame's content pane rather than adding them directly to the frame itself.

To add a component to a JFrame:

- Create a JFrame object as follows

```
JFrame frm = new JFrame ();
```

- Get a handle to its content pane

```
Container contPane = frm.getContentPane( );
```

- The Add the component

```
contPane.add(someComponent);
```

The following table describes some of the methods provided by *JFrame*:

Methods	Description
void setContentPane(Container) Container getContentPane()	Set or get the frame's content pane. The content pane contains the frame's visible GUI components. The content pane is opaque.

Methods	Description
JRootPane createRootPane() void setRootPane(JRootPane) JRootPane getRootPane()	Create, set or get the frame's root pane. The root panel manages the interior of the frame including the content pane, the glass pane and so on
void setGlassPane(Component) Component getGlassPane()	Set or get the frame's glass pane. Use the glass pane to intercept mouse events
void setLayeredPane(JLayeredPane) JLayeredPane getLayeredPane()	Set or get the frame's layered pane. Use the frame's layered pane to put components on top of or behind other components

Example:
Creating a *JLabel* object.

```
JLabel lblHello = new JLabel("Hello", JLabel.CENTER );
```

The above code snippet creates a label **Hello** that will be aligned to the center of the content pane.

Example:
The following code snippent displays a center aligned label "Hello World" in a Frame.

Scanned by CamScanner

Code spec for HelloWorldFrame.java:

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

public class HelloWorldFrame extends JFrame
{

  public static void main(String[] args)
  {
      JFrame frm = new JFrame("Welcome");
  /* Retrieving the content pane of the frame */
      Container container = frm.getContentPane();
  /* Adding labels to the frame */
      container.add(new JLabel("Hello World", JLabel.CENTER));
      frm.setSize(200, 200);
      frm.setVisible(true);

      WindowListener listener = new WindowAdapter()
      {
          public void windowClosing(WindowEvent winEvt)
          {
              System.exit(0);
          }
      };
  /* Window listener activating the windowClosing() method */
      frm.addWindowListener(listener);
  }
}
```

Explanation:

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

Following three interfaces are imported:

- **javax.swing.*:** Packages a set of GUI components. Unlike AWT components, that are associated to native screen resources [heavyweight], Swing components draw themselves on the screen [lightweight]. This result in slower execution but a Swing application will look the same on all platforms
- **java.awt.event.*:** Provides interfaces and classes for dealing with different types of events fired by AWT and Swing components
- **java.awt.*:** Contains all of the classes required for creating user interfaces and for painting graphics and images

Scanned by CamScanner

```
public class HelloWorldFrame extends JFrame
{
  ...
}
```

The class named **HelloWorldFrame** is created, which extends the JFrame class.

```
public static void main(String[] args)
{
  ...
}
```

The **public** keyword is an access specifier, which allows the programmer to control who or what can access the class and its members. When a class member is preceded by **public**, then that class and/or its member may be accessed by code from outside the class in which it is declared. In this case, the **main()** method must be declared public, since it must be called by code external to it when the application is run. The keyword **static** allows the main() method to be called without having to instantiate an instance of the class. This is necessary since the main() method is called by the Java interpreter before any objects are spawned. The keyword **void** simply informs the compiler that the main() method does not return a value.

String[] args declares a parameter named **args**, which is an array that will hold multiple instances of the class String. Objects of type String store alpha numeric characters. In this case, **args** receives any command-line arguments presented when the program is executed.

```
JFrame frm = new JFrame("Welcome");
```

An object named frm of JFrame is created. Welcome text is written which is shown as the title when the application is run. A JFrame object is the physical window that the developer works with in the Swing API.

```
Container container = frm.getContentPane();
```

An object named container of Container is created. The Container is a component that can contain other components. The getContentPane() method of the JFrame class retrieves the content pane object for the frame.

```
container.add(new JLabel("Hello World", JLabel.CENTER));
```

A Label is added to the frame via the add() method of the Container class. The add() method is passed the JLabel object. A JLabel class is used to display the text on the frame. Two parameters are passed to the JLabel object viz. the text Hello World which is displayed and the alignment of the text to be displayed.

```
frm.setSize(200, 200);
frm.setVisible(true);
```

Scanned by CamScanner

The size of the frame is set via the setSize() method of the JFrame object. The setSize() method is passed two parameters viz. the width and the height of the frame in pixels

The frame is shown via the setVisible() method of the JFrame object. The value of the setVisible() method is set to true to make the frame visible.

```
WindowListener listener = new WindowAdapter()
{
  public void windowClosing(WindowEvent winEvt)
  {
      System.exit(0);
  }
};
```

A Frame does not close by itself when its close icon is clicked. For a Frame to close and free the resources its using a WindowAdapter() method needs to be added to the Frame. The WindowAdapter() knows when a Window's close icon is clicked and thus recognizes a user request for the window to be closed. This user close request is captured by the adapter and passed to JFrame so that JFrame knows that the window has to be closed. Consequently, JFrame closes itself, releasing memory and other resources that were being used.

The adapter class which is used in the above code actually implements a listener interface with abstract methods [i.e. methods that have no signature], so they do nothing. These must be programmatically overridden for work to be done. There is an adapter class for each low-level listener interface defined in the **java.awt.event** package, plus an adapter class that is defined in the **javax.swing.event** package bound to the listener interface.

For example: FocusAdapter, WindowAdapter, MouseAdapter, KeyAdapter and so on.

To handle a user request for closing the window in the above application, the Window Adapter class is instantiated and the windowClosing() method is implemented.

```
frm.addWindowListener(listener);
```

The addWindowListener() method of the JFrame class adds the specified window listener to receive window events from a particular window.

Once the .java file is ready, it needs to be compiled. The Swing application cannot be compiled by the asant command. It is compiled by the Java compiler [javac] and then the .class file is interpreted by the Java interpreter [java].

So to use the Java compiler and the Java interpreter, a path needs to be added to the PATH variable in the Systems in Windows and in .bashrc

Scanned by CamScanner

The **Java compiler** and the **Java interpreter** can be launched from the application's directory using the command **javac** and **java**, respectively, provided the system's or user's PATH variable holds the complete path to the Java Application server's **jdk/bin** directory in case of a *Microsoft Windows based O/s* it is **C:\Sun\AppServer\jdk\bin** and in case of a *Linux based O/s* it is **/opt/SUNWappserver/jdk/bin**.

PATH variable can be set using System's Environment Variable User Interface, for example in Windows XP use **Control Panel → System → Advanced → Environment Variables** as shown in diagram 28.0.1.

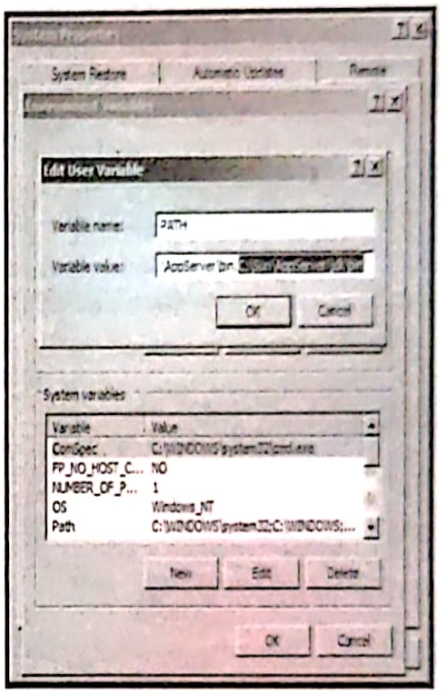


Diagram 28.0.1: Environment Variables On Windows

PATH variable on a Linux based O/s can be set using **.bashrc** file under the user's home directory. To do so, use a text editor such as **Kate** and open the .bashrc file available under /root/.bashrc as shown in diagram 35.0.2. This is assuming root is the user who works with Java and thus requires the **asant** utility.

HINT

.bashrc is a user-specific Bash init file, found in each user's home directory. Only interactive shells and user scripts read this file.

Scanned by CamScanner

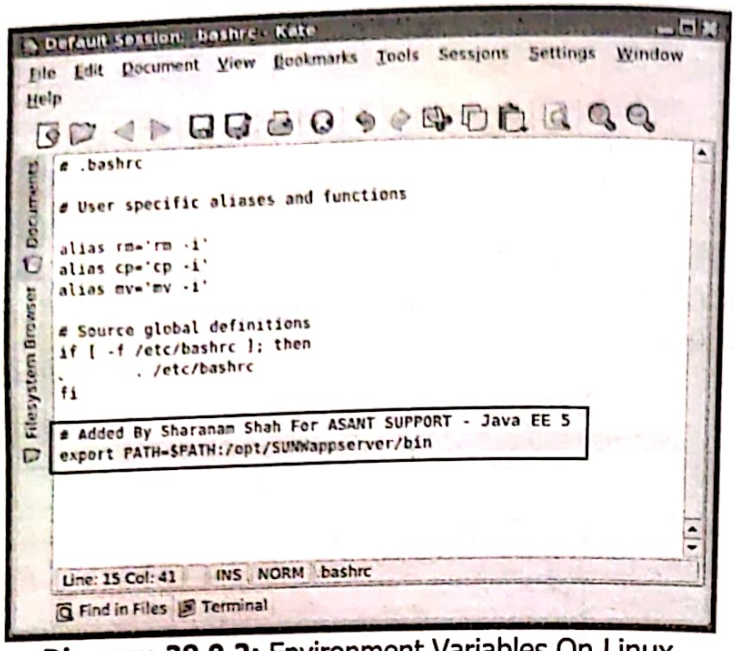


Diagram 28.0.2: Environment Variables On Linux

Add the export line as shown in diagram 28.0.2 and save the .bashrc file. Restart the terminal window for the change to take effect.

Restart the Java server.

Next at the command prompt type the following command:

```
<Command Prompt> javac HelloWorldFrame.java
```

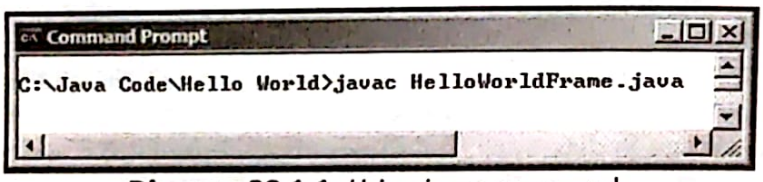


Diagram 28.1.1: Using javac command

javac is used to compile Java source code to platform independent Byte Code. This source code file must have a file extension of .java.

After compiling the source file with .java extension, *javac* produces an output file, which has a .class extension as shown in diagram 28.1.2. This is the file that all JVM interpret and execute.

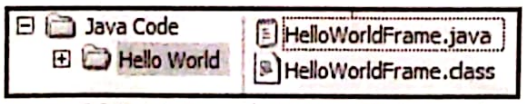


Diagram 28.1.2: Compiled .java file into .class file

Scanned by CamScanner

Next is the time to execute the .class file to see the output. At the command prompt type the following command:

```
<Command Prompt> java HelloWorldFrame
```

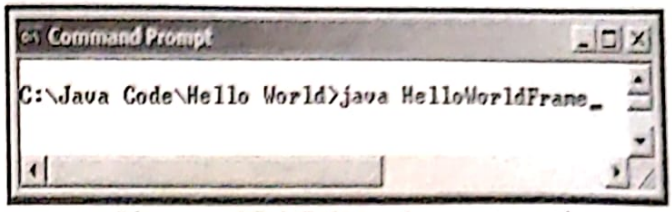


Diagram 28.1.3: Using java command

java is used to execute and interpret the .class file. After executing .class file, an output as shown in diagram 28.1.4 appears.

Diagram 28.1.4

JPanel

A JPanel is a Swing container [a component extension of *JComponent*] that is used for grouping components within one area of an applet or a frame. A panel can also be used to group other panels.

Panels are Swing components represented by the class *JPanel*, which belong to the *javax.swing* package. *JPanel* is a lightweight component.

To add a component to JPanel and then add the Panel to the Frame do the following:

- Create a JFrame object

```
JFrame frm = new JFrame ();
```

- Create a JPanel object

```
JPanel pnl = new JPanel ();
```

Scanned by CamScanner

... can be containers] to the *JPanel* object by using its *add()*

```
pnl.add(<ComponentName>);
```

- Add an intermediate *JPanel* object as part of the content pane by calling the *setContentPane()* method of the *JFrame* class

```
frm.setContentPane(pnl);
```

- Create a *JButton* object

```
JButton btnSave = new JButton ("Save");
```

The above code snippet creates a Button with the label **Save** displayed on it.

Example:

The following example displays four buttons on a panel.

Code spec for MasterForm.java:

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

public class MasterForm extends JFrame
{
  public MasterForm()
  {
      super("Address Book");

      JButton btnAdd = new JButton("Add");
      JButton btnEdit = new JButton("Edit");
      JButton btnDelete = new JButton("Delete");
      JButton btnView = new JButton("View");
      JPanel pnlMenu = new JPanel();

  /* Adding Buttons to the Panel */
      pnlMenu.add(btnAdd);
      pnlMenu.add(btnEdit);
      pnlMenu.add(btnDelete);
      pnlMenu.add(btnView);

  /* Creating Shortcut keys */
      btnAdd.setMnemonic('a');
      btnEdit.setMnemonic('e');
      btnDelete.setMnemonic('d');
      btnView.setMnemonic('v');

  /* Adding Tooltips for each button */
      btnAdd.setToolTipText("To add new contacts");
      btnEdit.setToolTipText("To edit existing contacts");
```

Scanned by CamScanner

```
        btnDelete.setToolTipText("To delete existing contacts");
        btnView.setToolTipText("To view existing contacts");

        setContentPane(pnlMenu);
    }

    public static void main(String[] args)
    {
    /* Calling the MasterForm constructor */
        MasterForm frmMaster = new MasterForm();

        frmMaster.setSize(300, 200);
        frmMaster.setVisible(true);

        WindowListener listener = new WindowAdapter()
        {
            public void windowClosing(WindowEvent winEvt)
            {
                System.exit(0);
            }
        };
        frmMaster.addWindowListener(listener);
    }
}
```

setDefaultCloseOperation (EXIT_ON_CLOSE)

Explanation:

```
public MasterForm()
{
  ...
}
```

A constructor named MasterForm is created for the MasterForm.java.

```
super("Address Book");
```

The text "**Address Book**" is displayed on the title bar of the frame via the super() method.

```
JButton btnAdd = new JButton("Add");
JButton btnEdit = new JButton("Edit");
JButton btnDelete = new JButton("Delete");
JButton btnView = new JButton("View");
```

Four objects of JButton named btnAdd, btnEdit, btnDelete and btnView are created. The text i.e. the name of the button is passed as a parameter to the object.

```
JPanel pnlMenu = new JPanel();
```

Scanned by CamScanner

```
pnlMenu.add(btnAdd);
pnlMenu.add(btnEdit);
pnlMenu.add(btnDelete);
pnlMenu.add(btnView);
```

The four buttons are added to the panel via the add() method of the JPanel object.

```
btnAdd.setMnemonic('a');
btnEdit.setMnemonic('e');
btnDelete.setMnemonic('d');
btnView.setMnemonic('v');
```

The setMnemonic() method is used to set the mnemonic for a button. The setMnemonic() method is only designed to handle character values which fall between 'a' and 'z' or 'A' and 'Z'. A character specifying the mnemonic value is passed as the parameter.

