

Chapter 19

SWING AND GUI COMPONENTS

In this chapter, creation of Graphical User Interface (GUI) screens using graphical components like buttons, labels, checkboxes defined in Swing classes are explained.

19.1 The Origin of Swing

Creation of a user-interface screen is an important feature of any modern computer language. Initially, Java introduced a package called Abstract Window ToolKit(AWT). This package contained a large number of classes and interfaces that supported the creation of Graphical User Interface(GUI) components on user-interface screens. As the goal of Java developers was to create platform-independent programs, the AWT classes made use of the native methods of the respective machines. That is, the AWT made use of graphical methods defined for Windows OS for Windows machines, made use of graphical methods defined for Mac OS for Macintosh machines. As a result, the appearance (look and feel) of the screen was different for different platforms. Therefore, to have the same appearance of the screen called pluggable look and feel (pl and f) in all platforms, new classes were added in the new versions of Java.(versions after JDK 1.1). These new classes are packed in Swing. Java2 contains Swing. The Swing classes are a part of the Java Foundation Classes(JFC). The Swing classes are contained in a Java extension package called javax.swing.

The methods in the Swing classes do not use the native graphical methods directly, but use them as interfaces. Methods in Swing classes are written purely in Java. The methods in Swing classes can be used to create screens with the same look and feel of the screens in different platforms. Swing classes have methods, which are far superior in capabilities than those in the corresponding AWT classes.

The Swing classes are subclasses of **java.awt.Container** and **java.awt.Component**. The name of the Swing class starts with the letter J. The top-level class of Swing is **JComponent**. **JComponent** is both a container and a component. GUI components like button, label, checkbox, panel, text, etc., are handled in **JComponent** class. GUI components can be added on a panel window or a frame window. The frame in Swing is handled in **JFrame** class. The **JComponent** class and AWT class hierarchy is shown in fig.19.1.

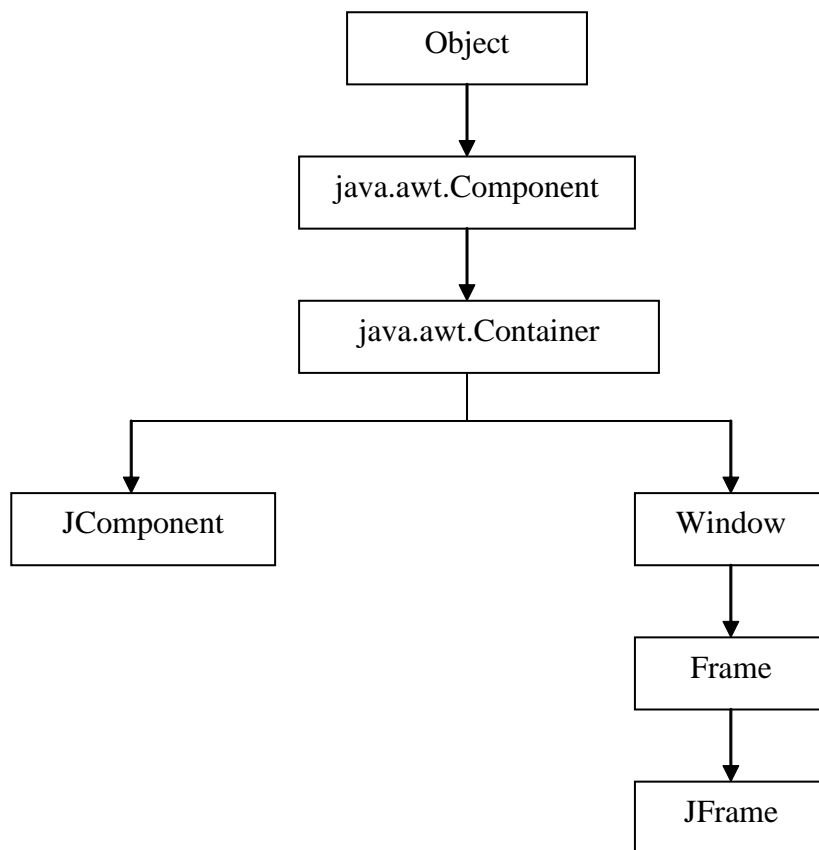
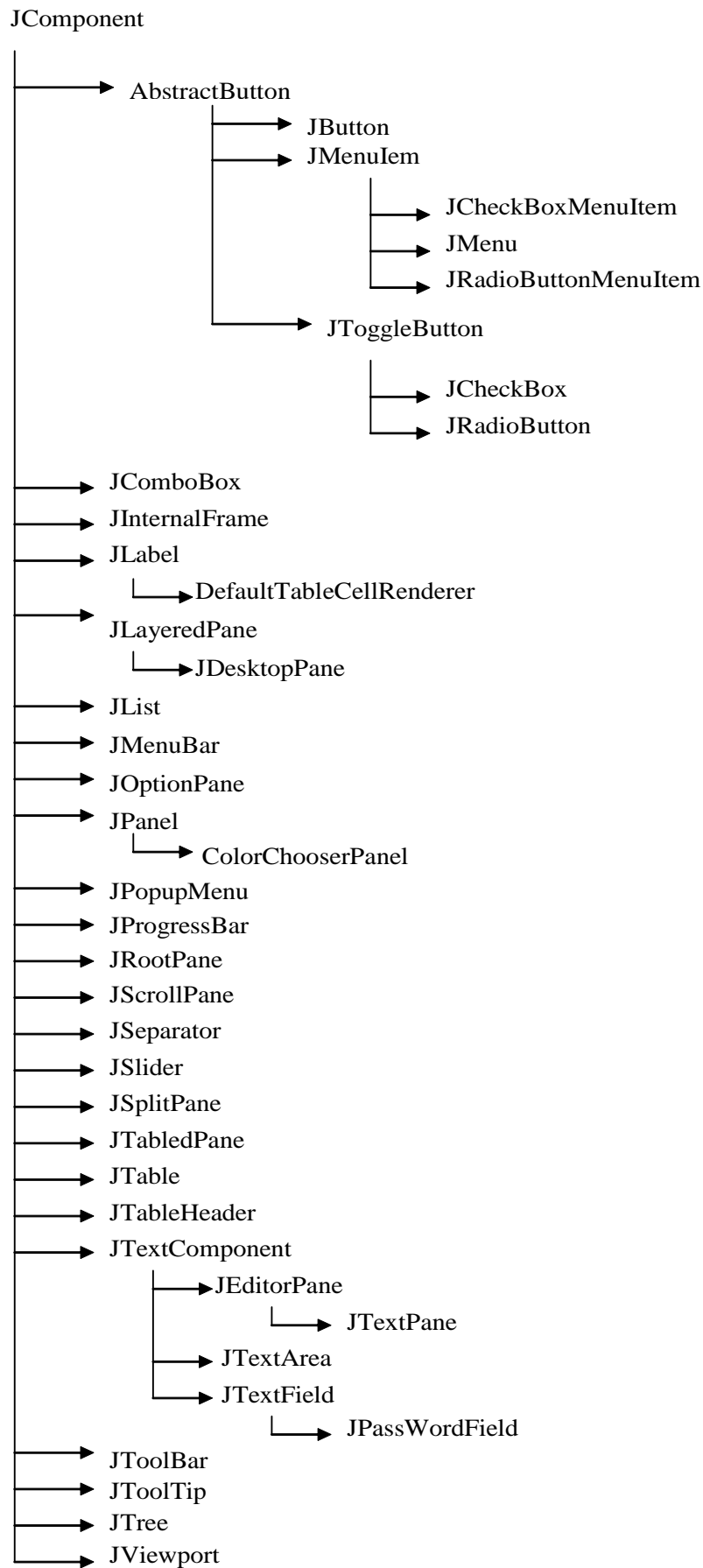


Fig.19.1 Class Hierarchy of AWT, JComponent and JFrame

19.1.1 JComponent

The **JComponent** class is a subclass of **Container**. This class contains a large number of subclasses, which define components like **JButton**, **JLabel**, which act as user-interface components. The subclasses of **JComponent** are given in fig.19.2.

**Fig.19.2 Subclasses of JComponent**

19.2 Creating Windows in Swing

A conventional window is created in Swing in a top-level window. The top-level window is supported by **JFrame**, **JApplet**, **JDialog** and **JWindow** classes. They are called root containers. These root containers have several panes, **JRootPane**, **JMenuBar**, **JLayeredPane**, **ContentPane** and **GlassPane**, as shown in fig. 19.3.

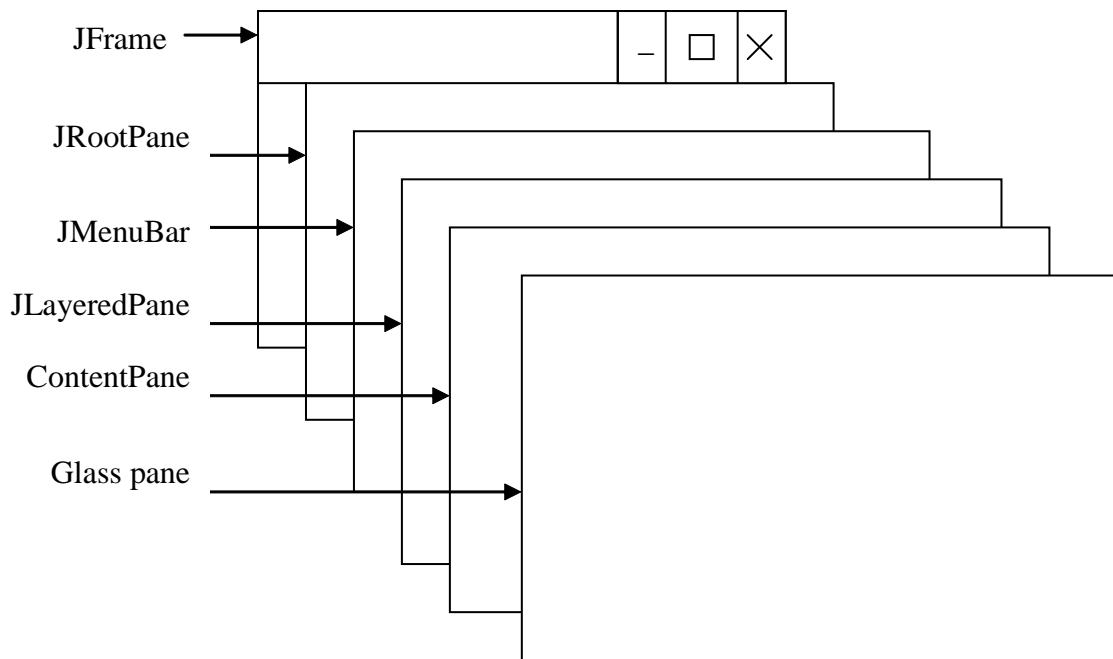


Fig.19.3 Structure of Root Containers

The above container model is implemented in **JRootPane** class. The root container classes have methods that forward requests to the **JRootPane**. These methods are defined in the **RootPaneContainer** interface.

The most frequently used method to get the content pane is:

```
getContentPane()
```

which returns a container object

In Swing, user-interface components like button, checkbox and setting out layout managers must be done on a content pane. The components are added to the content pane using **add()** method.

Therefore to create a user-interface window, the following steps are to be followed:

- Create a root container using **JFrame** or **JApplet** or **JDialog** or **JWindow**.
- Get the container using **getContentPane** method.
- Create components.
- Attach the components to a container using **add()** method.