In short, the button can be used by shortcuts i.e. by pressing keyboard keys **Alt + <Mnemonic Value>**.

```
btnAdd.setToolTipText("To add new contacts");
btnEdit.setToolTipText("To edit existing contacts");
btnDelete.setToolTipText("To delete existing contacts");
btnView.setToolTipText("To view existing contacts");
```

The setToolTipText() method registers the text to display in a tool tip. The text is displayed when the cursor lingers over the component. The string to be displayed is passed as the parameter.

```
setContentPane(pnlMenu);
```

An intermediate JPanel object is made part of the content pane by calling the setContentPane() method of the JFrame class.

```
MasterForm frmMaster = new MasterForm();
```

The MasterForm constructor declared is called by creating an object of the same.

Once the .java file is ready, it needs to be compiled by the Java compiler [javac] and then the .class file is interpreted by the Java interpreter [java]. After compiling and executing the window as shown in diagram 28.2 appears.

Scanned by CamScanner

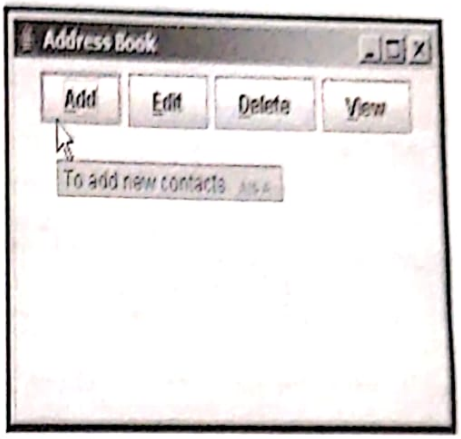


Diagram 28.2: Displaying button along with shortcuts and tool tips

Swing Basic Containers

Four new components have been introduced in Swing to deal with the complexities involved in mixing up the heavyweight and the lightweight components in the same container for use in an application's GUI. These components are the **content pane**, the **layered pane**, the **glass pane** and an **optional menu bar**.

In addition to this, the *root pane* is a virtual container that contains the content pane, the layered pane, the glass pane and if required the optional menu bar.

Root Pane

Swing containers such as JApplet, JFrame, JWindow and JDialog delegate their duties to the root pane represented by the class **JRootPane**. Because the root pane is made up of a content pane, layered pane, glass pane and an optional menu bar, other GUI components must be added to one of these root pane members, rather than to the root pane itself. Diagram 28.3 shows how root pane members are positioned inside the root pane. A reference to the underlying root pane can be achieved by using the method **getRootPane()**.

Scanned by CamScanner

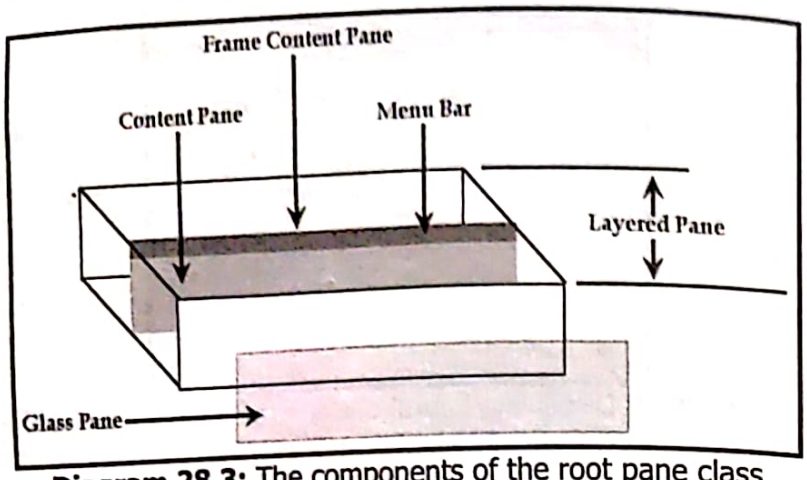


Diagram 28.3: The components of the root pane class

The root pane and its members are all considered fundamental to Swing/GUI design.

REMINDER

A root pane **cannot** have any children. Thus, GUI components **cannot** be added directly to the root pane. Instead, GUI components have to be added to one or mere root pane members.

Content Pane

The content pane actually sits on one of the layers of a specially layered pane. The layered pane contains several layers meant for displaying different Swing components, depending on the GUI's requirements. Diagram 35.3 shows the position of the content pane in the specially layered pane. The menu bar and the content pane sit on the **Frame-Content-Layer**.

Most Swing components are added to the content pane.

REMINDER

By default, the **border layout manager** is bound to the content pane.

Glass Pane

The glass pane is a member of the multi layered, root pane. The glass pane is used to draw over an area that already contains some components. A glass pane can also be used to catch mouse events because it sits over the top of the content pane and menu bar. Diagram 35.3 shows the position of the glass pane in a root pane. Because glass pane is a member of the root pane, it already exists in a Swing container. The glass pane is not visible by default.

Scanned by CamScanner

REMINDER

A glass pane can is made visible by invoking **setVisible(true)** bound to the glass pane object. A reference to a glass pane object is obtained by invoking the method **getGlassPane()** on the container.

Menu Bars

The menu bars can be optionally added to the menu bar area, which is a thin rectangular area along the upper edge of the frame's content pane. The remaining area of the content pane can be filled with other GUI components. Diagram 35.3 shows the location of the menu bar in a special layered pane. To obtain a reference to the optional menu bar that exists within a Swing frame, the method **getJMenuBar()** bound to that the container object can be called.

Layered Pane

A layered pane object is represented by the Swing class **JLayeredPane**, which extends **JComponent**. The class is located within in the **javax.swing** package.

For convenience, JLayeredPane divides a pane's depth range into several different layers. Putting a component into one of these layers ensures that components overlap properly. The layers are:

- DEFAULT_LAYER: The standard, bottommost layer, where most components go
- PALETTE_LAYER: The palette layer sits over the default layer and is useful for floating toolbars and palettes
- MODAL_LAYER: The layer used by modal dialog boxes
- POPUP_LAYER: The pop-up layer displays above the modal dialog layer
- DRAG_LAYER: When a component is dragged, assigning it to a drag layer makes sure it is positioned over all the other components in the container

Buttons

Swing buttons have their classes and methods held within the *JButton* class. A Swing button can feature a text label, an icon label or a combination of both.

The *JButton* class inherits its functionality from the *AbstractButton* class. The *AbstractButton* class defines the inherent behavior of all Swing buttons.

Scanned by CamScanner

Constructors of the *JButton* class include the following:

- **JButton(*String*):** A button with the specified text which will be displayed on the body of the button
- **JButton(*Icon*):** A button will be displayed with the icon specified on its body
- **JButton(*String, Icon*):** A button will be displayed with the specified text and the icon on its body

The following code snippet spawns a button with **Click Me** displayed on the button:

```
JButton btnClick = new JButton("Click Me");
```

Icons can be associated to buttons by creating an object of the *ImageIcon* class, which is similar to instantiating an *Image* object.

The constructor of the *ImageIcon* class takes a graphic filename and/or it's URL as the only argument.

The following table describes some of the methods provided by the *JButton* class:

Methods	Description
boolean isDefaultButton()	Sets the button as the default button in the GUI
void updateUI()	Used to update the look and feel of the application's GUI

Labels

Swing Labels are implemented in the *JLabel* class. Icons can be bound to labels.

Label alignment can be specified using one of the three class constants from within the *SwingConstants* class **LEFT, CENTER** or **RIGHT.**

To create a label, one of the following constructors can be used:

- **JLabel (*String, int*):** Creates a label with the specified text and text alignment
- **JLabel (*String, Icon, int*):** Creates a label with specified text, icon and text alignment

Scanned by CamScanner

The following table describes some of the methods provided by the *JLabel* class:

Methods	Description
void setHorizontalAlignment(int) void setVerticalAlignment(int) int getHorizontalAlignment() int getVerticalAlignment()	The set or get specifies where in the label its contents should be placed. The SwingContantsinterface defines five possible values for horizontal alignment: LEFT [default for text only labels], CENTER [default for image only labels], RIGHT, LEADING and TRAILING. For vertical alignment: TOP, CENTER [default] and BOTTOM
void setIconTextGap(int) int getIconTextGap()	Sets or gets the number of pixels between the label's text and its image
icon getIcon()	Returns an icon associated with the image
void setIcon(Icon)	Sets an icon to the label
string getText()	Returns the text string that the label displays
void setText(String)	Defines a single line of text the label will display
void setDisplayedKeyAccelerator(char)	Specifies a character that indicates the shortcut key
char getDisplayedKeyAccelerator()	Returns the char that indicates the shortcut key

The following code snippet displays an image and a string in a Label on a Frame:

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

public class Login extends JPanel
{
   public Login()
   {
        Icon imgIcon = new ImageIcon("Images/security.jpg");
        JLabel lblInValid = new JLabel("The Login username or password is not valid.",
                                imgIcon, SwingConstants.RIGHT);
        add(lblInValid);
        JLabel lblReEnter = new JLabel("Please enter correct username and passord.",
                                SwingConstants.RIGHT);
        add(lblReEnter);
   }

   public static void main(String args[])
   {
        JFrame frm = new JFrame("Invalid Login");
        Login lblLogin = new Login();
        frm.getContentPane().add("Center", lblLogin);
        frm.setSize(400, 200);
        frm.setVisible(true);
```

Scanned by CamScanner

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

Explanation:

```
Icon imgIcon = new ImageIcon("Images/security.jpg");
```

An object named imgIcon creates an ImageIcon from the specified URL. The image will be preloaded by using MediaTracker to monitor the loaded state of the image. The icon's description is initialized to be a string representation of the URL.

```
JLabel lblInValid = new JLabel("The Login username or password is not valid.", imgIcon,
                    SwingConstants.RIGHT);
```

An object of JLabel is created. Three parameters are passed viz. the text to be displayed on the frame, the path of the image and the alignment of the text.

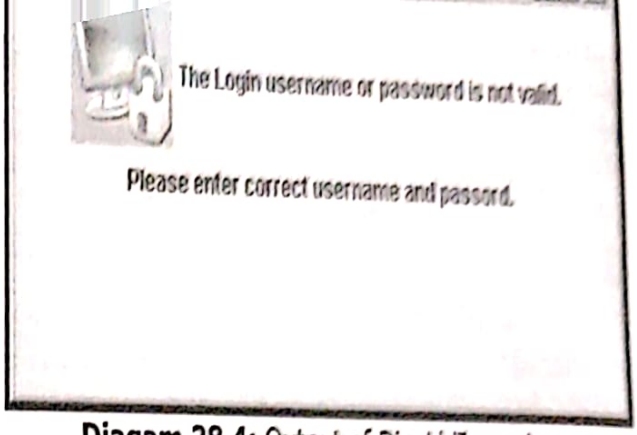
For the alignment of the text SwingConstants interface is used. SwingConstants is a collection of constants, which is generally used for positioning and orienting components on the screen.

```
Login lblLogin = new Login();
frm.getContentPane().add("Center", lblLogin);
```

The Login constructor declared is called by creating an object of the same. The getContentPane() method returns the contentPane object for this particular frame. The add() method adds the Login constructor at the center of the frame via the getContentPane() method.

Once the .java file is ready, it needs to be compiled by the Java compiler and then the .class file is interpreted by the Java interpreter. After compiling and executing the window as shown in diagram 28.4 appears.

Scanned by CamScanner



Diagam 28.4: Output of DispLblIcon.class

Text Fields

A text field is a component that lets a user enter a small amount of text, normally used with GUI based data entry screens in commercial application or Applets.

Swing Text fields are implemented in the *JTextField* class.

JTextField class constructors include the following:

- **JTextField(*int*):** A text field with the specified width
- **JTextField(*String, int*):** A text field which dislay the specified text and has the width specified

The following example creates a text field having a width of 10 characters:

```
JTextField txt = new JTextField(10);
```

The *JPasswordField* class creates a text field that uses a special character to obscure input. The *JPasswordField* class is a subclass of *JTextField* class.

This class has constructor methods similar to the constructors of the *JTextField* class:

- **JPasswordField(*int*):** The *int* argument specifies the desired width of the field
- **JPasswordField(*String, int*):** The string argument contains the fields initial text and the *int* argument specifies the desired width of the field

Once a password text field is created, *setEchoChar(char)* method can be used to obscure input with the character specified in *char*.

Scanned by CamScanner

The following table

Methods	Description
void setEditable(boolean) boolean getEditable()	Sets or gets whether the user can edit the text in the text field
void setColumns(int) int getColumns()	Sets or gets the number of columns displayed by the text field
int getColumnWidth()	Get the width of the text field's columns
void setHorizontalAlignment(int) int getHorizontalAlignment()	Set or get how the text is aligned horizontally within its area. One can use JTextField.LEFT, JTextField.CENTER and JTextField.RIGHT for aligning text in the text field
void setEchoChar(char) char getEchoChar() (in JPasswordField)	Set or get the echo character - the character displayed instead of the actual characters typed by the user

The following code snippet creates an object of the *JPasswordField* class and sets *Hash* as the character that obscures input:

```
JPasswordField passwdtxt = new JPasswordField(10);
passwdtxt.setEchoChar('#');
```

Example:

The following code snippet displays a Login form with one text field and one Password field.

Code spec for Login.java:

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

public class Login extends JPanel
{
/* Creating an object of the JLabel class */
  JLabel lblUserName;
  JLabel lblPassword;
/* Creating an object of the JPassword class */
  JTextField txtUserName;
  JPasswordField txtPassword;

  public Login()
  {
      lblUserName = new JLabel("Username: ");
      txtUserName = new JTextField(10);
      lblPassword = new JLabel("Password: ");
      txtPassword = new JPasswordField(10);
      txtPassword.setEchoChar('*');
```

Scanned by CamScanner

```
/* Adding tooltips to the text fields */
    txtUserName.setToolTipText("Enter Username");
    txtPassword.setToolTipText("Enter Password");
/* Adding labels to the Panel */
    add(lblUserName);
    add(txtUserName);
    add(lblPassword);
    add(txtPassword);
}

public static void main(String[] args)
{
/* Calling the PassDemo constructor */
    Login frmLogin = new Login();
/* Setting the text on the frame */
    JFrame frm = new JFrame("Login Form");
    frm.setContentPane(frmLogin);
    frm.setSize(200, 150);
    frm.setVisible(true);
    WindowListener listener = new WindowAdapter()
    {
        public void windowClosing(WindowEvent winEvt)
        {
            System.exit(0);
        }
    };
    frm.addWindowListener(listener);
  }
}
```

Explanation:

```
JTextField txtUserName;
JPasswordField txtPassword;
txtUserName = new JTextField(10);
txtPassword = new JPasswordField(10);
```

An object named txtUserName of JTextField is created and an object named txtPassword of JPasswordField is created. Each object is passed the size in characters as the parameter.

```
txtPassword.setEchoChar('*');
```

The setEchoChar() method of the JPasswordField class is used to create a text box, which can accept passwords from the users.

```
add(txtUserName);
add(txtPassword);
```

The JTextField and the JPasswordField object is added to the frame via the add() method.