In the following sections, we will see how different user-interface screens with different components are created. As Swing is an extension of AWT, some of the classes defined in AWT are still used.



All user-interface components are to be drawn only on a content pane.

19.2.1 Creating JFrame Windows

A **JFrame** window is a standard style window. **JFrame** is a subclass of **Frame** class. When a **JFrame** window is created, its size is (0,0) and is invisible. A **JFrame** window, though containing Iconify, Minimize and Close icons, the Close option, just sets the window invisible and does not close the window. A **JFrame** generates the following window events:

windowOpened	windowClosing
windowClosed	windowIconified
windowDeiconified	windowActivated
windowDeactivated	

These window events are handled in java.awt.event package.

Constructors

A JFrame window is created with the following constructors:

`JFrame()`

Creates an invisible window without a title

`JFrame(String title)`

Creates an invisible window with a title given by the String title

Methods

The JFrame class has numerous inherited methods and some of its own. Some of them are :

`String getTitle()`

Returns a string representing the title of the frame

`void setTitle(String title)`

Sets the title of the frame to this string

`void setVisible(boolean b)`

Shows or hides this frame window

If b is true, it shows the window, otherwise it hides it.

`void setSize(int width, int height)`

Sets the size of the window to the specified width and height in pixels

`void setLayout(LayoutManager mgr)`

Sets the layout manager for this container

`int getX()`

Returns the X component of the window location

`int getY()`

Returns the Y component of the window location

`void setLocation(int x, int y)`

Moves the frame window to a new location on the screen; the top left corner of the window is specified by x and y.

`int getHeight()`

Returns the current height of the window

`int getWidth()`

Returns the current width of the window

Once a **JFrame** window is created, there is no way to close it. The simplest manual method is to press CTRL + C in DOS mode to close the window. The other method is to capture the **windowClosing** event through **WindowListener** and call the **System.exit(0)** method.

The following program 19.1 illustrates the creation of a **JFrame** window:

Program 19.1

```
// This program creates a blank JFrame window.
// somasundaramk@yahoo.com
import javax.swing.JFrame;
class JFrame1
{
    public static void main(String args [])
    {
        JFrame f = new JFrame("Blank JFrame");
        f.setSize(200, 200);
        f.setVisible(true);
    }
}
```

The above program gives the following output screen:

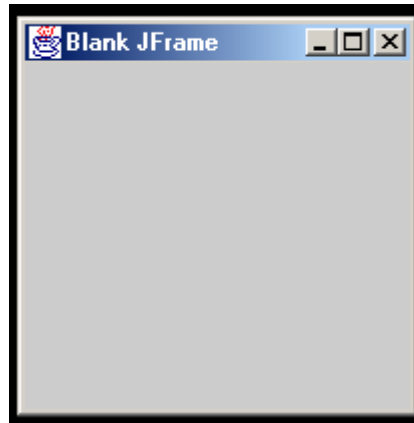


Fig.19.4 Output Screen for Program 19.1

If you try to close this window using close window icon, the window will disappear from the screen, but will not be closed. In DOS mode, press the CTRL+C key to close the window. In order to close the window, we include methods to capture window closing event and call **System.exit(0)**. In the following program 19.2, we show how to close the **JFrame** window by trapping window closing event.

Program 19.2

```
// This program creates a blank JFrame window.
// This JFrame window can be closed.
// somasundaramk@yahoo.com

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

// class to handle windowclosing event
class winclose
    extends WindowAdapter
    {
        public void windowClosing(WindowEvent we)
        {
            System.exit(0);
        }
    }

class JFrame2
    {
        public static void main(String args [])
        {
            JFrame f = new JFrame();
            winclose wc = new winclose();
            f.addWindowListener(wc);
            f.setTitle("Closable JFrame");
            f.setSize(200, 200);
            f.setVisible(true);
        }
    }
```

```

        f.setLocation(50, 100);
    }
}

```

The above program gives the following output:



Fig.19.5 Output Screen for Program 19.2

In program 19.2, the `winclose` class handles the **windowClosing** event. The window events are handled in `java.awt.event` package and hence it is imported. Initially, the frame window is created without a title. The title “Closable JFrame” is set using the **setTitle()** method. This title appears on the window border. The size of the window created is 200 x 200 pixel size. By default, this window will appear at (0,0) position of the screen. The **setVisible(true)** method makes the window visible. The window created is shifted to the new location (50,100) on the screen by the **setLocation()** method.

In the following program 19.3, the window closing event is handled in an anonymous inner class and is an improved version of program 19.2.

Program 19.3

```

// This program creates closable blank JFrame window
// using an anonymous inner class.
// somasundaramk@yahoo.com
import javax.swing.JFrame;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

class JFrame4
    extends JFrame
    {
        JFrame4(String title)
        {
            super(title);
            // using anonymous inner class

```



```
addWindowListener(new WindowAdapter()  
{  
    public void windowClosing(WindowEvent e)  
    {  
        System.exit(0);  
    }  
});  
setSize(200, 200);  
setVisible(true);  
setLocation(50, 100);  
}  
public static void main(String args [])  
{  
    new JFrame4("Closable JFrame");  
}  
}
```

Program 19.3 gives the same output as that of program 19.2.

19.3 JButton

The **JButton** is a concrete subclass of **AbstractButton**, which is a subclass of **JComponent**. The counterpart of **JButton** in AWT is **Button**. The **JButton** class is used to create labeled button components. When a button is clicked, an **ActionEvent** is created. Both mouse press and mouse release events can be processed separately. If an application wants to perform some action based on the button being pressed and released, it should implement the interface **ActionListener** and register the listener to receive events from the button. One of the advanced features of **JButton** is that an image can be used as a button.

19.3.1 Creating JButtons

Constructors used for creating a JButton component are:

JButton()

Constructs a button with no label

JButton(String label)

Constructs a button with a specified label

JButton(Icon i)

Constructs a button with the icon i as button

JButton(String label, Icon i)

Constructs a button with the icon i as button and the string as label

Methods

Some of the methods defined in a **JButton** class are :

`void addActionListener(ActionListener al)`

Adds the specified action listener to receive action event from this button

`String getActionCommand()`

Returns the command name of the action event fired by this button

`void removeActionListener(ActionListener al)`

Removes the action listener al so that it no longer receives action events from this button

`void processActionEvent(ActionEvent ae)`

Processes the action events occurring on this button by dispatching them to any action listener object

`void setText(String label)`

Sets the button's label to the specified string

`void getText()`

Returns the label of the button

`Icon getIcon()`

Returns the icon of the button

`void setIcon(Icon i)`

Sets the icon for this button

`void setRolloverIcon(Icon i)`

Sets the icon i as the rollover icon for this button

19.3.2 Creating JButtons on JFrame

As we have seen in the earlier section, **JFrame** is a top-level container. All the components can be added only on the content pane of **JFrame**. To get the content pane **getContentPane()** method is used, which returns a **Container** type object defined in java.awt package. The following program 19.4 shows how to add a **JButton** on a **JFrame**:

Program 19.4

```
// This program illustrates the use of JButton class.
// The Jbuttons are placed on JFrame window.
// somasundaramk@yahoo.com

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

```
class JFrmb1
    extends JFrame
    {
        JButton b1, b2, b3;
        Container con;
        JFrmb1(String s)
        {
            super(s);
            setSize(250, 150);
            con = getContentPane();
            b1 = new JButton("ZERO");
            b2 = new JButton("ONE");
            b3 = new JButton("TWO");
            con.add(b1);
            con.add(b2);
            con.add(b3);
            addWindowListener(new WindowAdapter()
            {
                public void windowClosing(WindowEvent e)
                {
                    System.exit(0);
                }
            });
        }
    }

class JFrmbut1
{
    public static void main(String args [])
    {
        JFrame fb = new JFrmb1("JFrame with JButton");
        fb.setVisible(true);
    }
}
```

The above program gives the following output:

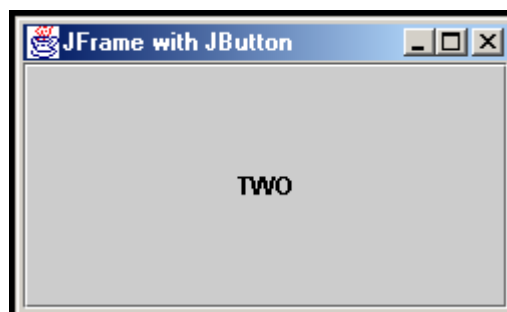


Fig.19.6 Output Screen for Program 19.4

The output screen in fig.19.6 for program 19.4 shows only one button, the last one, with label TWO displayed and not the first two buttons. It is because the default layout manager for JFrame is a border layout. A border layout

manager stacks one component over the other and hence only the last button is seen. To solve this problem, a layout manager, which arranges the components on the content pane, is to be set. Program 19.5 shows a flow layout manager set for the content pane. The components are arranged automatically in the window like the word flow in a text.

Program 19.5

```
// This program illustrates the use of JButton class.
// The Jbuttons are placed on a JFrame window.
// The JFrame is set with a FlowLayout manager.
// somasundaramk@yahoo.com

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

class JFrmb2
    extends JFrame
    {
        JButton b1, b2, b3;
        Container con;

        JFrmb2(String s)
        {
            super(s);
            setSize(250, 150);
            con = getContentPane();
            con.setLayout(new FlowLayout(FlowLayout.LEFT));
            b1 = new JButton("ZERO");
            b2 = new JButton("ONE");
            b3 = new JButton();
            b3.setText("TWO");
            con.add(b1);
            con.add(b2);
            con.add(b3);
            addWindowListener(new WindowAdapter()
                {
                    public void windowClosing(WindowEvent e)
                    {
                        System.exit(0);
                    }
                });
        }
    }

class JFrmbut2
    {
        public static void main(String args [])
        {
```

```
JFrame fb = new JFrmb2("JFrame with JButton");
fb.setVisible(true);
}
```

The output screen for the above program is given below:

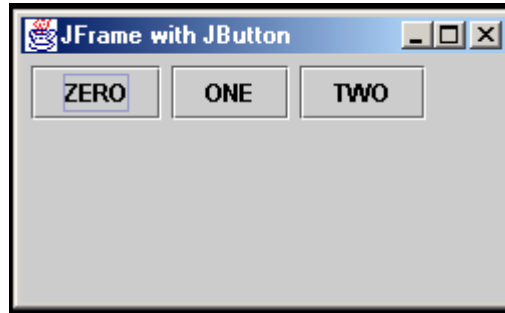


Fig.19.7 Output Screen for Program 19.5

JButtons are added on to the content pane in the order in which they are added using **add(JComponent comp)**. However, this order can be changed using the method **add(JComponent comp, int position)**. The following program 19.6 shows how to add the buttons in a different order:

Program 19.6

```
// This program illustrates the use of JButton class.
// The JButtons are placed on a JFrame window in a different
// order.
// The JFrame is set with a FlowLayout manager.
// somasundaramk@yahoo.com

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

class JFrmb3
    extends JFrame
    {
        JButton b1, b2, b3;
        Container con;
        JFrmb3(String s)
        {
            super(s);
            setSize(250, 150);
            con = getContentPane();
            con.setLayout(new FlowLayout(FlowLayout.LEFT));
            b1 = new JButton("ZERO");
            b2 = new JButton("ONE");
            b3 = new JButton();
        }
    }
```

```

        b3.setText("TWO");
        con.add(b1);
        con.add(b2, 0);
        con.add(b3, 1);
        addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent e)
            {
                System.exit(0);
            }
        });
    }
}

class JFrmbut3
{
    public static void main(String args [])
    {
        JFrame fb = new JFrmb3("JFrame with JButton");
        fb.setVisible(true);
    }
}

```

The above program gives the following output screen:

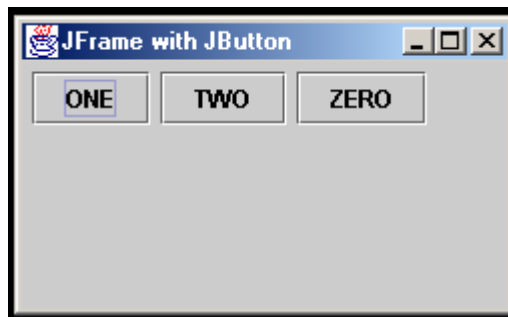


Fig.19.8 Output Screen for Program 19.6 Note the Order of Appearance of the JButtons.

Creating Icons as JButtons

JButton class has constructors to use images as buttons. An Icon type object is created using the **ImageIcon** class. The constructors of this class are:

ImageIcon (String filename)

Creates an Icon object with the image specified by the filename

ImageIcon(URL url)

Creates an Icon object with the image specified in the URL

The following program 19.7 illustrates the use of **Icon** and **JButton**:

Program 19.7

```
// This program illustrates the use of JButton with Icon.
// The JButtons are placed on a JFrame window.
// The JFrame is set with a FlowLayout manager.
// somasundaramk@yahoo.com

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

class JFrmimage
    extends JFrame
    {
        JButton b1, b2, b3;
        Icon mag, right;
        Container con;
        JFrmimage(String s)
        {
            super(s);
            setSize(350, 150);
            con = getContentPane();
            con.setLayout(new FlowLayout(FlowLayout.LEFT));
            mag = new ImageIcon("mag65.jpg");
            right = new ImageIcon("RIGHTt2.jpg");
            b1 = new JButton("Right", right);
            b2 = new JButton("ONE");
            b3 = new JButton("Magesh", mag);
            con.add(b1);
            con.add(b2);
            con.add(b3);
            addWindowListener(new WindowAdapter()
                {
                    public void windowClosing(WindowEvent e)
                    {
                        System.exit(0);
                    }
                });
        }
    }

class JFrmim
{
    public static void main(String args [])
    {
        JFrame fb = new JFrmimage("JFrame with Icon");
        fb.setVisible(true);
    }
}
```

The above program gives the following output:

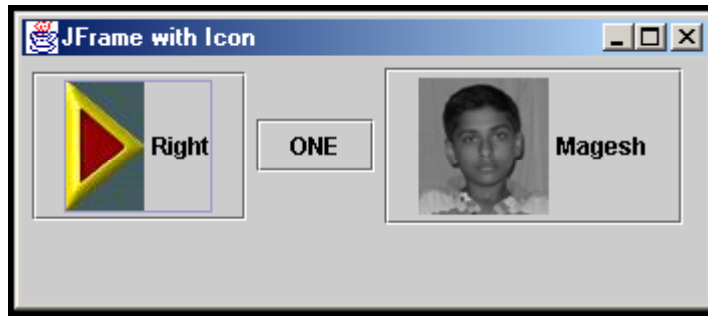


Fig.19.9 Output Screen for Program 19.7

19.3.3 Creating JButtons on JApplet

The **JApplet** class provides a top-level container on which buttons can be added in the same way as that on a **JFrame** window. In **JApplet** also the Container's content pane is to be obtained using `getContentPane()` method and `JComponents` added to it. The following program 19.8 shows the creation of `JButtons` on a **JApplet** window:

Program 19.8

```
// This program illustrates the use of a JButton class.
// The buttons are placed on a JApplet window.
// somasundaramk@yahoo.com

/*
<applet code = JApbut1 width = 300 height = 150>
</applet>
*/

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

public class JApbut1
    extends JApplet
    {
        JButton b1, b2, b3;
        Container con;
        Icon right;
        public void init()
        {
            con = getContentPane();
            con.setLayout(new FlowLayout());
            right = new ImageIcon("rightt2.jpg");
            b1 = new JButton("Right", right);
            b2 = new JButton("TWO");
            b3 = new JButton("THREE");
            con.add(b1);
        }
    }

```



```
con.add(b2);  
con.add(b3);  
}  
}
```

The above program gives the following output:

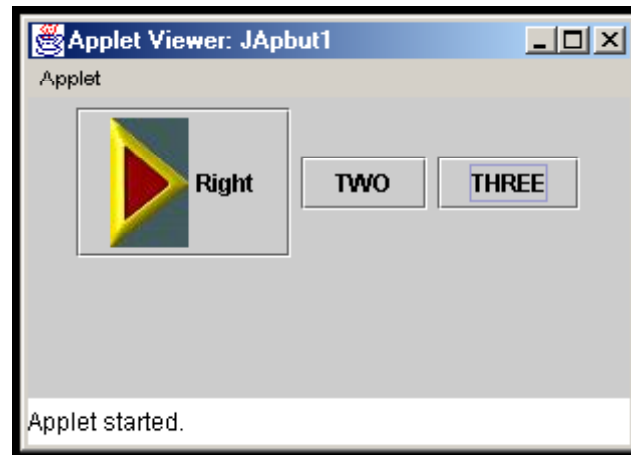


Fig.19.10 Output Screen for Program 19.8

19.3.4 Using JButton

Buttons act as an interface between the program and the user. The buttons when pressed are supposed to initiate some process related to the program being developed. In the following program 19.9, we show how to use `JButtons` in an application.

Two buttons with labels `Circle` and `Rectangle` are created and added on to a **JPanel**. **JPanel** is both a container and a component. **JPanel** is a window without border and title. **Graphics** object can be drawn on this **JPanel** using **paintComponent()** method. To draw any graphics, one must call the superclass method **super.paintComponent()**. **JPanel** implements a default **FlowLayout** manager.

Action listeners are added to the two buttons. When a button is pressed, the action is caught and processed in the **ActionPerformed()** method. The **getActionCommand()** method fetches the label of the button being pressed. When the button with label `Circle` is pressed, a circle is drawn and when the button with label `Rectangle` is pressed, a rectangle is drawn.

The **JPanel** is then attached to the content pane of a **JFrame** of the class `Upbut1`. The main method of the class `Jusbut1` creates the object that displays the user interface.

Program 19.9

```

// This program illustrates how to use JButtons in
// applications.
// The JButtons are placed on a JPanel component.
// The JPanel is then added to a JFrame window.
// somasundaramk@yahoo.com

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class JPbut
    extends JPanel
    implements ActionListener
    {
        JButton b1, b2;
        String cmd = null;

        JPbut()
        {
            setLayout(new FlowLayout(FlowLayout.LEFT));
            b1 = new JButton("Circle");
            b2 = new JButton("Rectangle");
            b1.addActionListener(this);
            b2.addActionListener(this);
            add(b1);
            add(b2);
        }

        public void actionPerformed(ActionEvent ae)
        {
            cmd = ae.getActionCommand();
            repaint();
        }

        public void paintComponent(Graphics gp)
        {
            super.paintComponent(gp);

            if (cmd == "Circle")
                gp.drawOval(20, 60, 75, 75);

            else if (cmd == "Rectangle")
                gp.drawRect(150, 60, 75, 75);
        }
    }

class Upbut1
    extends JFrame
    {
        Container con;
    }

```

```

Upbut1(String title)
{
    super(title);
    con = getContentPane();
    setSize(250, 200);
    con.add(new JPbut());
    addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent e)
            {
                System.exit(0);
            }
        });
}

class Juspbut1
{
    public static void main(String args [])
    {
        Upbut1 ufb = new Upbut1("Using JButton");
        ufb.setVisible(true);
    }
}

```

The above program gives the following output:

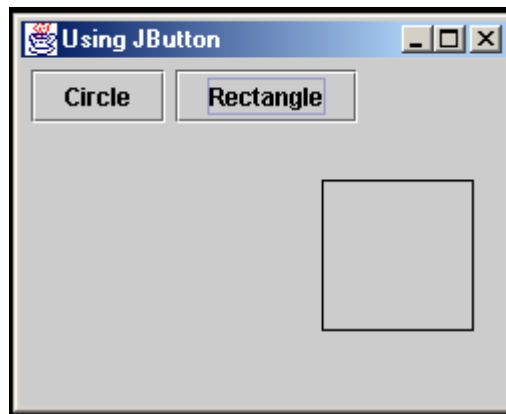


Fig.19.11 Output Screen for Program 19.9

19.4 JLabel

The **JLabel** class is a concrete subclass of **JComponent**. A **JLabel** object is a component for placing text or an icon or both. A **JLabel** displays a single line of read only text in a container. The text can be changed by the application and not by the user.

The **JLabel** has the following **int** type constants that indicate the alignment of the labels content:

JLabel.CENTER	JLabel.TOP
JLabel.LEFT	JLabel.BOTTOM
JLabel.RIGHT	

19.4.1 Creating JLabel

The labels are created using the following constructors:

JLabel()

Creates an empty label

JLabel(Icon i)

Creates a label using the icon i

JLabel(Icon i, int alignment)

Creates a label using the icon i; the icon is aligned as specified in the alignment. Possible values are JLabel.CENTER, JLabel.LEFT, JLabel.RIGHT.

JLabel(String str)

Creates a label with the specified string str

JLabel(String str, Icon i, int alignment)

Creates a label using the icon i, the label is set to the specified string str and with the specified alignment

JLabel(String str, int alignment)

Creates a label with the specified string str and with the specified alignment

Methods

The **JLabel** class has a number of methods. Some of them are:

int getHorizontalAlignment()

Returns the horizontal alignment for the label's content

Icon getIcon()

Returns the icon of the label

String getText()

Returns the text of the label

int getVerticalAlignment()

Returns the vertical alignment for the label's content

void setFont(Font font)

Sets the font for the label's text

```
void setHorizontalAlignment(int alignment)
    Sets the horizontal alignment for the label's content

void setIcon(Icon i)
    Sets the specified icon as the label's content

void setText(String str)
    Sets the specified string str as the label's content

void setVerticalAlignment(int alignment)
    Sets the vertical alignment for the label's content
```

19.4.2 Creating JLabel on JFrame

The following program 19.10 illustrates the creation of a **JLabel** on a **JFrame**:

Program 19.10

```
// This program illustrates the use of JLabel class.
// The JLabels are placed on JFrame window.
// somasundaramk@yahoo.com

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

class Labelframe
    extends JFrame
    {
        JLabel lbl1, lbl2, lbl3, lblicon;

        Labelframe(String s)
        {
            super(s);
            setSize(250, 150);
            Container con = getContentPane();
            con.setLayout(new FlowLayout(FlowLayout.LEFT));
            lbl1 = new JLabel("ZERO");
            lbl2 = new JLabel("ONE");
            lbl3 = new JLabel("TWO");
            Icon frut = new ImageIcon("fruits.jpg");
            lblicon = new JLabel("Fruits", frut, JLabel.LEFT);
            con.add(lbl1);
            con.add(lbl2);
            con.add(lbl3);
            con.add(lblicon);
            addWindowListener(new WindowAdapter()
                {
                    public void windowClosing(WindowEvent e)
```