Scanned by CamScanner

Once the .java file is ready, it needs to be compiled by the Java compiler and then the .class file is interpreted by the Java interpreter. After compiling and executing the window as shown in diagram 28.5 appears.

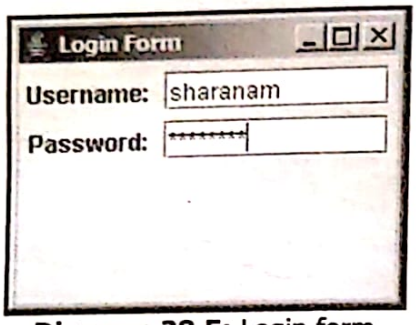


Diagram 28.5: Login form

Text Areas

A text area is a text control used in a GUI that lets a user enter multiple lines of text.

Swing Text areas are implemented in the JTextArea class. JTextArea takes the following constructor methods:

- **JTextArea(int, int):** A text area with the specified number of rows and columns
- **JTextArea(String, int, int):** A text area which displays the specified text and has the specified rows and columns

The following code snippet creates a text area object of 10 rows and 20 columns.

```
JTextArea taRemarks = new JTextArea(10, 20);
```

The following table describes some of the methods provided by the *JTextArea* class:

Methods	Description
Append(String)	Appends the given string to the end of the document
void getColumns(int) int getRows(int)	Returns the number of columns and rows in the text area
insert(String, int)	Inserts a specified text at the specified position in the text area
void setRows(int)	Sets the number of rows for the text area
void setFont(Font)	Sets a specific font for the text

Example:
The following example creates an Address Details form where in the First name, Last name and the Address of an individual is captured using JSwing components.

Scanned by CamScanner

Code spec for AddressDetails.java:

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

public class AddressDetails extends JPanel
{
   JLabel lblFirstName, lblLastName, lblAddress;
   JTextField txtFirstName, txtLastName;
   JTextArea txtareaAddress;

   public AddressDetails()
   {
   /* Setting the layout to none */
      setLayout(null);

      lblFirstName = new JLabel("First Name: ");
      lblLastName = new JLabel("Last Name: ");
      lblAddress = new JLabel("Address: ");
      txtFirstName = new JTextField(15);
      txtLastName = new JTextField(15);
      txtareaAddress = new JTextArea(10, 10);

   /* Using setBounds() method to specify the positioning of the
   objects by setting for row, column, width and height */
      lblFirstName.setBounds(10, 20, 120, 25);
      txtFirstName.setBounds(150, 20, 210, 25);
      lblLastName.setBounds(10, 60, 180, 25);
      txtLastName.setBounds(150, 60, 210, 25);
      lblAddress.setBounds(10, 100, 250, 25);
      txtareaAddress.setBounds(150, 100, 280, 100);

      add(lblFirstName);
      add(txtFirstName);
      add(lblLastName);
      add(txtLastName);
      add(lblAddress);
      add(txtareaAddress);
   }

   public static void main(String[] args)
   {
      AddressDetails frmAddress = new AddressDetails();
      JFrame frm = new JFrame("Address Details");
      frm.setContentPane(frmAddress);
      frm.setSize(500, 300);
      frm.setVisible(true);
```

Scanned by CamScanner

```
        WindowListener listener = new WindowAdapter()
        {
            public void windowClosing(WindowEvent winEvt)
            {
                System.exit(0);
            }
        };
        frm.addWindowListener(listener);
    }
}
```

Explanation:

```
JTextArea txtareaAddress;
txtareaAddress = new JTextArea(10, 10);
```

An object named txtareaAddress of JTextArea is created. The size of height and width of the text area is passed as the parameter.

```
setLayout(null);
```

The setLayout() method sets the layout of the frame. Here, the layout is set to none.

```
lblFirstName.setBounds(10, 20, 120, 25);
txtFirstName.setBounds(150, 20, 210, 25);
lblLastName.setBounds(10, 60, 180, 25);
txtLastName.setBounds(150, 60, 210, 25);
lblAddress.setBounds(10, 100, 250, 25);
txtareaAddress.setBounds(150, 100, 280, 100);
```

The labels, text fields and the text area are positioned using the setBounds() method. The setBounds() method is passed four parameters viz. the row, the column, the width and the height.

```
add(txtareaAddress);
```

The JTextArea object is added to the frame via the add() method.

Once the .java file is ready, it needs to be compiled by the Java compiler and then the .class file is interpreted by the Java interpreter. After compiling and executing the window as shown in diagram 28.6 appears.

Scanned by CamScanner

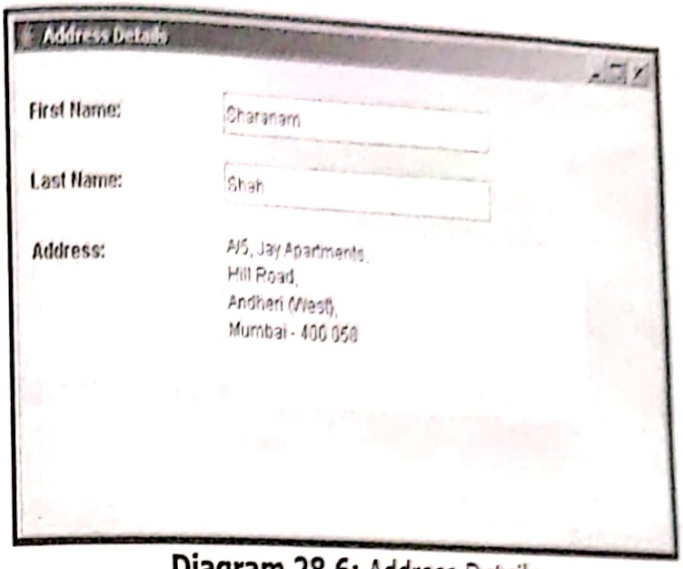


Diagram 28.6: Address Details

Check Boxes

The Swing check box is implemented in the *JCheckBox* class. Swing checkboxes can have icon displayed in their labels.

Swing check boxes can be constructed using the following constructor methods:

- **JCheckBox(*String*):** A check box which displays the specified text as its label
- **JCheckBox(*String, boolean*):** A check box which displays the specified text as its label and displays as ticked if the second argument is TRUE
- **JCheckBox(*Icon*):** A check box which displays the specified icon as its label
- **JCheckBox(*Icon, boolean*):** A check box which displays the specified icon as its label and displays as ticked if the second argument is TRUE
- **JCheckBox(*String, Icon*):** A check box with the specified text and the icon specified as well as its label
- **JCheckBox(*String, Icon, boolean*):** A check box which displays the specified text and icon as its label and is ticked if the second argument is TRUE

The following table describes some of the methods provided by the *JCheckBox* class

Methods	Description
String getLabel() void setLabel(String)	Sets or gets a label for a check box
boolean getState() void setState(boolean)	Sets or gets the status of the check box

Scanned by CamScanner

...pet creates two checkboxes one with a text as its label and the other ... a text and an icon as its label:

```
JCheckBox chk, chkIcon;
ImageIcon img = new ImageIcon("Img0001.gif");
chk = new JCheckBox("Check One");
chkIcon = new JCheckBox("Check Two", img);
```

Example:

The following code snippet displays a list of hobbies in the form of Checkboxes and Music is tick marked.

Code spec for Hobbies.java:

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

public class Hobbies extends JPanel
{
   JCheckBox chkMusic, chkDancing, chkReading, chkStamp, chkCoins, chkDriving;

   public Hobbies()
   {
        chkMusic = new JCheckBox("Music", true);
        chkDancing = new JCheckBox("Dancing", false);
        chkReading = new JCheckBox("Reading", false);
        chkDriving = new JCheckBox("Driving", false);
        chkStamp = new JCheckBox("Collecting Stamps", false);
        chkCoins = new JCheckBox("Collecting Old Coins", false);

        add(chkMusic);
        add(chkDancing);
        add(chkReading);
        add(chkDriving);
        add(chkStamp);
        add(chkCoins);
   }

   public static void main(String[] args)
   {
        Hobbies frmHobbies = new Hobbies();
        JFrame frm = new JFrame("Hobbies");
        frm.setContentPane(frmHobbies);
        frm.setSize(300, 150);
        frm.setVisible(true);
        WindowListener listener = new WindowAdapter()
        {
             public void windowClosing(WindowEvent winEvt)
```

Scanned by CamScanner

```
            {
                System.exit(0);
            }
        };
        frm.addWindowListener(listener);
    }
}
```

Explanation:

```
JCheckBox chkMusic, chkDancing, chkReading, chkStamp, chkCoins, chkDriving;
```

Six objects named chkMusic, chkReading, chkStamp, chkCoins and chkDriving of JCheckBox are created.

```
chkMusic = new JCheckBox("Music", true);
chkDancing = new JCheckBox("Dancing", false);
chkReading = new JCheckBox("Reading", false);
chkDriving = new JCheckBox("Driving", false);
chkStamp = new JCheckBox("Collecting Stamps", false);
chkCoins = new JCheckBox("Collecting Old Coins", false);
```

Each object of JCheckBox is passed the text to be displayed next to checkbox and whether the checkbox is marked or not as the parameters.

```
add(chkMusic);
add(chkDancing);
add(chkReading);
add(chkDriving);
add(chkStamp);
add(chkCoins);
```

The JCheckBox objects are added to the frame via the add() method.

Once the .java file is ready, it needs to be compiled by the Java compiler and then the .class file is interpreted by the Java interpreter. After compiling and executing the window as shown in diagram 28.7 appears.

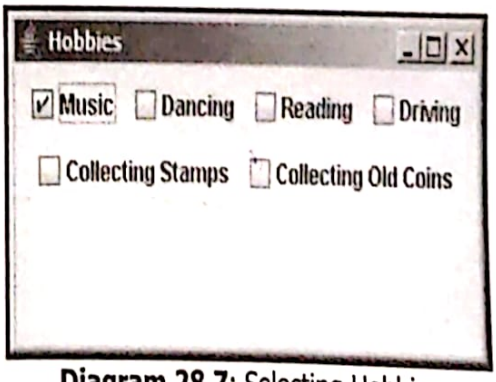


Diagram 28.7: Selecting Hobbies

Scanned by CamScanner

Radio Buttons

Swing Radio buttons are implemented in the *JRadioButton* class. The constructor methods are the same as those for the *JCheckBox* class.

The following code snippet creates two radio buttons one with text as its label and the other with text and an icon displayed on its label:

```
JRadioButton radio1, radioIcon1;
ImageIcon imgIcon = new ImageIcon("Img0001.gif");
rd = new JRadioButton("Radio One");
rdIcon = new JRadioButton("Radio Two", imgIcon);
```

The following table describes some of the methods provided by the *JRadioButton* class

Methods	Description
String getLabel() void setLabel(String)	Sets or gets a label for a radio button
boolean getState() void setState(boolean)	Sets or gets the status of the radio button

Example:
The following code snippet displays three radio buttons. The radio buttons are created so that only **one** can be selected out of the three.

Code spec for Signs.java:

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

public class Signs extends JPanel
{
  static JFrame frm;
  public Signs()
  {
  /* Creating the radio buttons */
      JRadioButton rbtnNumbers = new JRadioButton("Numbers", true);
      JRadioButton rbtnAlphabets = new JRadioButton("Alphabets");
      JRadioButton rbtnSymbols = new JRadioButton("Symbols");
  /* Grouping the radio buttons */
      ButtonGroup btnGroup = new ButtonGroup();
      btnGroup.add(rbtnNumbers);
      btnGroup.add(rbtnAlphabets);
      btnGroup.add(rbtnSymbols);
```

Scanned by CamScanner

```
/* Putting the radio buttons in a column in a panel */
    JPanel paneRadio = new JPanel();
    paneRadio.setLayout(new GridLayout(0, 1));
    paneRadio.add(rbtnNumbers);
    paneRadio.add(rbtnAlphabets);
    paneRadio.add(rbtnSymbols);

    setLayout(new BorderLayout());
    add(paneRadio);
    setBorder(BorderFactory.createEmptyBorder(20, 20, 20, 20));
}

public static void main(String args[])
{
    frm = new JFrame("Select Signs");
    frm.addWindowListener(new WindowAdapter()
    {
        public void windowClosing(WindowEvent e)
        {
            System.exit(0);
        }
    });
    frm.getContentPane().add(new Signs());
    frm.setSize(200, 200);
    frm.setVisible(true);
}
}
```

Explanation:

```
JRadioButton rbtnNumbers = new JRadioButton("Numbers", true);
JRadioButton rbtnAlphabets = new JRadioButton("Alphabets");
JRadioButton rbtnSymbols = new JRadioButton("Symbols");
```

Three objects named rbtnNumbers, rbtnAlphabets and rbtnSymbols of JRadioButton are created. Each object is passed the text to be displayed next to the radio button. The rbtnNumbers object is also passed the Boolean value, which indicates that the radio button is marked by default.

```
ButtonGroup btnGroup = new ButtonGroup();
btnGroup.add(rbtnNumbers);
btnGroup.add(rbtnAlphabets);
btnGroup.add(rbtnSymbols);
```

An object named btnGroup of ButtonGroup class is created. Then the three radio button objects created are added to this ButtonGroup class via the add() method.