```

        {
            System.exit(0);
        }
    });
}

class JFrmlbl
{
    public static void main(String args [])
    {
        JFrame fb = new Labelframe("JFrame with JLabel");
        fb.setVisible(true);
    }
}

```

The above program gives the following output:



Fig.19.12 Output Screen for Program 19.10

19.5 JToggleButton

The **JToggleButton** is a concrete subclass of **AbstractButton**. It is a superclass for **JCheckBox** and **JRadioButton** classes. Toggle buttons are two-state graphical components. They will be in selected(true) or deselected(false) state. The toggle buttons generate **ActionEvent**, **ChangeEvent** and **ItemEvent**.

19.5.1 Creating JToggleButton

The toggle buttons are created using the following constructors:

JToggleButton()

Creates a toggle button without a label; the toggle button is set to deselected state.

JToggleButton(Icon i)

Creates a toggle button using the icon; the toggle button is set to deselected state.

`JToggleButton(Icon i, boolean state)`

Creates a toggle button using the icon `i` and is set to the specified state.

`JToggleButton(String label)`

Creates a toggle button with the specified label; the toggle button is set to deselected state.

`JToggleButton(String label, boolean state)`

Creates a toggle button with the specified label and is set to the specified state.

`JToggleButton(String label, Icon i)`

Creates a toggle button with the specified label using the icon; the toggle button is set to deselected state.

`JToggleButton(String label, Icon i, boolean state)`

Creates a toggle button with the specified label using the icon `i` and is set to the specified state.

19.5.2 Adding JToggleButton on JApplet

The toggle buttons can be added in the **JApplet** in the same way as in **JFrame**. The following program 19.11 shows the creation of toggle buttons on a **JApplet**:

Program 19.11

```
// This program creates JToggleButtons and are
// added on to the JApplet.
// somasundaramk@yahoo.com
/*
<applet code = Togbutap width =200 height =200 >
</applet>
*/
import javax.swing.*;
import java.awt.*;
public class Togbutap
    extends JApplet
    {
        JToggleButton jtb1, jtb2, jtb3, jtbicon;
        Container conpan;
        public void init()
        {
            conpan = getContentPane();
            conpan.setLayout(new FlowLayout(FlowLayout.LEFT));
            Icon frut = new ImageIcon("fruits.jpg");
            jtb1 = new JToggleButton("Mango");
            jtb2 = new JToggleButton("Apple", true);
            jtb3 = new JToggleButton("Banana");
```

```

        jtbicon = new JToggleButton("Fruits", frut, true);
        conpan.add(jtb1);
        conpan.add(jtb2);
        conpan.add(jtb3);
        conpan.add(jtbicon);
    }
}

```

The above program gives the following output screen:



Fig.19.13 Output Screen for Program 19.11 Darkened buttons indicate that they are in selected state.

19.6 JCheckBox

JCheckBox is a subclass of **JToggleButton**, which is again a subclass of **AbstractButton**. A check box is a two-state graphical component that will be in either a selected (true) or a deselected (false) state. Unlike toggle button, a check box has visual display of the selected or deselected state. A selected state is indicated by a tick (✓) mark in a box and the deselected state by an empty box. A check box, when clicked, generates **ActionEvent**, **ChangeEvent** and **ItemEvent**.

19.6.1 Creating JCheckBox

The **JCheckBox** has the following constructors for creating check boxes:

JCheckBox()

Creates a check box with a blank label; the check box is set to deselected state.

JCheckBox(Icon i)

Creates a check box using the specified icon; the check box is set to deselected state.

`JCheckBox(Icon i, boolean state)`

Creates a check box using the specified icon; the check box is set to the specified state.

`JCheckBox(String str)`

Creates a check box with the label `str`; the check box is set to the deselected state.

`JCheckBox(String str, boolean state)`

Creates a check box with the label `str`; the check box is set to the specified state.

`JCheckBox(String str, Icon i)`

Creates a check box using the icon `i` with the label `str`; the check box is set to deselected state.

`JCheckBox(String str, Icon i, boolean state)`

Creates a check box using the icon `i` with the label `str`; the check box is set to the specified state.

Methods

The **JCheckBox** class has many methods. Some of them are given below:

`String getText()`

Returns the label of the check box

`void setText(String str)`

Sets the label of the check box to `str`

`Icon getIcon()`

Returns the icon of the check box

`void setIcon(Icon i)`

Sets the specified icon `i` as the check box

`boolean isSelected()`

Returns the state of the check box

`void setSelected(boolean state)`

Sets the check box to the specified state

19.6.2 Creating JCheckBox on JFrame

The following program 19.12 illustrates the creation of **JCheckBox** and adding them on a **JFrame**. Three **JCheckBoxes** with labels Mango, Apple and Banana are created and added on the content pane of the **JFrame**. The fourth one is a check box with an image and label Fruits.

Program 19.12

```

// This program creates JCheckBoxes and are
// added on the JFrame.
// somasundaramk@yahoo.com

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

class JChkbox
    extends JFrame
    {
        Container con;
        JCheckBox jbx1, jbx2, jbx3, jbxicon;

        JChkbox(String title)
        {
            super(title);
            // using anonymous inner class
            addWindowListener(new WindowAdapter()
                {
                    public void windowClosing(WindowEvent e)
                    {
                        System.exit(0);
                    }
                }
            );

            setSize(250, 150);
            Icon frut = new ImageIcon("fruits.jpg");
            jbx1 = new JCheckBox("Mango");
            jbx2 = new JCheckBox("Apple");
            jbx3 = new JCheckBox("Banana");
            jbxicon = new JCheckBox("Fruits", frut);
            con = getContentPane();
            con.setLayout(new FlowLayout(FlowLayout.LEFT));
            con.add(jbx1);
            con.add(jbx2);
            con.add(jbx3);
            con.add(jbxicon);
            setVisible(true);
        }
    }

class JCKbox1
    {
        public static void main(String args [])
        {
            new JChkbox("JCheckBox on JFrame");
        }
    }

```

The above program gives the following output:

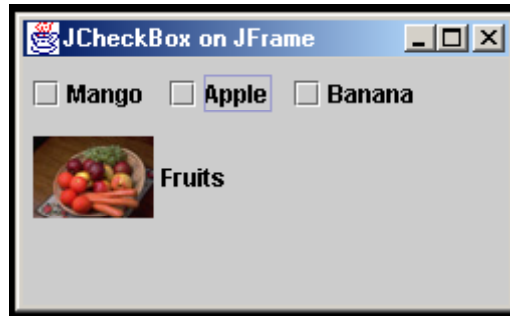


Fig.19.14 Output Screen for Program 19.12

19.6.3 Creating JCheckBox on JPanel

The check boxes can be added on a **JPanel** component, which is also a container. The **JPanel** is then added on to a **JFrame**. The following program 19.13 shows the creation of check boxes on the **JPanel**:

Program 19.13

```
// This program creates JCheckBoxes and they are
// added on to the JPanel. The JPanel is then added on
// to a JFrame.
// somasundaramk@yahoo.com

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

class JChkbox3
    extends JPanel
    {
        JCheckBox jbx1, jbx2, jbx3, jbxicon;
        JChkbox3()
        {
            Icon frut = new ImageIcon("fruits.jpg");
            jbx1 = new JCheckBox("Mango");
            jbx2 = new JCheckBox("Apple", true);
            jbx3 = new JCheckBox("Banana");
            jbxicon = new JCheckBox("Fruits", frut, true);
            setLayout(new FlowLayout(FlowLayout.LEFT));
            add(jbx1);
            add(jbx2);
            add(jbx3);
            add(jbxicon);
        }
    }

class JPchkbox
```

```

extends JFrame
{
    Container con;

    JPchkbox(String title)
    {
        super(title);
        con = getContentPane();
        con.add(new JChkbox3());
        setSize(250, 150);
        setVisible(true);

        // using anonymous inner class
        addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent e)
            {
                System.exit(0);
            }
        });
    }
}

class JChkbox3
{
    public static void main(String args [])
    {
        new JPchkbox("JCheckBox on JPanel");
    }
}

```

The above program gives the following output:



Fig.19.15 Output Screen for Program 19.13

19.6.4 Using JCheckBox

The check boxes can be used for any kind of application, in which a user is given an option to select any number of options out of several options given. In the following program 19.14, three choices are provided as check boxes. The **JLabel** is used to display the selections. The check boxes and label are created and are added on a **JFrame**.

Program 19.14

```
// This program creates JCheckBoxes and are
// added on to the JFrame.
// somasundaramk@yahoo.com

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

class JChkbox2
    extends JFrame
    implements ActionListener
    {
        Container con;
        JCheckBox jbx1, jbx2, jbx3;
        JLabel answer;
        boolean select1, select2, select3, select4;
        String cmd, selection;

        JChkbox2(String title)
        {
            super(title);
            // using anonymous inner class
            addWindowListener(new WindowAdapter()
                {
                    public void windowClosing(WindowEvent e)
                    {
                        System.exit(0);
                    }
                }
            );

            setSize(250, 150);
            jbx1 = new JCheckBox("Mango");
            jbx2 = new JCheckBox("Apple");
            jbx3 = new JCheckBox();
            answer = new JLabel("Nothing Selected");
            jbx3.setText("Banana");
            jbx1.addActionListener(this);
            jbx2.addActionListener(this);
            jbx3.addActionListener(this);
            con = getContentPane();
            con.setLayout(new FlowLayout(FlowLayout.LEFT));
            con.add(jbx1);
            con.add(jbx2);
            con.add(jbx3);
            con.add(answer);
            setVisible(true);
        }

        public void actionPerformed(ActionEvent ae)
        {

```

```
select1 = jbx1.isSelected();
select2 = jbx2.isSelected();
select3 = jbx3.isSelected();

if (select1 | select2 | select3)
{
    selection = "";

    if (select1)
        selection = "Mango";

    cmd = ae.getActionCommand();

    if (select2)
        selection = selection + ", Apple";

    if (select3)
        selection = selection + ", Banana";
}

if (selection != null)
{
    answer.setText(selection.trim() + "Selected");
    selection = null;
}

else
    answer.setText("Nothing Selected");
}

class JChkbox2
{
    public static void main(String args [])
    {
        new JChkbox2("JCheckBox on JFrame");
    }
}
```

The above program gives the following output:

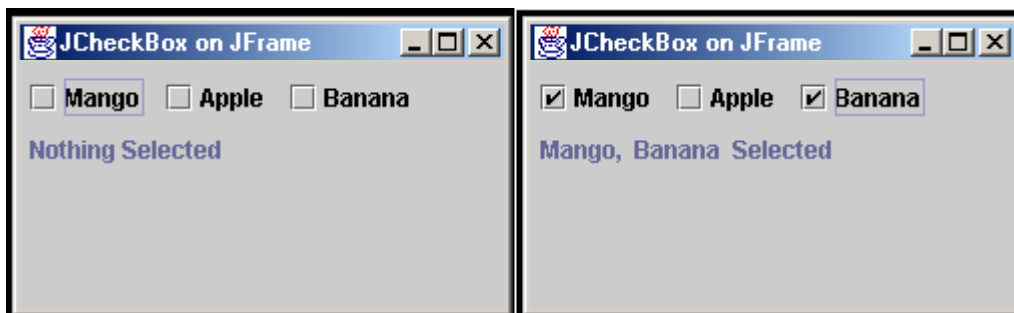


Fig.19.16 Output Screens for Program 19.14 The first window shows the initial display and the second on the right shows the screen after the check boxes are selected.



JToggleButton and JCheckBox are two-state buttons. A visual rectangle box with ✓ mark indicates that a JCheckBox is in a selected state. A blank box indicates that it is in a deselected state.

19.7 JRadioButton

Radio buttons are like check boxes. In a radio button, the selection is displayed in a round graphics. Radio buttons are generally used to represent a collection of mutually exclusive options. That is, out of several options, only one will be in selected (true) state and all the remaining are in deselected (false) state. The radio buttons are created using **JRadioButton** class, which is a subclass of **JToggleButton**. The **JRadioButton** must be placed in a button group. The button group is created using the **ButtonGroup** class, which has no argument constructor. After creating **JRadioButton**, the radio buttons are to be added to the **ButtonGroup** using **add()** method. **JRadioButton** generates **ActionEvent**, **ItemEvent** and **ChangeEvent**. If the radio buttons are not grouped using **ButtonGroup**, then each radio button will behave exactly like **JCheckBox**.

19.7.1 Creating JRadioButton

The radio buttons are created using the following constructors:

JRadioButton()

Creates a radio button without any label; the radio button is set to deselected state.

JRadioButton(Icon i)

Creates a radio button using the icon i; the radio button is set to deselected state.

JRadioButton(Icon i, boolean state)

Creates a radio button using the icon i; the radio button is set to the specified state.

JRadioButton(String str)

Creates a radio button with the string str as label; the radio button is set to deselected state.

JRadioButton(String str, boolean state)

Creates a radio button with the string str as label; the radio button is set to the specified state.

JRadioButton(String str, Icon i)

Creates a radio button using the icon i with the string str as label

JRadioButton(String str, Icon i, boolean state)

Creates a radio button using the icon i with the string str set as label; the radio button is set to the specified state.

19.7.2 Creating JRadioButton on JFrame

The radio buttons can be created using constructors and added to a container. In the following program 19.15, radio buttons are created and added to a **JFrame**. The radio buttons are grouped using the **ButtonGroup** class.

Program 19.15

```
// This program creates JRadioButtons and they are added on
// to the JFrame. All the radio buttons are grouped to bg.
// somasundaramk@yahoo.com

import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
class Radioframe
    extends JFrame
    {
        Container conpan;
        JRadioButton rbut1, rbut2, rbut3, rbuticon;

        Radioframe(String title)
        {
            super(title);
            // using anonymous inner class
            addWindowListener(new WindowAdapter()
                {
                    public void windowClosing(WindowEvent e)
                    {
                        System.exit(0);
                    }
                }
            ));

            setSize(250, 150);
            Icon frut = new ImageIcon("fruits.jpg");
            rbut1 = new JRadioButton("Mango");
            rbut2 = new JRadioButton("Apple", true);
            rbut3 = new JRadioButton("Banana");
            rbuticon = new JRadioButton("Fruits", frut);
            conpan = getContentPane();
            conpan.setLayout(new FlowLayout(FlowLayout.LEFT));
            conpan.add(rbut1);
            conpan.add(rbut2);
```



```

        conpan.add(rbut3);
        conpan.add(rbuticon);
        // create button group
        ButtonGroup bg = new ButtonGroup();
        bg.add(rbut1);
        bg.add(rbut2);
        bg.add(rbut3);
        bg.add(rbuticon);
        setVisible(true);
    }
}

class JRadio
{
    public static void main(String args [])
    {
        new Radioframe("JRadioButton on JFrame");
    }
}

```

The above program gives the following output:



Fig.19.17 Output Screens for Program 19.15 First window shows the initial display. The second window shows the status after selecting another radio button.

19.7.3 Using JRadioButton

The radio buttons, when grouped, can be used to select one option out of many options in a mutually exclusive form. As the **JRadioButton** fires **ItemEvent** when clicked, it can be used to detect the item state change and appropriate process can be started. In the following program 19.16, three options C++, Java and Pascal are given. The user is asked to identify which of them is not an OOP language. When the user selects one choice, the program responds by displaying whether the selection is correct or wrong. The radio buttons are created and added on a **JPanel**. The panel is then added to a **JFrame** and called in another class with **main()** method.

Program 19.16

```

// This program illustrates the use of JRadioButtons.
// The radio buttons are placed on JPanel.
// The JPanel is then added to a JFrame.
// somasundaramk@yahoo.com

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

class Usrcadiopanel
    extends JPanel
    implements ItemListener
{
    JRadioButton rb1, rb2, rb3;
    ButtonGroup bg;
    boolean state, flag = false;

    Usrcadiopanel()
    {
        bg = new ButtonGroup();
        rb1 = new JRadioButton("C++");
        rb2 = new JRadioButton("Java");
        rb3 = new JRadioButton("Pascal");
        add(rb1);
        add(rb2);
        add(rb3);
        bg.add(rb1);
        bg.add(rb2);
        bg.add(rb3);
        rb1.addItemListener(this);
        rb2.addItemListener(this);
        rb3.addItemListener(this);
    }

    public void itemStateChanged(ItemEvent ie)
    {
        flag = true;
        repaint();
    }

    public void paintComponent(Graphics gp)
    {
        super.paintComponent(gp);
        state = rb3.isSelected();
        gp.drawString("Which of the above is not an OOP
                                language?", 20, 70);

        if (flag)
        {
            if (state)
                gp.drawString("Yes, you are correct", 20, 110);
        }
    }
}

```

```

        else
            gp.drawString("Sorry, it is wrong", 20, 110);
        }
    }
}

class Radioframe
    extends JFrame
    {
        Container conpan;

        Radioframe(String str)
        {
            super(str);
            conpan = getContentPane();
            conpan.add(new Uusradiopanel());
            // using anonymous inner class
            addWindowListener(new WindowAdapter()
                {
                    public void windowClosing(WindowEvent e)
                    {
                        System.exit(0);
                    }
                });

            setSize(275, 150);
        }
    }

class Uusradpnl
    {
        public static void main(String args [])
        {
            JFrame radframe = new Radioframe("RadioButtons on
                                                JPanel");

            radframe.setVisible(true);
        }
    }

```

The above program gives the following output:

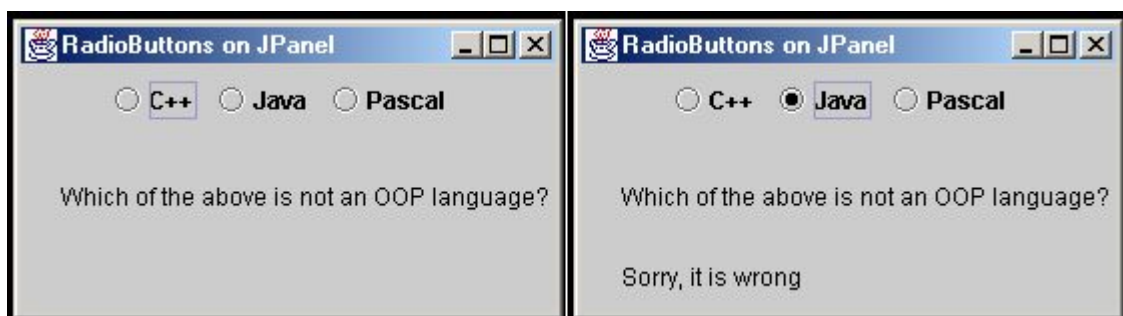


Fig.19.18 Output Screens for Program 19.16. The first window shows the initial display and the second one shows the display after the user has made a selection.



JRadioButton is like a **JCheckBox**. To use **JRadioButton** in mutually exclusive selection of one option out of many options, it is to be grouped using **ButtonGroup**. All **JRadioButtons** are to be added to the **ButtonGroup**.

19.8 JList

JList is a subclass of **JComponent**. **JList** creates a graphical display of a list of items and allows the user to select one or more items from the list. A list is used when the number of items for selection is large. Either strings or images can be elements of the list. Selection of an item is done by clicking on the item itself. A Swing list does not have a scroll bar to display the list. Hence, the list is to be placed inside a **JScrollPane()** object. A **JList** delegates the responsibilities of data handling, item selection and cell rendering to the interfaces **ListModel**, **ListSelectionModel** and **ListCellRenderer**. The text in a **JList** cannot be edited. Some basics of **JList** are given in this section.

19.8.1 Creating JList

A **JList** is created with the following constructors:

JList()

Creates an empty list

JList(Vector vec)

Creates a list with items specified in the Vector vec

JList(Object[] obj)

Creates a list with items specified in the object array

JList(ListModel lm)

Creates a list with items contained in the list model lm; the list model is an instance of **DefaultListModel**.

Methods

The **JList** class has a large number of inherited methods and some of its own. Some of them are :

void setVisibleRowCount(int c)

Sets the number of rows in the list to be displayed

void addListSelectionListener(ListSelectionListener ls)

Adds a list selection listener to this list

int getFirstVisibleIndex()
Returns the index of the topmost item that is visible

int getLastVisibleIndex()
Returns the index of the bottom item that is visible

int getSelectedIndex()
Returns the index of the first selected item

Object getSelectedValue()
Returns the topmost selected item

Object[] getSelectedValues()
Returns an array of all selected items

int getVisibleRowCount()
Returns the number of rows visible in the JList

19.8.2 Creating JList with an Array

A **JList** is created by passing a **Vector** or an array of **Object** as argument. In **JList**, there is no method for adding or removing items in the list. The following program 19.17 illustrates the creation of a **JList** with an array of **String**. First, two arrays are created. Then two lists, monthlist and countrylist, are created by passing the arrays as argument in **JList**. As **JList** has no scroll pane, the lists are then passed as argument to the **JScrollPane** constructor. The scroll pane enabled lists are then added to a **JPanel**. The **JPanel** is then attached to a **JFrame** and displayed using the **main()** method.

Program 19.17

```
/* This program illustrates how to create JList.
   JListsts are placed on the JScrollPane.
   The JScrollPane is then added to a JPanel.
   The JPanel is then added to a JFrame.
   somasundaramk@yahoo.com
*/

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

class Listpanel
    extends JPanel
    {
        JScrollPane monthsp, countrysp;
        JList monthlist, countrylist;
        ListModel monthmodel, countrymodel;
    }
```

```

Listpanel()
{
    // create array of String
    String monthname [] =
    {
        "January", "February", "March",
        "April", "May", "June", "July",
        "August", "September", "October"
    };

    String countryname [] =
    {
        "Australia", "Africa", "Bangladesh", "India",
        "Pakistan", "Singapore", "Russia", "USA"
    };

    // create JList with array as argument
    monthlist = new JList(monthname);
    countrylist = new JList(countryname);
    // set the number of visible rows
    monthlist.setVisibleRowCount(4);
    countrylist.setVisibleRowCount(3);

    // attach JScrollPane to the list
    monthsp = new JScrollPane(monthlist);
    countrysp = new JScrollPane(countrylist);