```
JPanel paneRadio = new JPanel();
paneRadio.setLayout(new GridLayout(0, 1));
```

Scanned by CamScanner

```
paneRadio.add(rbtnNumbers);
paneRadio.add(rbtnAlphabets);
paneRadio.add(rbtnSymbols);
```

An object named paneRadio of JPanel class is created. The GridLayout layout of the panel is set via the setLayout() method. Then the three radio button objects are added to the panel via the add() method.

```
add(paneRadio);
setBorder(BorderFactory.createEmptyBorder(20, 20, 20, 20));
```

The panel is added to the frame by the add() method. The border of the frame is set using the setBorder() method. The setBorder() method is passed the BorderFactory class's createEmptyBorder() method.

The BorderFactory class returns objects that implement the Border interface.

The createEmptyBorder() method creates an empty border that takes up space but which does no drawing, specifying the width of the top, left, bottom and right sides.

Once the .java file is ready, it needs to be compiled by the Java compiler and then the .class file is interpreted by the Java interpreter. After compiling and executing the window as shown in diagram 28.8 appears.

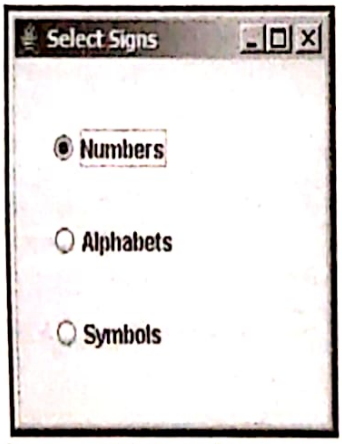


Diagram 28.8: Selecting Signs

JApplet

Swing introduces its own version of Applet, called JApplet whose parent is the AWT Applet. Swing Applets are more sophisticated, they can support menu bars and layering of components, allowing painting over components already laid inside the Applet.

Scanned by CamScanner

...ing Applets can display a menu bar. This requires the layering of components, which is essentially placing components in multiple layers which can overlap each other. This allows components to be positioned on specific layers. Swing Applets contain a special component called a *layered pane* to achieve this.

The feature that allows painting over components inside the Applet is accomplished by using JApplet component called a *glass pane*. The glass pane is a transparent component that allows the background components to become visible.

A Swing Applet is represented by the **JApplet** class, which is in the package **javax.swing**. A Swing applet can be created by instantiating a class that extends **JApplet**. Just like JFrame in JApplet components have to be added to the JApplet content pane first. The Swing library has his intermediate container which deals with the complexities of mixing up lightweight and the heavyweight components in an Applet. The following code snippet shows how a component can be added to the content pane inside a Swing Applet:

```
/* Retrieving a handle on the applet's content pane */
  Container contentPane = this.getContentPane();
/* Assigning a new layout */
  contenPane.setLayout(<SomeLayoutObject>);
/* Adding the component to the applet */
  contentPane.add(<SomeComponent>);
```

References to the Applet's layered pane, glass pane and menu bar can be obtained by calling the following methods of the Applet's instance:

```
public void getLayeredPane();
public void getGlassPane();
public void getJmenuBar();
```

Example:

The following is a code snippet, which implements a Swing Applet. It displays the text "Welcome To The www.sharanamshah.com website".

Code spec for Welcome.java:

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

public class Welcome extends JApplet
{
  public void init()
  {
  /* Retrieving a reference to the content pane of the applet */
     Container content = getContentPane();
```

Scanned by CamScanner

```
        JLabel lbl = new JLabel ("Welcome To The www.sharanamshah.com website",
                    JLabel.CENTER);
    /* Adding the label to the applet's content pane */
        content.add(lbl);
    }
}
```

Explanation:

The class Welcome extends the JApplet class. JApplet inherits from the Applet. The JApplet class allows the Java programs to run in a browser. In other words it can be said that JApplet is the Swing version of Applet and it behaves the same.

The init() method creates an Applet's GUI and executes a job on event-dispatching thread.

Code spec of index.jsp:

```
<%@ page language="java" contentType="text/html" import="java.lang.*" %>
<HTML>
  <HEAD>
      <TITLE>Welcome To The World Of Swing</TITLE>
  </HEAD>
  <BODY>
      <jsp:plugin type="applet" code="Welcome.class" codebase="/MyWebApplication/"
                  width="300" height="200">
          <jsp:fallback>Unable To load Applet</jsp:fallback>
      </jsp:plugin>
  </BODY>
</HTML>
```

Make a war file and deploy the application. After deploying the file, move the class file to the root directory of the web application as explained earlier.

Next, run the JSP file in the browser as shown in diagram 28.9.

Scanned by CamScanner

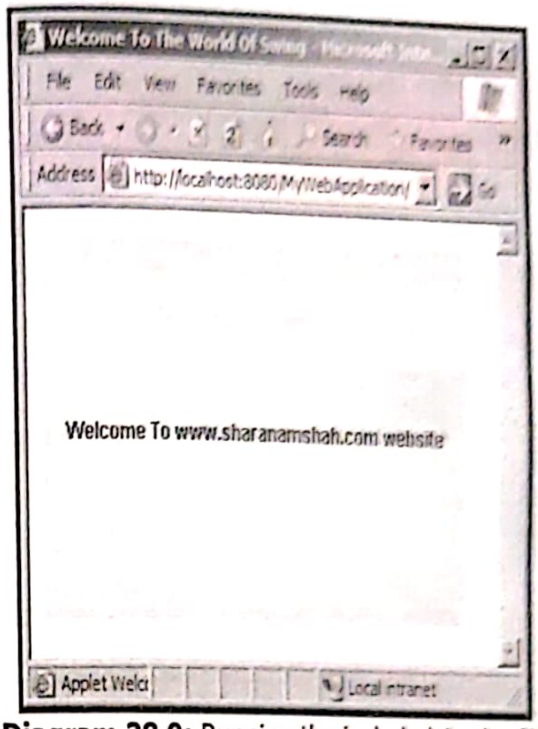


Diagram 28.9: Running the included Applet file

Combo box

Swing Choice lists, are implemented using the *JComboBox* class. The *JComboBox* class is similar to the *Choice* class in the AWT.

The following shows how a Swing choice list is created:

- A *JComboBox()* object is instantiated with no arguments
- The combo box's *addItem(Object)* method is used to add items to its list
- The combo box's *setEditable(boolean)* method is used with FALSE as its argument

The *setEditable()* method allows the entry of another value if its parameter is set to TRUE apart from the list of choices being displayed.

If *setEditable()* is set to FALSE, the combo box is converted into a choice list. The only choice a user can make are those items which are in the list.

If *setEditable()* is set to TRUE, the user can enter text into the combo box instead of using the choice list to pick an item. This is the combination that gives combo boxes their name.

Scanned by CamScanner

The following code snippet creates a combo box with two items:

```
JComboBox cmb;
cmb = new JComboBox();
cmb.addItem("Choice1");
cmb.addItem("Choice2");
add(cmb);
```

The following table describes some of the methods provided by the *JComboBox* class

Methods	Description
void addItem(Object) void insertItemAt(Object, int)	Add or insert the specified object into the combo box's menu. The insert method places the specified object at the specified index, thus inserting it before the object currently at that index
Boolean getState() void setState(boolean)	Sets or gets the number the status of the combo box
Object getItemAt(int) Object getSelectedItem()	Get an item from the combo box's menu from specific position. getSelectedItem() method returns an array containing the selected items
void removeAllItems() void removeItemAt(int)	Remove one or more items from the combo box's menu
int getItemCount()	Get the number of items in the combo box' menu
selectedItemChanged()	This method is called when the selected item is changed

Example:
The following is the code snippet which displays a list of hobbies in a combo box.

Code spec for Hobbies.java:

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

public class Hobbies extends JApplet
{
   public void init()
   {
        Container content = getContentPane();
        content.setLayout(new FlowLayout());
        JLabel lblHobbies = new JLabel("Hobbies: ");
        add(lblHobbies);
   /* Creating a combo box and adding it to the panel */
        JComboBox cmbHobbies = new JComboBox();
        cmbHobbies.addItem("Driving");
        cmbHobbies.addItem("Collecting Stamps");
```

Scanned by CamScanner

```
        cmbHobbies.addItem("Collecting Old Coins");
        cmbHobbies.addItem("Music");
        cmbHobbies.addItem("Dancing");
        content.add(cmbHobbies);
    }
}
```

Explanation:

```
JComboBox cmbHobbies = new JComboBox();
```

An object named cmbHobbies of JComboBox is created.

```
cmbHobbies.addItem("Driving");
cmbHobbies.addItem("Collecting Stamps");
cmbHobbies.addItem("Collecting Old Coins");
cmbHobbies.addItem("Music");
cmbHobbies.addItem("Dancing");
```

Items are added to the combo box via the addItem() method. These items are later displayed in the combo box i.e. when the application is run.

```
content.add(cmbHobbies);
```

The combo box object is added to the content pane by using the add() method.

Code spec of index.jsp:

```
<%@ page language="java" contentType="text/html" import="java.lang.*" %>
<HTML>
  <HEAD>
      <TITLE>Welcome To The World Of Swing</TITLE>
  </HEAD>
  <BODY>
      <jsp:plugin type="applet" code="Hobbies.class" codebase="/MyWebApplication/"
                  width="300" height="200">
          <jsp:fallback>Unable To load Applet</jsp:fallback>
      </jsp:plugin>
  </BODY>
</HTML>
```

Make a war file and deploy the application. After deploying the file, move the class file to the root directory of the web application as explained earlier.

Next, run the JSP file in the browser as shown in diagram 28.10.

Scanned by CamScanner

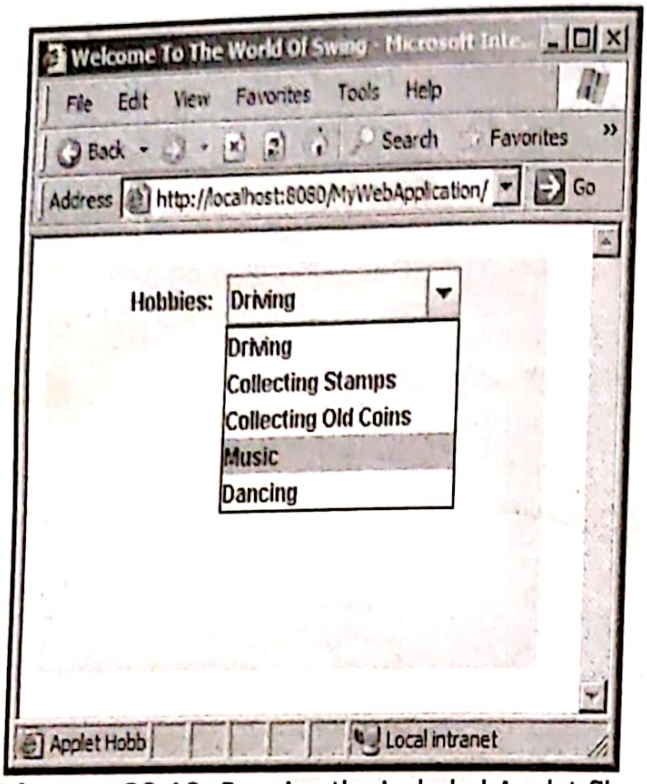


Diagram 28.10: Running the included Applet file

Event Handling

Every time the user types a character in a text field or text area or pushes a mouse button on a GUI, an event occurs. Any object [in this case the GUI] can be notified that an event occured. All the object has to do is implement an appropriate interface that is registered as its event listener bound to an appropriate event source.

Each event that occurs is represented by an object [i.e. spawns an object], which provides information about the event and identifies the event source. Many Event sources are components; however, other kinds of objects can also be event sources.

Implementing An Event Handler

Every event handler requires three bits of code:

- When the event handler class is declared, code spec specifies that the class either implements a listener interface or extends a class that implements a listener interface

  Example:

```
public class MyClass implements ActionListener
{
/* Code spec to be implemented */
}
```

Scanned by CamScanner

- Code spec that registers [binds] the instance of the event handler as a listener with one or more components

  Example:

```
<someComponent>.addActionListener(<instanceOfMyClass>);
```

- Code spec that implements method signatures within the event handler

  Example:

```
public void actionPerformed(ActionEvent actEvt)
{
  ...
/* Code spec that reacts to the action */
  ...
}
```

Event Class	Event Method	Listener Class	Listener Method
ActionEvent [for buttons]	public Object getSource()	ActionListener	public void actionPerformed(ActionEvent)
AdjustmentEvent [for scrollbars]	public int getValue() public Adjustment getAdjustable() public int getAdjustmentType()	AdjustmentListener	public void adjustmentValueChanged (AdjustmentEvent)
FocusEvent [When the component gains or losses focus]	public Boolean isTemporary()	FocusListener	public void focusGained(FocusEvent) public void focusLost(FocusEvent)
ItemEvent [for choice list, check boxes, List]	public void getItem() public ItemSelectable getItemSelectable() public int getStateChange()	ItemListener	public void itemStateChanged (ItemEvent)
KeyEvent [for trapping keyboard events]	public char getKeyChar() public int getKeyCode() public boolean isActionKey()	KeyListener	public void keyPressed(KeyEvent) public void keyReleased(KeyEvent) public void keyTyped(KeyEvent)

Scanned by CamScanner

Event Class	Event Method	Listener Class	Listener Method
MouseEvent [for trapping mouse events]	public int getClickCount() public Point getPoint() public int getX() public int getY()	MouseListener	public void mouseClicked (MouseEvent) public void mouseEntered (MouseEvent) public void mousePressed (MouseEvent) public void mouseExited (MouseEvent) public void mouseClicked (MouseEvent) public void mouseReleased (MouseEvent)

Apart from the above listener classes, Java Swing has additional listener classes. Refer to the **Help** provided with the JDK which reveal these classes.

Caret Listener

Caret events occur when the cursor in a text component moves or when selected text in a text component is changed. A caret listener can be attached to any instance of any *JTextComponent* subclass using the *addCaretListener*() method.

If the program has a custom caret, it may be more convenient to attach a listener to the caret object itself rather than to the text component for which it behaves as a caret. A caret fires **change events**, a change listener is required and not caret listener.

The following is a caret event handling code snippet from an application named TextComponentDemo:

```
...
/* Where initialization occurs */
  CaretListenerLabel crtListenerLbl = new CaretListenerLabel("Caret Status");
...
  textPane.addActionListener(crtListenerLbl);
...
  protected class CaretListenerLabel extends JLabel implements CaretListener
  {
      ...
      public void caretUpdate(CaretEvent crtEvt)
      {
      /* Retrieving the location in the text */
          int dot = crtEvt.getDot();
```

Scanned by CamScanner

```
        int mark = crtEvt.getMark();
        . . .
    }
}
```

The methods in the *CaretEvent* class are as follows:

- void caretUpdate(CaretEvent): This method gets called when the caret, which is within the listened to component moves **or** when the text selected in the listened to component changes
- int getDot(): This method returns the current location of the caret. If text is selected, the caret marks the start of a selection
- int getMark(): This method returns the end of the same selection. If nothing is selected, the value returned by this method is equal to the value returned by **getDot()**. Note that the value returned by getDot() is not guaranteed to be less than getMark()

Change Listeners

In Swing the *ChangeEvent class* represents slider movement. This class is part of the **javax.swing.event package.**

In order to register component change events, a listener must be added to the component by using it's *addChangeListener(Object)* method. The argument to this method is the name of the object that will handle the change event. The keyword *this* can be used if the same object is being used to send the event and to receive it.

The object [i.e. class] that receives a *ChangeEvent* must implement the *ChangeListener interface.* This interface includes only one method, *stateChanged(ChangeEvent)* which takes the following format:

```
public void stateChanged(ChangeEvent chngEvt)
{
/* Method code */
  ...
}
```

The event listener's *getSource()* method can be used to find out which component generated the change event.

Change events in sliders are generated very rapidly when the user drags a slider in one direction or another. Change events are generated throughout the movement, while the slider is being dragged.

The listener's *getValueIsAdjusting()* method can be used to test whether the slider is still being moved. This returns TRUE while the slider is moving and FALSE otherwise.

Scanned by CamScanner

Using Dialogs

A dialog box is a window that is displayed within the context of another window which is its parent. Dialog boxes are used to manage input that cannot be handled conveniently, selecting from a range of options for instance or enabling data to be entered from the keyboard.

Dialogs can also be used to display information/messages or warnings to a user. Swing dialogs are defined within the JDialog class in the **javax.swing** package. A JDialog object is a specialized sort of a Window. A JDialog object typically contains one or more components for displaying information **or** allowing data to be entered by the user **plus** buttons for selection of dialog options [including closing the dialog]. However, for many of the typical dialogs, the JOptionPane class also provides an easy shortcut to creating dialogs.

Modal And Non-Modal Dialogs

There are two different kinds of dialog boxes i.e. **Modal Dialog** and **Non-Modal Dialog**

Modal Dialog

When a modal dialog is displayed in an application it takes control of application and prevents any background processing in the application until the modal dialog is closed. No operation of the application can continue until the modal dialog has its **OK** button clicked.

Modal dialogs that require user input normally have at least two buttons, an **OK** button that is used to accept whatever input has being entered [and then close the dialog] **and** a **CANCEL** button to that closes the dialog and aborts. Dialogs that manage input are almost always modal dialogs, simply because no other application interaction must be permitted until user input is complete.

Non-Modal Dialog

A non-modal dialog box can be left on the screen as long as needed, since it does not block any user interaction with other windows within the application's work flow.

Whether a modal or non-modal dialog is created, is determined an argument passed to JDialog() the dialog class constructor. The constructor will create a non-modal dialog, unless instructed otherwise.

Scanned by CamScanner

There is a choice of five constructors for a **JDialog** object:

Contructor	Description Title Bar	Mode
JDialog ()	(empty)	Non-modal
JDialog (Frame parent)	(empty)	Non-modal
JDialog (Frame parent, String title)	Title	Non-modal
JDialog (Frame parent, boolean modal)	(empty)	Modal [when modal argument is TRUE] Non-modal [when modal argument is FALSE]
JDialog (Frame parent, String title, boolean modal)	Title	Modal [when modal argument is TRUE] Non-modal [when modal argument is FALSE]

After the JDialog object is created using its constructor, the kind of dialog it is can be changed by using the **setModal()** method of the object. If the argument is specified as TRUE, then the dialog will behave athough it is modal, if the argument is specified as FALSE the dialog will behave as though it was non-modal. The **isModal()** method can be used to check whether the dialog is modal or not. isModal() will return TRUE is the dialog is modal or FALSE if its not.

Deriving a class from JDialog gives complete functionality as to how the dialog works, but there is still an easier way to do it. Fortunately, the Swing library contains an intermediate class called *JOptionPane*, whose static methods enable the creation of option panes [standard dialog boxes] directly with just few lines of code.

Option panes can be used to display a feedback message or confirmation or to permit a user to provide input to an application.

The JOptionPane Class

JOptionPane is a convenience class that delivers the functionality of creating option panes that are modal. JOptionPane is a direct subclass of JComponent and belongs to the package javax.swing.

JOptionPane static methods are used to create and display option panes type of a JDialog box i.e. dialog boxes that ask a question, warn a user or provide a brief, important message. When a system crashes, a dialog box appears and breaks the bad news, on deleting files a dialog box appears to make sure whether the user really wants to delete the file or not.

Dialog boxes are an effective way of communicating with a user without the overhead of creating a new class to represent a window, adding components to it, writing event handling

Scanned by CamScanner

methods to take user input. All of these issues are handled automatically when one of the standard modal dialog boxes offered by *JOptionsPane* class is used.

There are four standard modal dialog boxes:

- **ConfirmDialog:** Asks a question with buttons for Yes, No and Cancel responses
- **InputDialog:** Prompts user for text input with buttons for Ok and Cancel
- **MessageDialog:** Displays a message with a button for Ok
- **OptionDialog:** Comprises all three of the other dialog box types

Each of these Dialogs has their own constructor in the *JOptionPane* class.

Confirm Dialog Boxes

The easiest way to create a *Yes/No/Cancel dia*log box is to use the **showConfirmDialog(Component, Object)** method call.

The *Component* argument specifies which container should be considered the parent of the dialog box. This information is used to determine where on the screen the dialog will be displayed. If *null* is used instead of a container **or** if the container is not a *Frame* object, then the dialog box will be centered on the screen.

The second argument, *Object*, can be a string, a component **or** an Icon Object. If it is a string, then that text will be displayed in the dialog box. If it is a component or an icon, then that object will be displayed in place of a text message.

This method returns one of three possible integer values, each is a class constant of *JOptionPane* viz. YES_OPTION, NO_OPTION and CANCEL_OPTION.

The following code snippet displays a confirm dialog with a text message displayed in it and stores user response in the variable *response*:

```
import javax.swing.*;

public class MyConfirm extends JPanel
{
  int response;

  public MyConfirm()
  {
```

Scanned by CamScanner

```
        response = JOptionPane.showConfirmDialog(null, "Should I Delete all your Personal
                                                Files");
    /* Determine the option selected using the integer constants in
    the JOptionPane class */
        if (response = = JOptionPane.YES_OPTION)
            System.out.println("Yes Clicked");
    }
}
```

These are the other options of a confirm dialog:

```
showConfirmDialog(Component, Object, String, int, int)
```

The first two arguments have the same functionality as those in other *showConfirmDialog()*. The last three arguments are as follows:

- The **String** passed will be displayed in the dialog box's title bar
- The first **int** passed is actually a class constant that determines the number of buttons displayed in the Dialog. **YES_NO_CANCEL_OPTION** constant displays the YES, NO and CANCEL buttons. The **YES_NO_OPTION** constant displays YES and NO buttons
- The second **int** passed is also a class constant such as ERROR_MESSAGE, INFORMATION_MESSAGE, PLAIN_MESSAGE, QUESTION_MESSAGE or WARNING_MESSAGE. This argument determines which icon is displayed in the dialog box adjacent to its message

Example:

```
int response = JOptionPane.showConfirmDialog(null,"Error reading file. Want to try again?",
                "File Input Error", JOptionPane.YES_NO_OPTION,
                JOptionPane.ERROR_MESSAGE);
```

Message Dialog Boxes

A message dialog box is a Dialog that displays information to a user.

This can be spawned by calling *showMessageDialog(Component, Object)* method. As with other dialog boxes the arguments indicate the parent object of which this message dialog is a child, component the second argument can be an Object or a String component to be displayed or an icon to display.

Unlike the other dialog boxes, message dialog boxes do not return any kind of response value.

Scanned by CamScanner

The following statement creates a message dialog:

```
JOptionPane.showMessageDialog(null, "The program has been uninstalled.");
```

The message input dialog box can also be spawned using the *showMessageDialog(Component, Object, String, int)* method. The use is identical to the *showInputDialog()* method, with the same arguments, except that *showMessageDialog()* **does not return a value.**
The following statement creates a message dialog box using this technique:

```
JOptionPane.showMessageDialog(null, "An asteroid has destroyed the Earth.", "Asteroid
                                Destruction Alert", JOptionPane.WARNING_MESSAGE);
```

Example:
The following example displays a Login form as shown in diagram 35.11.1. A check is made whether the login name is "sharanam" and the password is "sct2306". In case the username and password is right, then a message dialog is appears informing the user that the login name and password is correct. In case the username or the password or both are incorrect, then a confirm dialog is displayed with "Yes" and "No" options.

Code spec for ConfirmDialogBox.java:

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

public class ConfirmDialogBox extends JPanel implements ActionListener
{
   JLabel lblUserName;
   JLabel lblPassword;
   JTextField txtUserName;
   JPasswordField txtPassword;
   JButton btnLogin;
   JButton btnReset;

   int response;
   String username;
   String password;
   String username1 = "sharanam";
   String password1 = "sct2306";

   public ConfirmDialogBox()
   {
       lblUserName = new JLabel("User Name: ");
       txtUserName = new JTextField(10);
       lblPassword = new JLabel("Password: ");
       txtPassword = new JPasswordField(10);
       txtPassword.setEchoChar('*');
```

Scanned by CamScanner

```
    btnLogin = new JButton("Login");
    btnReset = new JButton("Reset");

    txtUserName.setToolTipText("Enter username");
    txtPassword.setToolTipText("Enter password");

    add(lblUserName);
    add(txtUserName);
    add(lblPassword);
    add(txtPassword);
    add(btnLogin);
    add(btnReset);
    btnLogin.addActionListener(this);
    btnReset.addActionListener(this);
}

public void actionPerformed(ActionEvent actEvt)
{
    if (actEvt.getSource() == btnLogin)
    {
        username = txtUserName.getText();
        password = txtPassword.getText();
    /* Checking for the user name and password */
        if (username.equalsIgnoreCase(username1) &&
            password.equalsIgnoreCase(password1))
        {
        /* In case user name and password is correct, then a
        message dialog box is displayed */
            JOptionPane.showMessageDialog(null, "Welcome to
                                www.sharanamshah.com website", "Information",
                                JOptionPane.INFORMATION_MESSAGE);
            System.exit(0);
        }
        else
        {
        /* In case the user name and password is wrong, then a
        Confirm Dialog box is displayed with "yes" and "no"
        options. If the user chooses the "yes" option, then the
        user is given another try by setting the focus to the
        text field user name. If the user chooses "no" option,
        then the application is closed */
            int response = JOptionPane.showConfirmDialog(null, "The User Name and
                                Password entered is not valid. Want to try again?",
                                "Information", JOptionPane.YES_NO_OPTION,
                                JOptionPane.INFORMATION_MESSAGE);
```

Scanned by CamScanner

```
                if (response == 0)
                {
                    txtUserName.requestFocus();
                }
                else
                {
                    System.exit(0);
                }
            }
        }
        else if (actEvt.getSource() == btnReset)
        {
        /* If the user chooses cancel, the application is closed */
            JOptionPane.showMessageDialog(null, "Are you sure to quit!", "Warning
                                        Message", JOptionPane.WARNING_MESSAGE);
            System.exit(0);
        }
    }

    public static void main(String[] args)
    {
        ConfirmDialogBox confirmDialog = new ConfirmDialogBox();
        JFrame frm = new JFrame("Login Form");
        frm.setContentPane(confirmDialog);
        frm.setSize(250, 350);
        frm.setVisible(true);
        WindowListener listener = new WindowAdapter()
        {
            public void windowClosing(WindowEvent winEvt)
            {
                System.exit(0);
            }
        };
        frm.addWindowListener(listener);
    }
}
```

Explanation:

```
public class ConfirmDialogBox extends JPanel implements ActionListener
{
    ...
}
```

The class named ConfirmDialogBox extends the JPanel class. The JPanel class in return implements the ActionListener interface. Action listeners are probably the easiest and most

Scanned by CamScanner

common event handlers to implement. An action listener is implemented to define what should be done when a user performs certain operation.

```
public void actionPerformed(ActionEvent actEvt)
{
   ...
}
```

The actionPerformed() method has all the instructions that needs to be executed when a particular event occurs. The ActionEvent passed as the parameter to the actionPerformed() method, invokes the actionPerformed() method conducts the programmed battles.

```
if (actEvt.getSource() == btnLogin)
{
   ...
}
```

A check is made whether the Login button was clicked. This is done via the getSource() method of the ActionEvent interface.

```
username = txtUserName.getText();
password = txtPassword.getText();
```

If the Login button is clicked, then the username and the password typed by the user in the text box is retrieved via the getText() method and stored in the variable username and password of String type.

```
if (username.equalsIgnoreCase(username1) && password.equalsIgnoreCase(password1))
{
   ...
}
```

Next, a check is made whether the username and password typed by the user matches with the username and password set in the program. The equalsIgnoreCase() method is used to ignore the case of the letters in the string while matching the two Strings.

```
JOptionPane.showMessageDialog(null, "Welcome to www.sharanamshah.com website",
                              "Information", JOptionPane.INFORMATION_MESSAGE);
System.exit(0);
```

If the username and password matches with the username and password set in the program, then a message dialog box is shown.

JOptionPane class makes it easy to pop up a standard dialog box that prompts users for a value or informs them of something.

Scanned by CamScanner

The showMessageDialog() method informs the user about something that has happened. Following four parameters are passed to the showMessageDialog() method:

- **parentComponent:** Defines the Component that is to be the parent of the particular dialog box. If this parameter is null, then a default Frame is used as the parent and the dialog box is centered on the screen
- **message:** A descriptive message, which is displayed in the dialog box
- **title:** The text, which appears on the title bar of the dialog box
- **messageType:** Defines the style of the message. The look and feel manager may lay out the dialog box differently depending on the messageType value and will often provide a default icon. Following are the values for messageType:
  - ERROR_MESSAGE
  - INFORMATION_MESSAGE
  - WARNING_MESSAGE
  - QUESTION_MESSAGE
  - PLAIN_MESSAGE

System.exit(0) method terminates the currently running JVM. The argument i.e. 0 [zero] serves as a status code. A non-zero status code indicates abnormal termination.

```
else
{
  ...
}
```

If the username and password typed by the user does not match with the username and password set in the program, then the server automatically steps into the else part of the If condition.

```
int response = JOptionPane.showConfirmDialog(null, "The User Name and Password
                entered is not valid. Want to try again?", "Information",
                JOptionPane.YES_NO_OPTION, JOptionPane.INFORMATION_MESSAGE);
```

A variable named response of type integer is declared. It holds the value of confirm dialog box.

The showConfirmDialog() method prompts a confirming question such as Yes / No / Cancel. In addition to the four parameters passed in showMessageDialog() method, one more parameter is passed in the showConfirmDialog() method:

Scanned by CamScanner

- **parentComponent**
- **message**
- **title**
- **optionType:** Defines the set of option buttons, which appear at the bottom of the dialog box. Following are the default values for optionType:
  - DEFAULT_OPTION
  - YES_NO_OPTION
  - YES_NO_CANCEL_OPTION
  - OK_CANCEL_OPTION
  - PLAIN_MESSAGE
- **messageType**

```
if (response == 0)
{
   txtUserName.requestFocus();
}
else
{
   System.exit(0);
}
```

In the else part of the If condition, another check is made whether the Yes button of the confirm dialog box was clicked or not. If The Yes button was clicked, then the user is given another try by returning the window frame with empty text boxes and the cursor focused on the username text box. If the No button was clicked, then the application was terminated.

```
else if (actEvt.getSource() == btnReset)
{
   JOptionPane.showMessageDialog(null, "Are you sure to quit!", "Warning Message",
                                 JOptionPane.WARNING_MESSAGE);
   System.exit(0);
}
```

If the Login button is not clicked, then another check is made whether the Reset button was clicked. If the Rest button is clicked, then a warning message pops up stating that the application will terminate and then the application is terminated.

Once the .java file is ready, it needs to be compiled by the Java compiler and then the .class file is interpreted by the Java interpreter. After compiling and executing the window as shown in diagram 28.11.1 appears.

Scanned by CamScanner

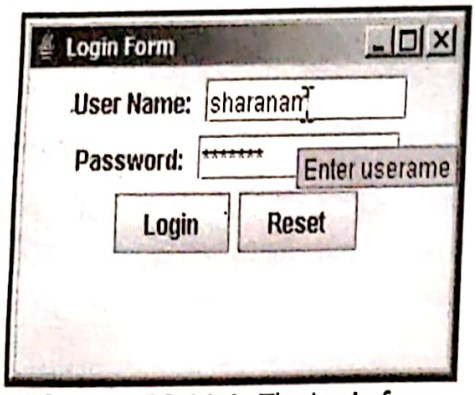


Diagram 28.11.1: The Login form

Enter a wrong username or password. The confirm dialog box pops up stating whether want another try or want to quit the application.

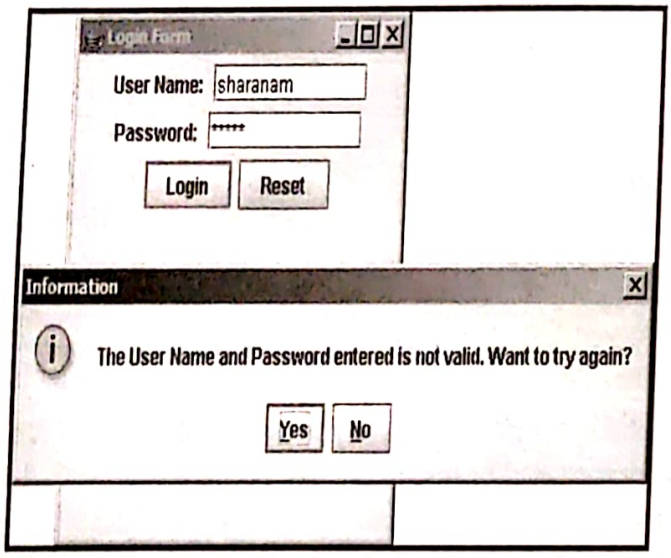


Diagram 28.11.2: Message displayed when a wrong data is entered

Click **Yes**. Next enter the correct username and password. The Message dialog box pops up stating that the user is welcomed to the www.sharanamshah.com website.

Scanned by CamScanner

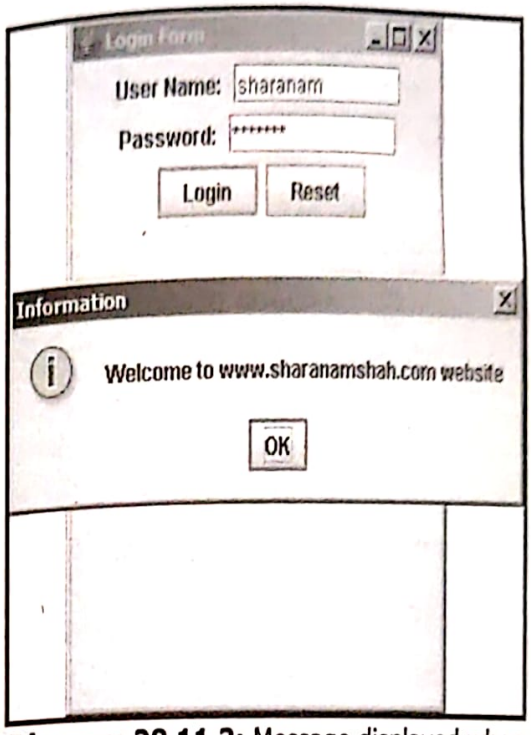


Diagram 28.11.3: Message displayed when correct data is entered

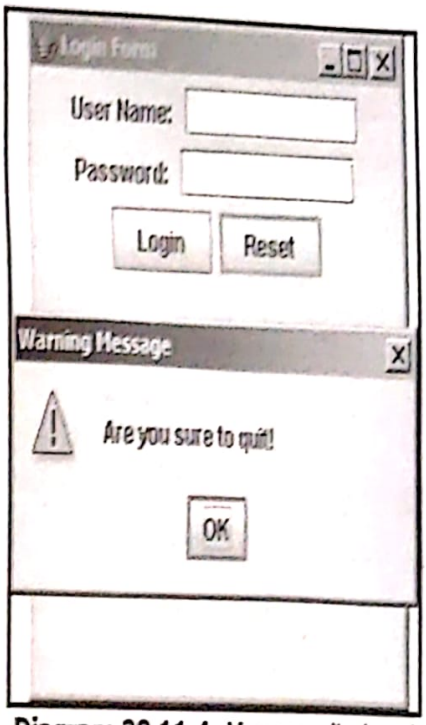


Diagram 28.11.4: Message displayed when the Reset button is clicked

If the **Reset** button is clicked, then a Message dialog box pops up stating that the user really wants to quit. If the **OK** button is clicked, then the application is closed [terminated].

Input Dialog Boxes

An input dialog box displays a question and uses a text field to capture user response. The easiest way to create an input dialog is to use the *showInputDialog(Component, Object)* method. The arguments are the parent component and the string component [i.e. question to be asked] or icon to display within the input dialog. The input dialog method always returns a string which is the user's response to the question asked.

An input dialog can also be created using **showInputDialog(*Component, Object, String, int*)** method.

The arguments described are as follows:

- The value of **String** will be displayed as the input dialog's title
- The **int** will be a class constant which determines which icon will be displayed within the input dialog: ERROR_MESSAGE, INFORMATION_MESSAGE, PLAIN_MESSAGE, QUESTION_MESSAGE or WARNING_MESSAGE

Scanned by CamScanner

inners

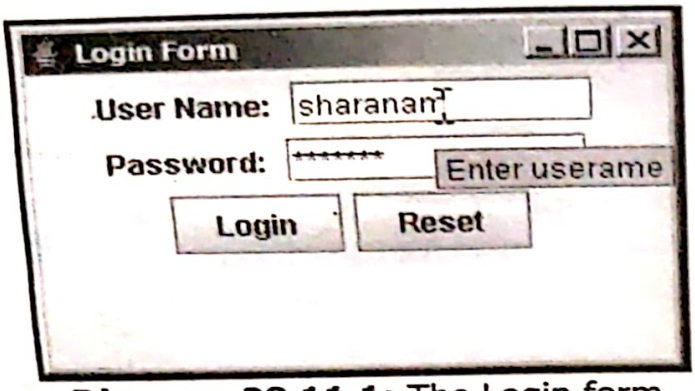
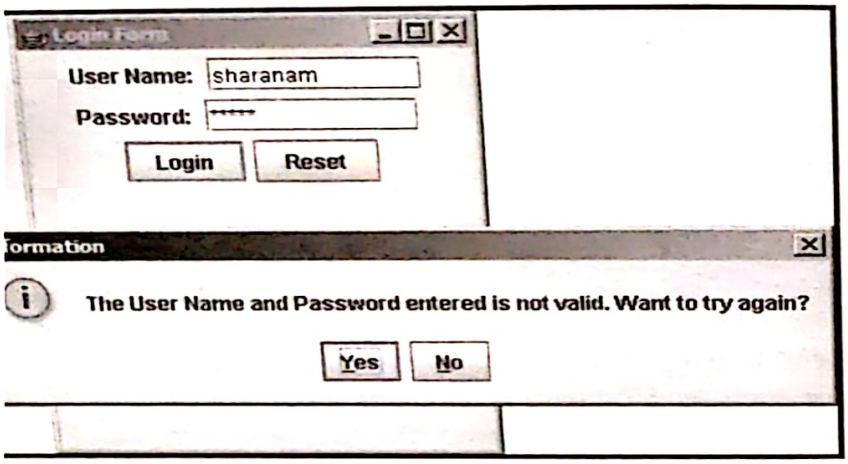


Diagram 28.11.1: The Login form

e or password. The confirm dialog box pops up stating whether want uit the application.



m 28.11.2: Message displayed when a wrong data is entered

er the correct username and password. The Message dialog box pops up r is welcomed to the www.sharanamshah.com website.

Scanned by CamScanner

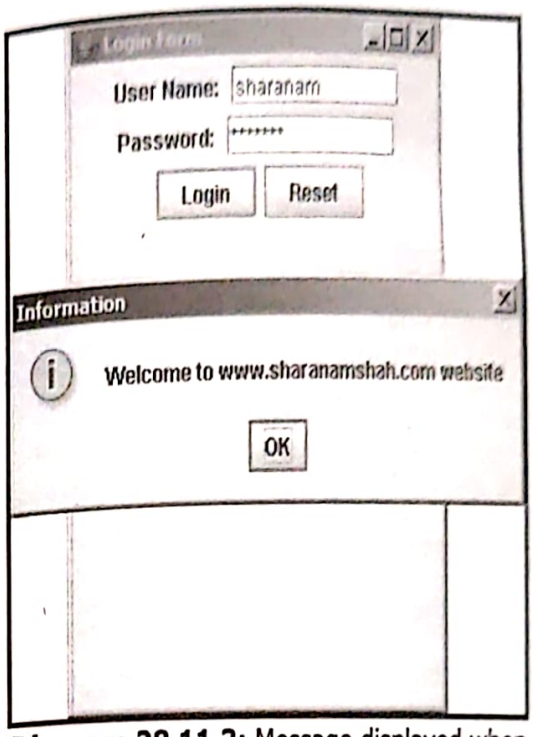


Diagram 28.11.3: Message displayed when correct data is entered

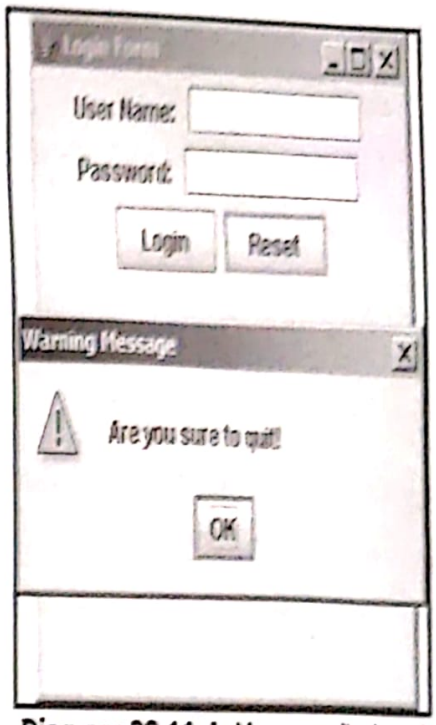


Diagram 28.11.4: Message displayed when the Reset button is clicked

If the **Reset** button is clicked, then a Message dialog box pops up stating that the user really wants to quit. If the **OK** button is clicked, then the application is closed [terminated].

Input Dialog Boxes

An input dialog box displays a question and uses a text field to capture user response. The easiest way to create an input dialog is to use the *showInputDialog(Component, Object)* method. The arguments are the parent component and the string component [i.e. question to be asked] or icon to display within the input dialog. The input dialog method always returns a string which is the user's response to the question asked.

An input dialog can also be created using **showInputDialog(*Component, Object, String, int*)** method.

The arguments described are as follows:

- The value of **String** will be displayed as the input dialog's title
- The **int** will be a class constant which determines which icon will be displayed within the input dialog: ERROR_MESSAGE, INFORMATION_MESSAGE, PLAIN_MESSAGE, QUESTION_MESSAGE or WARNING_MESSAGE

Scanned by CamScanner

The following code snippet creates an input dialog box using this technique:

```
String response = JOptionPane.showInputDialog(null, "What is your ZIP code?", "Enter ZIP
                                                code", JOptionPane.QUESTION_MESSAGE);
```

Option Dialog Boxes

The most complex of the dialogs is the **option** dialog, which combines the features of all the other dialogs. It can be spawned using the *showOptionDialog(Component, Object, String, int, int, Icon, Object[], Object)* method.