    // add the lists to the JPanel
    add(monthsp);
    add(countrysp);
    monthmodel = monthlist.getModel();
    countrymodel = countrylist.getModel();
}

public void paintComponent(Graphics gp)
{
    super.paintComponent(gp);
    gp.drawString("Monthlist", 45, 120);
    gp.drawString("Countrylist", 155, 120);
    gp.drawString("Size"+monthmodel.getSize(), 5, 140);
    gp.drawString(" "+countrymodel.getSize(), 160, 140);
}

}

class Scrollframe
    extends JFrame
    {
        Container conpan;

        Scrollframe(String str)
        {

```

```

super(str);
conpan = getContentPane();
conpan.add(new Listpanel());
// using anonymous inner class
addWindowListener(new WindowAdapter()
{
    public void windowClosing(WindowEvent e)
    {
        System.exit(0);
    }
});

setSize(250, 250);
setVisible(true);
}

class Jlstpnl
{
    public static void main(String args [])
    {
        new Scrollframe("JList on JPanel");
    }
}

```

The above program gives the following output:



Fig.19.19 Output Screen for Program 19.17



For creating a JList, first create an array or a Vector with the given items. Then construct a JList by passing them as argument in the constructor.

JList does not have a scroll pane. Create a JScrollPane by passing the list as the argument in its constructor.

19.8.3 DefaultListModel

Lists created in **JList** by passing an array or **Vector** do not have many methods for manipulating the items in the list. There is no method in the **JList** to add an item or remove an item in the list. The **DefaultListModel**, which is a concrete subclass of **AbstractListModel**, has methods for adding and removing items from a **JList**. This class implements an interface **ListModel**.

A list with **DefaultListModel** is created in the following steps:

- Create an instance of **DefaultListModel**.
- Add the items to this model.
- Create the **JList** by passing this model as argument in the constructor.

The following example shows the creation of a **JList** with **DefaultListModel** by following the above steps:

```
DefaultListModel monthmodel = new DefaultListModel();
monthmodel.addElement("January");
monthmodel.addElement("February");
monthmodel.addElement("March");
monthmodel.addElement("April");

JList monthlist = new JList(monthmodel);
```

A **JList** created using **DefaultListModel** has a large number of methods. Some of the methods defined for **DefaultListModel** are given below:

`void addElement(Object obj)`

Adds the given object at the end of the list

`void clear()`

Removes all elements from the list

`Object firstElement()`

Returns the first element in the list

`Object getElementAt(int index)`

Returns the list item at the specified index

`int getSize()`

Returns the number of items in the list

`void insertElementAt(Object obj, int index)`

Inserts the given object at the specified index

`boolean removeElement(Object obj)`

Removes the first occurrence of the object in the list

`void removeElementAt(int index)`

Removes the item at the specified index in the list

19.8.4 Creation of JList Using DefaultListModel

The following program 19.18 shows the creation of **JList** using **DefaultListModel**:

Program 19.18

```

/* This program illustrates how to create a JList
   using the DefaultListModel. List items are added to the
   DefaultListModel object. A JList is created with the
   model as an argument.
   somasundaramk@yahoo.com
*/
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class Listpanel
    extends JPanel
    {
        JScrollPane monthsp, countrysp;
        JList monthlist, countrylist;
        DefaultListModel dfmodel1, dfmodel2;
        public Listpanel()
        {
            // create DefaultListModel
            dfmodel1 = new DefaultListModel();
            dfmodel2 = new DefaultListModel();
            String monthname [] =
                {
                    "January", "February", "March",
                    "April", "May", "June", "July",
                    "August", "September", "October"
                };

            String countryname [] =
                {
                    "Australia", "Africa", "Bangladesh",
                    "India", "Pakistan", "Singapore",
                    "Russia", "USA"
                };

            // add elements to the model
            for (int i = 0; i < monthname.length; i++)

```

```

        dfmodel1.addElement(monthname[i]);

        for (int i = 0; i < countryname.length; i++)
            dfmodel2.addElement(countryname[i]);

        //create lists with the model as argument
        monthlist = new JList(dfmodel1);
        countrylist = new JList(dfmodel2);

        // set the number of visible rows in the display
        monthlist.setVisibleRowCount(4);
        countrylist.setVisibleRowCount(3);

        //insert the list in a scroll pane
        monthsp = new JScrollPane(monthlist);
        countrysp = new JScrollPane(countrylist);
        add(monthsp);
        add(countrysp);
    }
    public void paintComponent(Graphics gp)
    {
        super.paintComponent(gp);
        gp.drawString("Monthlist", 45,120);
        gp.drawString("Countrylist", 155,120);
        gp.drawString("Size"+ dfmodel1.getSize(),5,140);
        gp.drawString(" " + dfmodel2.getSize(),160,140);
        gp.drawString("First" + dfmodel1.firstElement(),
                    5,160);

        gp.drawString("item ", 5, 170);
        gp.drawString(""+dfmodel2.firstElement(),160,160);
        gp.drawString("Last"+dfmodel1.lastElement(),5,190);
        gp.drawString("item ", 5, 200);
        gp.drawString(""+dfmodel2.lastElement(),160,190);
    }
}

class DfListframe
    extends JFrame
    {
        Container conpan;
        DfListframe(String str)
        {
            super(str);
            conpan = getContentPane();
            setSize(250, 250);
            // using anonymous inner class
            addWindowListener(new WindowAdapter()
                {
                    public void windowClosing(WindowEvent e)
                    {
                        System.exit(0);
                    }
                }
            );
        }
    }

```

```

    });
    conpan.add(new Listpanel());
    setVisible(true);
}
}
class Dflstpn1
{
    public static void main(String args [])
    {
        new DfListframe("JList on JPanel");
    }
}

```

The above program gives the following output:



Fig.19.20 Output Screen for Program 19.18



When a JList is created using DefaultListModel, first add elements to the model and create the JList by passing the model as argument.

19.8.5 Using JList

A **JList** is used when the number of items for selection is large. In the following program 19.19, we create a **JList** and show how the list can be used for an application.

Program 19.19

```

/*This program illustrates how to use JList in an
application.
JList is placed on the JScrollPane.
The JScrollPane is then added to a JPanel.
The JPanel is then added to a JFrame.

```

```

    somasundaramk@yahoo.com
*/

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

class Listpanel
    extends JPanel
    implements ListSelectionListener
    {
        JScrollPane monthsp;
        JList monthlist, source;
        int monthdays [] = new int[10];
        String selected;
        ListModel lstmodel;
        int selectindex;

    public Listpanel()
        {
            String monthname [] = {"January", "February", "March",
                                   "April", "May", "June", "July", "August",
                                   "September", "October" };

            int mdays [] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31 };

            for (int i = 0; i < mdays.length; i++)
                monthdays[i] = mdays[i];
            monthlist = new JList(monthname);
            monthlist.setVisibleRowCount(4);
            monthlist.addListSelectionListener(this);
            monthsp = new JScrollPane(monthlist);
            add(monthsp);
            lstmodel = monthlist.getModel();
        }

    public void valueChanged(ListSelectionEvent lse)
        {
            source = (JList)lse.getSource();
            selected = (String)source.getSelectedValue();
            selectindex = source.getSelectedIndex();
            repaint();
        }

    public void paintComponent(Graphics gp)
        {
            super.paintComponent(gp);
            gp.drawString("Size of the list : " +
                          lstmodel.getSize(), 20, 130);

            if (selected != null)
                {
                    gp.drawString("Selected month : " + selected,
                                  20, 150);
                }
        }
    }

```

```

        gp.drawString("No of days in this month : " +
                      monthdays[selectindex], 20, 170);
        gp.drawString("Selected index : " +
                      selectindex, 20, 190);
    }
}

class Listframe
    extends JFrame
    {
        Container conpan;
        Listframe(String str)
        {
            super(str);
            conpan = getContentPane();
            setSize(250, 250);
            // using anonymous inner class
            addWindowListener(new WindowAdapter()
            {
                public void windowClosing(WindowEvent e)
                {
                    System.exit(0);
                }
            });
            conpan.add(new Listpanel());
            setVisible(true);
        }
    }

class Uslstpnl
    {
        public static void main(String args [])
        {
            new Listframe("JList on JPanel");
        }
    }

```

The above program gives the following output:

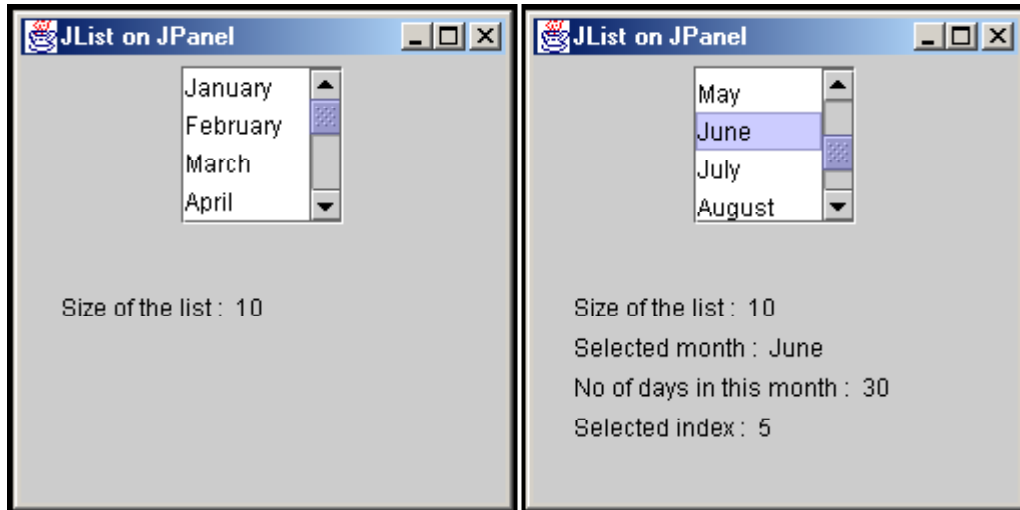


Fig.19.21 Output Screens for Program 19.19. First window shows the initial screen and the second window shows the output after a selection is made in the list.

In the program 19.19, `monthlist`, an instance of **JList**, is created by passing the string array as the argument for the constructor. A scroll pane is attached to the list. Whenever a selection is made by clicking on the list, some process related to the selected item is to be carried out. To detect the list selection, the method **addListSelectionListener()** is used to register the `monthlist`. When a list selection is made by an user, a **ListSelectionEvent** is created. This event is passed to the listener `ListPanel` which contains the `monthlist`. The **ListSelectionEvent** is processed by the **valueChanged()** method defined in the interface **ListSelectionListener**. Hence, the class `ListPanel` implements the interface **ListSelectionListener**. The **ListSelectionEvent** is defined in the **javax.swing.event** package. The **getSource()** method returns the source object which generated the event. The **getSize()** method finds the number of elements in the list. This method is defined only in the **ListModel** class and not in **JList**. Hence, **getModel()** method is called on the `monthlist` and the `lstmodel`, which is an object of type **ListModel**.

19.9 JScrollBar

JScrollBar is a subclass of **JComponent**. Scroll bars are visual components that can be used to bring a section of a large area of text, image or other items to view by scrolling. The scroll bars are available in two orientations, horizontal and vertical. The two orientations are defined by two constants:

JScrollBar.HORIZONTAL
JScrollBar.VERTICAL

Though a better facility for scrolling is available in **JScrollPane**, on certain occasions, manual adjustment is needed for scrolling. **JScrollBar** is used in such applications. The scroll bar has three visual parts. Unit increment and decrement are represented by two arrow marks. A slider is available in between the two arrow marks, which can be dragged in forward and backward directions to select a view for display. In the space between slider and unit increment lies the block increment area. Similarly, in between slider and unit decrement lies the block decrement area. The distance between minimum visible position and maximum visible position is called visible area. Usually, it is between 0-100 and does not represent the pixel value. The distance between the minimum value and maximum value of a large area is called a range.

19.9.1 Creating JScrollBar

The constructors used for creating a scroll bar are:

`JScrollBar()`

Creates a scroll bar in the vertical orientation

`JScrollBar(int Orientation)`

Creates a scroll bar in the specified orientation

The Orientation can either be `JScrollBar.HORIZONTAL` or `JScrollBar.VERTICAL`.

`JScrollBar(int Orientation, int scrollpos, int visible, int minimum, int maximum)`

Creates a scroll bar in the specified orientation; the initial scroll position is specified by scrollpos, visible indicates the visual area of the scroll bar, minimum indicate the minimum position value and maximum indicates the maximum position value of the scroll bar.

Methods

Some of the methods defined in **JScrollBar** are:

`void addAdjustmentListener(AdjustmentListener al)`

Adds adjustment listener to this component

`int getBlockDecrement()`

Returns the amount of scroll bar units that give the distance through which the slider moves when block decrement is clicked

`int getBlockIncrement()`

Returns the amount of scroll bar units that give the distance through which the slider moves when block increment is clicked

`int getMaximum()`

Returns the scroll bar's maximum value