The arguments to this method are as follows:

- **Component** indicates what object is the parent of this dialog
- **Object** indicates the text, icon or component to display within the dialog
- **String** will hold what must be displayed in the title bar of the dialog
- **Int** which is actually a class constant YES_NO_OPTION or YES_NO_CANCEL_OPTION indicates which buttons will be displayed by the dialog
- **Int** which is actually a class constant ERROR_MESSAGE, INFORMATION_MESSAGE, PLAIN_MESSAGE, QUESTION_MESSAGE or WARNING_MESSAGE determines the icon that the dialog will display adjacent to String **or** the literal 0 if none of these icons must be used
- **Object**, an Icon object to display instead of one of the icons in the preceding argument
- **Object[]**, An array of objects holding the components or the other objects that represent choices in the dialog box, if YES_NO_OPTION or YES_NO_CANCEL_OPTION are not being used
- Object, which represents the default selection if YES_NO_OPTION or YES_NO_CANCEL_OPTION are not being used

The last two arguments enable a programmer to determine a wide range of choices for the option dialog. The programmer can create an array of buttons, labels, text fields or even a mixture of different components as an object array. These components are displayed using the flow layout manager within the option dialog.

REMINDER

There is no way to specify a different layout manager within the option dialog.

The following example creates an option dialog box that uses an array of *JButton* objects for the options it displays in the dialog with the gender[2] element default selection:

```
JButton [] gender = new JButton [3];
```

Scanned by CamScanner

```
gender[0] = new JButton("Male");
gender[1] = new JButton("Female");
gender[2] = new JButton("None of your Business");
int response = JOptionPane.showOptionDialog(null, "What is your gender?", "Gender", 0,
                                JOptionPane.QUESTION_MESSAGE, null, gender, gender[2]);
```

Example:

The following example accepts user information through a series of input dialogs such as User name, Web site name and whether the site is a Personal, Commercial or an Unknown site. The information accepted is displayed in another the text field.

Code spec for WebsiteAddress.java:

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

public class WebsiteAddress extends JFrame
{
    private JLabel lblUserName = new JLabel("UserName: ", SwingConstants.RIGHT);
    private JTextField txtUserName;
    private JLabel lblWebsite = new JLabel("Web site: ", SwingConstants.RIGHT);
    private JTextField txtWebsite;
    private JLabel lblType = new JLabel("Type: ", SwingConstants.RIGHT);
    private JTextField txtType;
    public WebsiteAddress()
    {
        super("Website Information");

        String strUserName = JOptionPane.showInputDialog(null, "Enter Username: ");
        txtUserName = new JTextField(strUserName, 20);

        String strWebsite = JOptionPane.showInputDialog(null, "Enter the website name: ");
        txtWebsite = new JTextField(strWebsite, 20);

        String[] choices = { "Personal", "Commercial", "Unknown" };
        int intType = JOptionPane.showOptionDialog(null,"What type of site is it?", "Site
                                Type", 0, JOptionPane.QUESTION_MESSAGE, null,
                                choices, choices[0]);
        txtType = new JTextField(choices[intType], 20);

        JPanel pane = new JPanel();
        pane.setLayout(new GridLayout(3, 2));
        pane.add(lblUserName);
        pane.add(txtUserName);
        pane.add(lblWebsite);
        pane.add(txtWebsite);
        pane.add(lblType);
```

Scanned by CamScanner

```
        pane.add(txtType);
        setContentPane(pane);
    }

    public static void main(String[] args)
    {
        WebsiteAddress frame = new WebsiteAddress();
        WindowListener winlist = new WindowAdapter()
        {
            public void windowClosing(WindowEvent e)
            {
                System.exit(0);
            }
        };
        frame.addWindowListener(winlist);
        frame.setVisible(true);
    }
}
```

Explanation:

```
String strUserName = JOptionPane.showInputDialog(null, "Enter Username: ");
txtUserName = new JTextField(strUserName, 20);
String strWebsite = JOptionPane.showInputDialog(null, "Enter the website name: ");
txtWebsite = new JTextField(strWebsite, 20);
```

The variables named strUserName and strWebsite of type String are declared. The showInputDialog() method prompts the user for entering the value. The value, thus, entered are stored in the variables strUserName and strWebsite.

The value stored in variables strUserName and strWebsite are then shown in the text box, which are displayed at the end as shown in diagram 35.12.4.

```
String[] choices = { "Personal", "Commercial", "Unknown" };
int intType = JOptionPane.showOptionDialog(null,"What type of site is it?", "Site Type", 0,
                        JOptionPane.QUESTION_MESSAGE, null, choices, choices[0]);
txtType = new JTextField(choices[intType], 20);
```

An array named choices of type String is declared and three values are passed to it. The variable named intType of type integer is declared. The showOptionDialog() method grands unification of the Input dialog box, Message dialog box and Confirm dialog box.

Here the three buttons are custom made i.e. the choices array's values are shown as the name of the buttons as shown in diagram 28.12.3.

The button type clicked is stored in the variable intType.

Scanned by CamScanner

The value stored in variable intType is then shown in the text box, which is displayed at the end as shown in diagram 28.12.4.

Once the .java file is ready, it needs to be compiled by the Java compiler and then the .class file is interpreted by the Java interpreter. After compiling and executing the window as shown in diagram 28.12.1 appears.

Input dialog box appears prompting the user to enter the user name as shown in diagram 28.12.1.

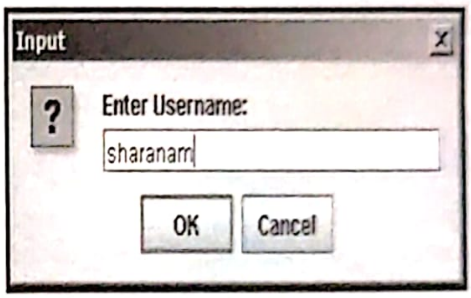


Diagram 28.12.1: Input Dialog box to accept the user name

Enter the user name and click OK. Another Input dialog box pops up prompting to enter the website name as shown in diagram 28.12.2.

Diagram 28.12.2: Input Dialog box to accept the website's name

Enter the name of the web site and click OK. Option dialog box pops up prompting the user to select the type of the web site as shown in diagram 28.12.3.

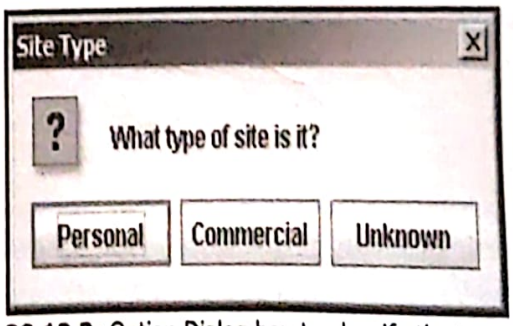


Diagram 28.12.3: Option Dialog box to classify the type of the site

Scanned by CamScanner

Select the type of the site. A window appears which displays the value entered and selected by the user as shown in diagram 28.12.4.

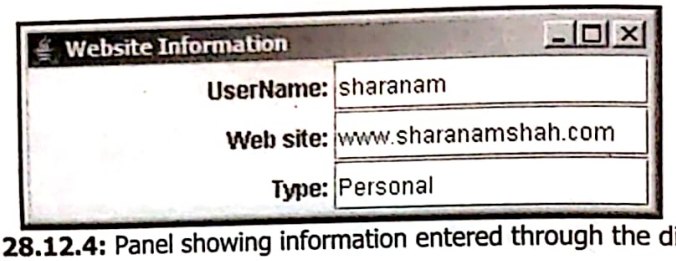


Diagram 28.12.4: Panel showing information entered through the dialog boxes

Timers And Sliders

Swing timers can be spawned using the class Timer. This is a subclass of the java.lang.Object. Timers periodically trigger events of type ActionEvent that are listened for by their target listener. The listener class must implement the interface ActionListener and enclose execution code inside the method actionPerformed(). The target listener is registered with the timer through its constructor.

To use a timer in a program, instantiate a timer object and execute its start() method. The start() method makes the timer fire events in a separate thread and at the specified delay interval passed to start() as an argument. By passing the method setRepeats() the argument FALSE, the timer fire only once. Timers can be restarted or stopped by invoking its stop() and restart() methods when required.

Timer Constructor

The Swing timer can be instantiated with only one constructor. The constructor requires specifying the specified delay at which the timer will fire and its action listener. The action listener implements the interface ActionListener. The following is the constructor of the Timer class:

```
public Timer (int delay, ActionListener listener)
```

Sliders

Swing Sliders are implemented using the *JSlider class*. This enables a number to be set by sliding the control's 'thumb' within the range of its minimum and maximum values. In many cases, a slider can be used for numeric input instead of a text field. This has the advantage of restricting numeric input to between a range of acceptable [pre-determined] values.

Scanned by CamScanner

A Slider is often seen in media players to adjust audio volume, change channel frequencies, set audio channel frequency filters, the contrast and/or brightness of a picture displayed and so on.

Sliders are horizontal by default. The orientation can be explicitly set by using two class constants of the *SwingConstants* class viz. **HORIZONTAL** or **VERTICAL**.

The following are the *JSlider* constructor methods:

- **JSlider(int, int):** Slider with a specified minimum value and maximum value
- **JSlider(int, int, int):** Slider with a specified minimum value, maximum value and a starting value at which the slider's 'Thumb' will be displayed
- **JSlider(int, int, int, int):** Slider with the specified orientation, minimum and maximum value and a starting value at which the slider's 'Thumb' will be displayed

Slider components have an optional label that can be used to indicate the minimum value, maximum value and two different sets of tick marks ranging between the values.

The elements of this label are established by calling several methods of JSlider:

- **setMajorTickSpacing(int):** Separate major tick marks by the specified distance. The distance is not in pixels, but is a value between the minimum and maximum values represented by the slider
- **setMinorTickSpacing(int):** Separate minor tick marks by the specified distance. These are normally displayed between Major tick marks. Minor ticks are half as tall as major tick marks
- **setPaintTicks(boolean):** Determine whether the tick marks should be displayed [TRUE] or not [FALSE]
- **setPaintLabels(boolean):** Determine whether the numeric label of the slider should be displayed [TRUE] or not [FALSE]

These methods should be called and appropriate arguments passed to them before the slider is added to a container.

Example:
The following code snippet displays a slider that controls wave length.

Code spec for SineWave.java:

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

Scanned by CamScanner

```
class SineDraw extends JPanel
{
   private static final int SCALEFACTOR = 200;
   private int cycles;
   private int points;
   private double[] sines;
   private int[] pts;

   public SineDraw()
   {
       setCycles(5);
   }

   public void setCycles(int newCycles)
   {
       cycles = newCycles;
       points = SCALEFACTOR * cycles * 2;
       sines = new double[points];
       for (int i = 0; i < points; i++)
       {
           double radians = (Math.PI / SCALEFACTOR) * i; sines[i] = Math.sin(radians);
       }
       repaint();
   }

   public void paintComponent(Graphics g)
   {
       super.paintComponent(g);
       int maxWidth = getWidth();
       double hstep = (double) maxWidth / (double) points;
       int maxHeight = getHeight();
       pts = new int[points];
       for (int i = 0; i < points; i++)
           pts[i] = (int) (sines[i] * maxHeight / 2 * .95 + maxHeight / 2);
       g.setColor(Color.RED);
       for (int i = 1; i < points; i++)
       {
           int x1 = (int) ((i - 1) * hstep);
           int x2 = (int) (i * hstep);
           int y1 = pts[i - 1];
           int y2 = pts[i];
           g.drawLine(x1, y1, x2, y2);
       }
   }
}

public class SineWave extends JApplet
{
```

Scanned by CamScanner

```
private SineDraw sines = new SineDraw();
private JSlider adjustCycles = new JSlider(1, 30, 5);

public void init()
{
    Container cp = getContentPane();
    cp.add(sines);
    adjustCycles.addChangeListener(new ChangeListener()
    {
        public void stateChanged(ChangeEvent e)
        {
            sines.setCycles(((JSlider) e.getSource()).getValue());
        }
    });
    cp.add(BorderLayout.SOUTH, adjustCycles);
}

public static void main(String[] args)
{
    run(new SineWave(), 700, 400);
}
public static void run(JApplet applet, int width, int height)
{
    JFrame frame = new JFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.getContentPane().add(applet);
    frame.setSize(width, height);
    applet.init();
    applet.start();
    frame.setVisible(true);
}
}
```

Explanation:

```
class SineDraw extends JPanel
{
  ...
}
```

A class named SineDraw is declared which extends the JPanel class. When the application is compiled, this SineDraw class becomes a new .class file as compared to the file name SineWave.java.

Inside the SineDraw class, following code spec does:

- Private variables are declared

```
private static final int SCALEFACTOR = 200;
private int cycles;
```

Scanned by CamScanner

```
private int points;
private double[] sines;
private int[] pts;
```

- A constructor named SineDraw is created, which calls the setCycles() method declared in the same class SineDraw. The setCycles() method is passed the number of cycles as the parameter

```
public SineDraw()
{
   setCycles(5);
}
```

- The setCycles() method is declared, which passes an integer type variable named newCycles. The variable named cycles holds the value passed in the SineDraw constructor. Then the value of the variable named points is calculated and the same is stored in the variable named sines in the type double. A For loop is generated where by the radians variable is calculated. Then the repaint() method is called, which repaints the component

```
public void setCycles(int newCycles)
{
   cycles = newCycles;
   points = SCALEFACTOR * cycles * 2;
   sines = new double[points];
   for (int i = 0; i < points; i++)
   {
        double radians = (Math.PI / SCALEFACTOR) * i; sines[i] = Math.sin(radians);
   }
   repaint();
}
```

- The paintComponent() method is declared, which passes a graphics as the parameter. Here the wave of red color is drawn by the drawLine() method, which takes four parameters viz. the first point's x coordinate, the first point's y coordinate, the second point's x coordinate and the second point's y coordinate

```
public void paintComponent(Graphics g)
{
   super.paintComponent(g);
   int maxWidth = getWidth();
   double hstep = (double) maxWidth / (double) points;
   int maxHeight = getHeight();
   pts = new int[points];
   for (int i = 0; i < points; i++)
        pts[i] = (int) (sines[i] * maxHeight / 2 * .95 + maxHeight / 2);
   g.setColor(Color.RED);
   for (int i = 1; i < points; i++)
```

Scanned by CamScanner

```
    {
        int x1 = (int) ((i - 1) * hstep);
        int x2 = (int) (i * hstep);
        int y1 = pts[i - 1];
        int y2 = pts[i];
        g.drawLine(x1, y1, x2, y2);
    }
}
```

The class SineDraw ends here. Next in the same SineWave.java file another class is constructed.