```

int getMinimum()
    Returns the scroll bar's minimum value

int getOrientation()
    Returns the orientation of the scroll bar

int getUnitIncrement(int orientation)
    Returns the amount of the slider which will be incremented when the
    scroll bar's increment/decrement arrow is clicked

int getValue()
    Returns the current value of the position of the slider

void setMaximum(int max)
    Sets the scroll bar's maximum value in scroll bar unit to the specified
    value

void setMinimum(int min)
    Sets the scroll bar's minimum value, in scroll bar unit, to the specified value

void setOrientation(int orientation)
    Sets the orientation of the scroll bar

void setUnitIncrement(int inc)
    Sets the amount of slider should move in scroll bar units when the
    increment/decrement arrow is clicked

```

19.9.2 Creating JScrollBar on JPanel

The following program 19.20 illustrates the creation of one scroll bar in horizontal orientation and another one in vertical orientation. Note the slider position of the horizontal scroll bar is being shifted from the starting point by 50 scroll bar units as specified in the constructor.

Program 19.20

```

// This program shows how to create JScrollBars
// and add them on to a JPanel and add the JPanel
// to a JFrame window.
// somasundaramk@yahoo.com

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

class Scrollbarpanel
    extends JPanel
    {
        JScrollBar sbh, sbv;

```



```

Scrollbarpanel()
{
    sbh = new JScrollBar(JScrollBar.HORIZONTAL, 50,
                        50, 0, 250);
    sbv = new JScrollBar(JScrollBar.VERTICAL, 0, 50,
                        0, 200);

    add(sbh);
    add(sbv);
}

public void paintComponent(Graphics gp)
{
    super.paintComponent(gp);
    gp.drawString("Demo for scroll bars", 10, 150);
}
}

class Scrollpanecontainer
extends JFrame
{
    Scrollpanecontainer(String str)
    {
        super(str);
        setSize(250, 200);
        addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent we)
            {
                System.exit(0);
            }
        });

        Container con = getContentPane();
        con.add(new Scrollbarpanel());
    }
}

public class JScrlbar
{
    public static void main(String args [])
    {
        JFrame jf = new Scrollpanecontainer("JScrollBar on
                                           JPanel");

        jf.setVisible(true);
    }
}

```

The above program gives the following output screen:



Fig.19.22 Output Screen for Program 19.20

19.10 JScrollPane

JScrollPane is a subclass of **JComponent**. A **JScrollPane** is a visual component that helps to scroll around a large-sized item that is too big to be seen all at once. A **JScrollPane** provides a horizontal scroll bar and a vertical scroll bar automatically. A **JScrollPane** is a better choice to view a large item in a window than using a **JScrollBar**.

A schematic drawing of the **JScrollPane** visual component in relation to the large item to be scrolled is given in fig.19.23.

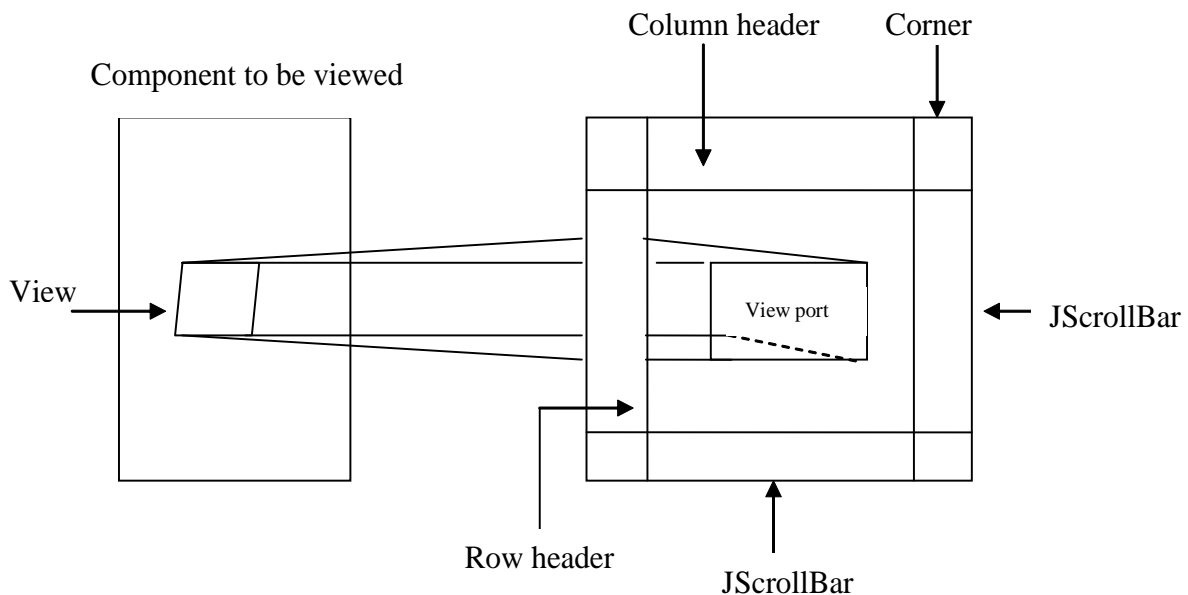


Fig.19.23 Schematic Drawing of JScrollPane

A section of the component that is to be viewed is called a view. The display area managed by the **JScrollPane** on the screen is called a view port. There are four corners. Each corner is identified by the following int type constants defined in the interface **ScrollPaneConstants**:

ScrollPaneConstants.UPPER_LEFT_CORNER
ScrollPaneConstants.UPPER_RIGHT_CORNER
ScrollPaneConstants.LOWER_LEFT_CORNER
ScrollPaneConstants.LOWER_RIGHT_CORNER

The top border is called column header and the left border is called row header. The right border represents an automatically adjustable vertical **JScrollBar**. The border at the bottom represents an automatically adjustable horizontal **JScrollBar**. Whether a horizontal scroll bar is needed or not is defined by the following int type constants called **HorizontalScrollBarPolicy**:

ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED
ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER
ScrollPaneConstants.HORIZONTAL_SCROLLBAR_ALWAYS

Similarly, whether a vertical scroll bar is needed or not is defined by the following int type constants called **VerticalScrollBarPolicy**:

ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED
ScrollPaneConstants.VERTICAL_SCROLLBAR_NEVER
ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS

Both scroll bars by default assume **SCROLLBAR_AS_NEEDED**.

19.10.1 Creating JScrollPane

The scroll panes are created using the following constructors:

JScrollPane()

Creates a scroll pane without any component

JScrollPane(int vconstant, int hconstant)

Creates a scroll pane with scroll bars as specified in the vertical scroll bar constant vconstant and horizontal scroll bar constant hconstant

JScrollPane(Component comp)

Creates a scroll pane for the specified component

JScrollPane(Component Comp, int vconstant, int hconstant)

Creates a scroll pane for the component comp with scroll bars as specified in the vertical scroll bar constant vconstant and horizontal scroll bar constant hconstant

Methods

The **JScrollPane** class has a number of methods. Some of them are :

Component getCorner(String corner)

Returns the component, if any, at the specified corner

`JScrollBar getHorizontalScrollBar()`

Returns the horizontal scroll bar

`int getHorizontalScrollBarPolicy()`

Returns the constant that tells whether the horizontal scroll bar will be displayed as needed, always or never

`JScrollBar getVerticalScrollBar()`

Returns the vertical scroll bar

`int getVerticalScrollBarPolicy()`

Returns the constant that tells whether the vertical scroll bar will be displayed as needed, always or never

`void setColumnHeaderView(Component comp)`

Sets the specified component as the column header view port

`void setCorner(String corner, Component comp)`

Sets the specified component to be put at the specified corner

`void setHorizontalScrollBarPolicy(int hconstant)`

Sets the hconstant that specifies whether the horizontal scroll bar will be displayed as needed, always or never

`void setRowHeaderView(Component comp)`

Sets the specified component as the row header view port

`void setVerticalScrollBarPolicy(int vconstant)`

Sets the vconstant that specifies whether the vertical scroll bar will be displayed as needed, always or never

19.10.2 Creating JScrollPane on JPanel

In the following program 19.21, we show how to create a **JScrollPane** on a **JPanel**. The source component we used is an image. First an image icon is created using an image “vegetable.jpg”. It is then used to create a **JLabel** type object picholder. This **JLabel** is then used as component for the **JScrollPane**. This scroll pane is then added to a **JFrame** window and displayed.

Program 19.21

```
// This program shows how to create a JScrollPane
// and add it on a JFrame window.
// somasundaramk@yahoo.com

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

```

class Scrollpanframe
    extends JFrame
    {
        JScrollPane jspan;
        JLabel picholder;
        Icon veg;
        Container conpan;

        Scrollpanframe(String str)
        {
            super(str);
            conpan = getContentPane();
            //create an image icon
            veg = new ImageIcon("vegetable.jpg");
            // create a JLabel using the image
            picholder = new JLabel(veg);

            // insert the JLabel inside a JScrollPane
            jspan = new JScrollPane(picholder);
            conpan.add(jspan);
            setSize(200, 250);
            addWindowListener(new WindowAdapter()
                {
                    public void windowClosing(WindowEvent we)
                    {
                        System.exit(0);
                    }
                });
        }
    }

public class JScrlpan1
{
    public static void main(String args [])
    {
        JFrame jsp = new Scrollpanframe("JScrollPane on a
                                           JFrame");

        jsp.setVisible(true);
    }
}

```

The above program gives the following output screen:



Fig.19.24 Output Screens for Program 19.21. The first one is the initial window and the second is after scrolling the horizontal and vertical scroll bars by the user.