```
public class SineWave extends JApplet
{
    . . .
}
```

A class named SinwWave is constructed which extends JApplet class.

Inside the SineWave class, following code spec does:

- An object of SineDraw class is created. An object of JSlider class is created and is passed the minimum value of the slider, the maximum value of the slider and the initial value of the slider as the parameters

```
private SineDraw sines = new SineDraw();
private JSlider adjustCycles = new JSlider(1, 30, 5);
```

- An initialization method is declared. Inside the init() method the content pane is retrieved and the object of the SineDraw class is added to the content pane. A change listener is registered with the slider via the addChangeListener() method. A change listener informs the server that there is a change in the value of the slider

```
public void init()
{
    Container cp = getContentPane();
    cp.add(sines);
    adjustCycles.addChangeListener(new ChangeListener()
    {
        public void stateChanged(ChangeEvent e)
        {
            sines.setCycles(((JSlider) e.getSource()).getValue());
        }
    });
    cp.add(BorderLayout.SOUTH, adjustCycles);
}
```

- The run() method is called. The run() method is passed an object of the SineWave class, the width and height of the frame as the parameters

Scanned by CamScanner

```
public static void main(String[] args)
{
    run(new SineWave(), 700, 400);
}
```

- The run() method is declared and its parameters are an object of the SineWave class, the width and height of the frame. Inside the run() method the following is done:
  - An object of JFrame is created
  - The setDefaultCloseOperation() method of the JFrame class is used to set the operation that will happen by default when the user initiates a "close" on the frame. Here, EXIT_ON_CIOSE means exit the application using the System exit method
  - The content pane is retrieved and the object of the SineWave class is added to it
  - The size of the frame is set with the value entered while calling the run() method in the main() method
  - The SineWave class is initialized via the init() method
  - The SineWave class is started via the start() method
  - The frame is made visible via the setVisible() method

```
public static void run(JApplet applet, int width, int height)
{
    JFrame frame = new JFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.getContentPane().add(applet);
    frame.setSize(width, height);
    applet.init();
    applet.start();
    frame.setVisible(true);
}
```

Code spec of index.jsp:

```
<%@ page language="java" contentType="text/html" import="java.lang.*" %>
<HTML>
  <HEAD>
      <TITLE>Welcome To The World Of Swing</TITLE>
  </HEAD>
  <BODY>
      <jsp:plugin type="applet" code="SineWave.class" codebase="/MyWebApplication/"
                  width="300" height="200">
          <jsp:fallback>Unable To load Applet</jsp:fallback>
      </jsp:plugin>
  </BODY>
</HTML>
```

Scanned by CamScanner

Make a war file and deploy the application. After deploying the file, move the class file to the root directory of the web application as explained earlier.

Next, run the JSP file in the browser as shown in diagram 28.13.

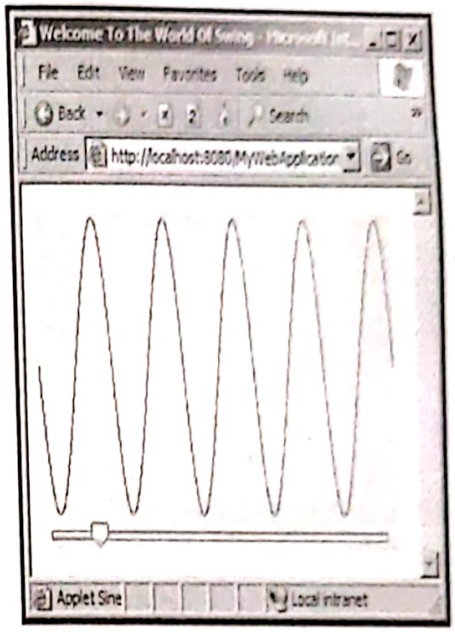


Diagram 28.13: Running the Applet

Progress Bars

Progress bars are components used when long tasks are being performed by any code spec to communicate to the user how much time is left before the task completes. A task's progress can be represented numerically. Progress bars are created by specifying a minimum and maximum value. These two values represent the points at which the task begins and ends. Progress bars also provide proof that the program is still running and the system has not crashed.

Swing progress bars are implemented using the *JProgressBar class*

To spawn and display a progress bar in any application the following constructor methods can be used:

- **JProgressBar():** Creates a new Progress bar
- **JProgressBar(int, int):** Creates a new Progress bar with a programmer specified minimum and maximum value
- **JProgressBar(int, int, int):** Creates a new Progress bar with the specified orientation i.e. Horizontal or Vertical, minimum and maximum value

Scanned by CamScanner

The first **int** is actually an environment constant i.e. *SwingConstants.VERTICAL* and *SwingConstants.HORIZONTAL* class constants. Progress bars are horizontal by default.

A progress bar can have its minimum and maximum values set up by using the progress bar's *setMinimum(int)* and *setMaximum(int)* methods with the appropriate values.

To move a progress bar's level indicator, invoke its *setValue(int)* method with a value indicating how much of the task is complete at that moment. This value should be any value between the minimum and maximum values specified when creating the bar.

Progress bars often include a text label in addition to the graphic of an empty box filling up. This label can display the percentage of the task that has to be completed. This is done by calling the *setStringPainted(boolean)* method with TRUE as an argument. A FALSE argument turns this label off.

Example:
The following code snippet demonstrates how a JProgressBar can be used. The code snippet creates a thread that updates the progress bar as the program executes as shown diagram 35.14.

Code spec for ProgressBarSample.java:

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

public class ProgressBarSample extends JPanel
{
   static ProgressThread progressThread;
   static JProgressBar progressBar;
   static JFrame frm;

   public ProgressBarSample()
   {
      setLayout(new BorderLayout());
      progressBar = new JProgressBar();
      add(progressBar, "Center");

      JPanel buttonPanel = new JPanel();
      JButton startButton = new JButton("Start");
      buttonPanel.add(startButton);

      startButton.addActionListener(new ActionListener()
      {
         public void actionPerformed(ActionEvent actEvt)
```

Scanned by CamScanner

```
        {
            startProgressBar();
        }
    });

    JButton stopButton = new JButton("Stop");
    buttonPanel.add(stopButton);
    stopButton.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent actEvt)
        {
            stopProgressBar();
        }
    });
    add(buttonPanel, BorderLayout.SOUTH);
}

public void startProgressBar()
{
    if(progressThread == null || !progressThread.isAlive())
    {
        progressThread = new ProgressThread(progressBar);
        progressThread.start();
    }
}

public void stopProgressBar()
{
    progressThread.setStop(true);
}

public static void main(String args[])
{
    frm = new JFrame("Copying Files.....");
    ProgressBarSample progressSample = new ProgressBarSample();
    frm.getContentPane().add("Center", progressSample);
    frm.setSize(200, 100);
    frm.addWindowListener(new WindowAdapter()
    {
        public void windowClosing(WindowEvent winEvt)
        {
            System.exit(0);
        }
    });
    frm.setVisible(true);
}
}

class ProgressThread extends Thread
```

Scanned by CamScanner

```
{
  JProgressBar progressBar;
  boolean stopStatus = false;
  boolean aliveStatus = false;

  public ProgressThread(JProgressBar progressBar)
  {
      this.progressBar = progressBar;
  }

  public void setStop(boolean value)
  {
      stopStatus = value;
  }

  public void run()
  {
      int minimum = 0;
      int maximum = 50;
      progressBar.setMinimum(minimum);
      progressBar.setMaximum(maximum);
      progressBar.setValue(minimum);
      for (int x = minimum; x <= maximum; x++)
      {
          if (stopStatus)
          {
              break;
          }
          else
          {
              progressBar.setValue(x);
              try
              {
                  Thread.sleep(100);
              }
              catch (InterruptedException intrptExcp)
              {
              }
          }
      }
      aliveStatus = false;
  }
}
```

Scanned by CamScanner

Explanation:

```
public class ProgressBarSample extends JPanel
{
   . . .
}
```

A class named SinwWave is constructed which extends JPanel class.

Inside the ProgressBarSample class, following code spec does:

- Three static objects are declared viz.
  - ProgressThread: Is the class declared at the end of the ProgressBarSample class
  - JProgressBar: Implements a progress bar
  - JFrame

```
static ProgressThread progressThread;
static JProgressBar progressBar;
static JFrame frm;
```

- A constructor named ProgressBarSample is declared. Inside the constructor the following code spec does:
  - A BorderLayout is set via the setLayout() method
  - A new object of JProgressBar is created and is added via the add() method
  - An object of JPanel is created
  - Two objects of JButton are created and the same are added to the panel via the add() method of the JPanel class
  - An action listener is registered for the startButton on whose click startProgressBar() method is called
  - An action listener is registered for the stopButton on whose click stopProgressBar() method is called
  - The panel is added to the frame via the add() method

```
public ProgressBarSample()
{
   setLayout(new BorderLayout());
   progressBar = new JProgressBar();
   add(progressBar, "Center");
   JPanel buttonPanel = new JPanel();

   JButton startButton = new JButton("Start");
   buttonPanel.add(startButton);

   startButton.addActionListener(new ActionListener()
```

Scanned by CamScanner

```
{
    public void actionPerformed(ActionEvent actEvt)
    {
        startProgressBar();
    }
});
JButton stopButton = new JButton("Stop");
buttonPanel.add(stopButton);

stopButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent actEvt)
    {
        stopProgressBar();
    }
});
add(buttonPanel, BorderLayout.SOUTH);
}
```

- The startProgressBar() method is created. Inside the startProgressBar() method, a check is made whether the progressThread is null or is not alive. If either of the two is true, then a new object of ProgressThread is created and then a connection is started via the start() method when the Start button is clicked

```
public void startProgressBar()
{
  if(progressThread == null || !progressThread.isAlive())
  {
      progressThread = new ProgressThread(progressBar);
      progressThread.start();
  }
}
```

- The stopProgressBar() method is created. Inside this method the setStop() method of the ProgressThread class is called when the Stop button is clicked

```
public void stopProgressBar()
{
  progressThread.setStop(true);
}
```

- The main() method is create. Inside the main() method the following is done:
  - An object of JFrame is created
  - An object of ProgressBarSample class is created
  - The content pane is retrieved and the object of the ProgressBarSample class is added to it

Scanned by CamScanner

- The size of the frame is set
- The application is terminated via the WindowAdapter() method
- The frame is made visible

```
public static void main(String args[])
{
    frm = new JFrame("Copying Files.....");
    ProgressBarSample progressSample = new ProgressBarSample();
    frm.getContentPane().add("Center", progressSample);
    frm.setSize(200, 100);
    frm.addWindowListener(new WindowAdapter()
    {
        public void windowClosing(WindowEvent winEvt)
        {
            System.exit(0);
        }
    });
    frm.setVisible(true);
}
```

The class ProgressBarSample ends here. Next in the same ProgressBarSample.java file another class is constructed.

```
class ProgressThread extends Thread
{
    ...
}
```

A class named ProgressThread is declared which extends the Thread class. When the application is compiled, this ProgressThread class becomes a new .class file as compared to the file name ProgressBarsample.java.

Inside the ProgressThread class, following code spec does:

- An object of JProgressBar and two Boolean variables with the false value are declared

```
JProgressBar progressBar;
boolean stopStatus = false;
boolean aliveStatus = false;
```

- A constructor named Progressthread is created and it accepts the object of JProgressBar as the parameter. The object of JProgressBar is stored in this object of JProgressBar

```
public ProgressThread(JProgressBar progressBar)
{
    this.progressBar = progressBar;
}
```

Scanned by CamScanner

- The setStop() method is created, which accepts the value and stores the same value in the stopStatus variable

```
public void setStop(boolean value)
{
    stopStatus = value;
}
```

- The run() method is declared. Inside the run() method, the following is done:
  - Two integer variables are declared and its values are set
  - The minimum, maximum and the current value of the progress bar is set via the setMinimum(), setMaximum() and setValue() methods respectively
  - A For loop is generated. Inside the For loop
    - A check is made whether the Stop button was clicked. If the Stop button was clicked, then the progress bar status is stopped
    - Else the current value of the progress bar is set
      - Inside a else block, a try block is created
      - Inside the try block the current thread is blocked for a specified number of milliseconds
  - The value of the aliveStatus is set to false

```
public void run()
{
    int minimum = 0;
    int maximum = 50;
    progressBar.setMinimum(minimum);
    progressBar.setMaximum(maximum);
    progressBar.setValue(minimum);
    for (int x = minimum; x <= maximum; x++)
    {
        if (stopStatus)
        {
            break;
        }
        else
        {
            progressBar.setValue(x);
            try
            {
                Thread.sleep(100);
            }
            catch (InterruptedException intrptExcp)
            {
            }
```

Scanned by CamScanner

```
        }
    }
    aliveStatus = false;
}
```

Once the .java file is ready, it needs to be compiled by the Java compiler and then the .class file is interpreted by the Java interpreter. After compiling and executing the window as shown in diagram 28.14 appears.

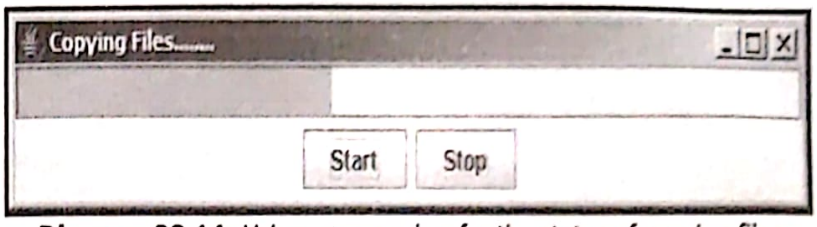


Diagram 28.14: Using progress bar for the status of copying files

Tables

A table is an object that can display data in a row and column format in a GUI. Optionally a table can allow editing.

Swing table models are powerful, flexible and easy to implement.

Simple Tables

Swing tables are created using the class JTable that extends the class JComponent. JTable is stored in the package javax.swing. This section demonstrates how to create simple tables by using two table constructors.

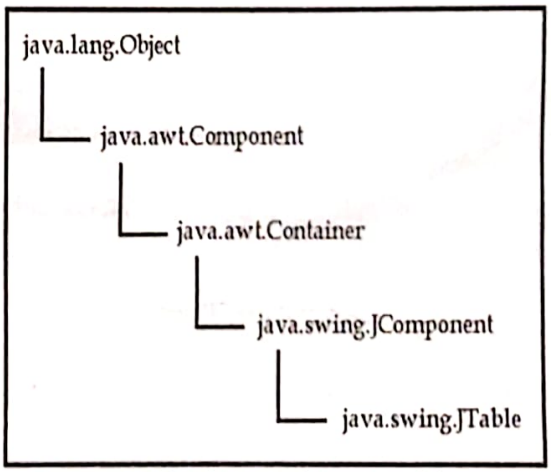


Diagram 28.15: The JTable class hierarchy

Scanned by CamScanner