We will give another illustration in program 19.22 for creating **JScrollPane**. In this program, the source component is an image “vegetable.jpg”, which is to be viewed in a scroll pane. After creating the image icon veg, it is used to create a **JLabel** type object picholder. This **JLabel** component is passed as an argument to create a **JScrollPane**. A text is created for display in the column header. Using the **setColumnHeaderView()** the text is set for the scroll pane. Similarly an image “BARV.jpg” is used to create an image and set as row header, using the method **setRowHeaderView()**. The upper left corner and lower right corner are set with text “UL” and “LR” respectively using the **setCorner()** method.

Program 19.22

```
/* This program shows how to create a JScrollPane
   and add it on to a JFrame window.
   Column header, row header and two corner
   components are also set for the scroll pane.
   somasundaramk@yahoo.com
*/

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

class Scrollpanframe2
    extends JFrame
    {
        JScrollPane jspan;
        JLabel picholder;
        Icon veg;
        Container conpan;
```

```

Scrollpanframe2(String str)
{
    super(str);
    conpan = getContentPane();

    //create an image icon
    veg = new ImageIcon("vegetable.jpg");

    // create a JLabel using the image
    picholder = new JLabel(veg);

    // insert the JLabel inside a JScrollPane
    jspan = new JScrollPane(picholder);

    // create a title string for column header
    JLabel colhead = new JLabel("Text in Column
                                Header");

    jspan.setColumnHeaderView(colhead);
    //create an image for row header
    Icon vbar = new ImageIcon("BARV.jpg");
    JLabel rowhead = new JLabel(vbar);
    jspan.setRowHeaderView(rowhead);
    // adding corner components
    JLabel ulc = new JLabel("UL");
    jspan.setCorner(
        ScrollPaneConstants.UPPER_LEFT_CORNER, ulc);
    JLabel lrc = new JLabel("LR");
    jspan.setCorner(
        ScrollPaneConstants.LOWER_RIGHT_CORNER,
        lrc);

    conpan.add(jspan);
    setSize(200, 250);
    addWindowListener(new WindowAdapter()
    {
        public void windowClosing(WindowEvent we)
        {
            System.exit(0);
        }
    });
}

public class JScrlpan2
{
    public static void main(String args [])
    {
        JFrame jsp = new Scrollpanframe2("JScrollPane on a
                                           JFrame");

        jsp.setVisible(true);
    }
}

```

The above program gives the following output screen:



Fig.19.25 Output screens for Program 19.22. The first window is the initial display and the second is after the scroll bars are moved by the user. Note the text in the column header, which is left aligned has moved out of view in the second window.



A JScrollPane is used to view a part of a large text or image. When the view port of the JScrollPane is larger than the source, the whole source is displayed. When the view port is smaller than the area of the source, the horizontal and vertical scroll bars are added automatically to view a part of the source.

19.11 JPasswordField

Swing provides different classes to handle the text of different styles using the Swing's model-view concept. Basically, Swing's text component deals with two distinct types of text. One text component deals with simple text of one font and one color of text. The other type is a styled text with multiple fonts and multiple colors. The simple type texts are dealt by **JTextField**, **JPasswordField** and **JTextArea** classes. The styled texts are handled in **JEditorPane** and **JTextPane** classes.

JTextField is a subclass of **JTextComponent**, which is a subclass of **JComponent**. A **JTextField** object is a visual component that can display one line of editable text of one font and color at a time. The text is placed inside a box. The alignment of the text is defined by the following int type constants:

JTextField.LEFT JTextField.CENTER

JTextField.RIGHT

19.11.1 Creating JTextField

The text field component is created using the following constructors:

`JTextField()`

Creates a new text field; the text is set to null and the number of columns is set to 0.

`JTextField(String text)`

Creates a new text field with the specified string as text

`JTextField(int columns)`

Creates a new empty text field with the specified number of columns; each column can accommodate a character.

`JTextField(String text, int columns)`

Creates a new text field with the specified string as text and with a width to display the specified number of columns

Methods

The **JTextField** has a number of inherited methods and some of its own. Some of the methods to handle the **JTextField** object are given below:

`void addActionListener(ActionListener al)`

Adds the action listener to receive action events from this text field

`int getColumns()`

Returns the number of columns set for this text field

`void removeActionListener(ActionListener al)`

Removes the action listener from this text field

`void setColumns(int columns)`

Sets the specified number of columns for this text field

`void setText(String text)`

Sets the specified text as the text for this text field

`String getText()`

Returns the text contained in this text field

`String getSelectedText()`

Returns the selected text contained in this text field

`void setEditable(boolean edit)`

Sets the text field to editable (true) or not editable (false)

19.11.2 Creating JTextField on JPanel

In the following program 19.23, we will show how to create text fields using different constructors and to add them on a **JPanel**. This **JPanel** is then attached to a **JFrame** window and displayed.

Program 19.23

```

/* This program shows how to create JTextFields and to
   add them on to JPanel. This JPanel is then attached
   to a JFrame window.
   somasundaramk@yahoo.com
*/

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

class JTxtfldpanel
    extends JPanel
    {
        JTextField txtf1, txtf2, txtf3;

        JTxtfldpanel()
        {
            setLayout(new FlowLayout(FlowLayout.LEFT));
            txtf1 = new JTextField();
            txtf2 = new JTextField("Demo Text Field");
            txtf3 = new JTextField("Second Text Field", 20);
            add(txtf1);
            add(txtf2);
            add(txtf3);
        }
    }

class JTxtfldframe
    extends JFrame
    {
        Container conpan;

        JTxtfldframe(String title)
        {
            super(title);

            conpan = getContentPane();
            conpan.add(new JTxtfldpanel());
            addWindowListener(new WindowAdapter()
                {
                    public void windowClosing(WindowEvent e)
                    {
                        System.exit(0);
                    }
                }
            );
        }
    }

```

```

        }
    });

    setSize(200, 200);
    setVisible(true);
}

class JTxtfp1
{
    public static void main(String args [])
    {
        new JTxtfldframe("JTextField on JPanel");
    }
}

```

The above program gives the following output screen:

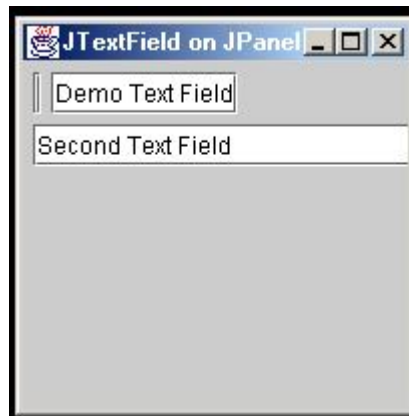


Fig.19.26 Output Screen for Program 19.23

The first text field in fig.19.26 is created without any text using `JTextField()` constructor. The second text field is constructed using a text. The required column width is automatically adjusted to fit the text. The third text field is created with a text and set with 20 character column width. Hence, the unused columns are displayed empty.

19.11.3 Creating JTextField on JApplet

In the following program 19.24, we will show how to create text fields and add them on to a **JApplet** window. Three text fields, as in program 19.23, are created and attached to the content pane of the **JApplet**.

Program 19.24

```

// This program shows how to create JTextFields
// and add them on to a JApplet window.
// somasundaramk@yahoo.com

```

```

import java.awt.*;
import javax.swing.*;
import java.applet.*;
/*
<applet code =JTxtfda1 width =250 height = 150>
</applet>
*/
public class JTxtfda1
    extends JApplet
    {
        JTextField txtf1, txtf2, txtf3;
        Container conpan;
        public void init()
        {
            conpan = getContentPane();
            conpan.setLayout(new FlowLayout(FlowLayout.LEFT));
            txtf1 = new JTextField();
            txtf2 = new JTextField("Demo Text Field");
            txtf3 = new JTextField("Second Text Field", 20);
            conpan.add(txtf1);
            conpan.add(txtf2);
            conpan.add(txtf3);
        }
    }

```

The above program gives the following output screen:

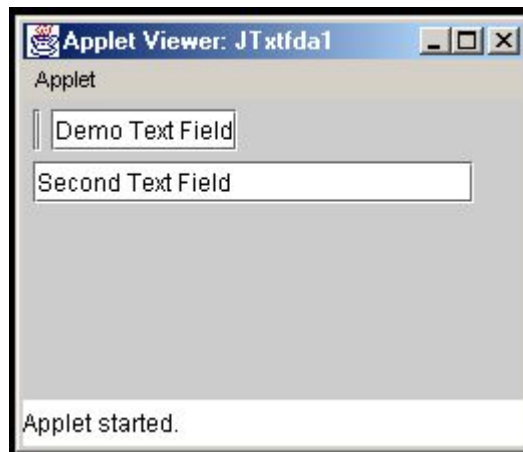


Fig.19.27 Output Screen for Program 19.24

19.11.4 Creating JTextField on JFrame

In the following program 19.25, we will show how to create text fields and add them on a **JFrame** window.

Program 19.25

```
/*This program shows how to create JTextFields
   and add them on to JFrame window.
   somasundaramk@yahoo.com
*/

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

class Txtfdfr1
    extends JFrame
    {
        JTextField txtf1, txtf2, txtf3;
        Container conpan;

        Txtfdfr1(String str)
        {
            super(str);
            conpan = getContentPane();
            conpan.setLayout(new FlowLayout(FlowLayout.LEFT));
            txtf1 = new JTextField();
            txtf2 = new JTextField("Demo Text Field");
            txtf3 = new JTextField("Second Text Field", 20);
            conpan.add(txtf1);
            conpan.add(txtf2);
            conpan.add(txtf3);
            addWindowListener(new WindowAdapter()
                {
                    public void windowClosing(WindowEvent we)
                    {
                        System.exit(0);
                    }
                });
            setSize(250, 200);
        }
    }

class JTxtf1
{
    public static void main(String args [])
    {
        Txtfdfr1 tffr = new Txtfdfr1("JFrame with
                                     JTextField");

        tffr.setVisible(true);
    }
}
```

The above program gives the following output screen:

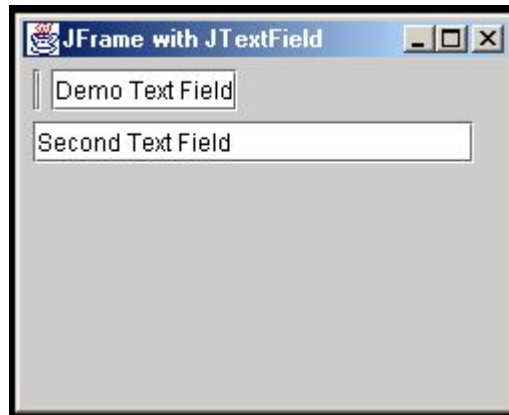


Fig.19.28 Output Screen for Program 19.25

19.11.5 Using JTextField

So far, we have seen how to create text field. Now, we will show how text field can be used in an application. In the following program 19.26, we develop an application for e-banking. A screen initially displays fields for the customer to key in their bank details. Each customer will have to fill in his details to withdraw money from the bank. A customer can withdraw a maximum of Rs.5000 in one withdrawal. Withdrawal of more than Rs 5000 is not allowed in this system.

In the text field, after filling in the entries, the Return key is to be pressed. Pressing of the Enter key generates **actionEvent**. This is captured and a response is given. A user has to delete the default display, enter the relevant data and press the Enter key, which generates an **actionEvent**. This action event is processed by the **actionPerformed** method.

Program 19.26

```
/* This program shows how to create JTextField
   and add them on to JPanel window
   and use the JTextField in an application.

   somasundaramk@yahoo.com
*/
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
class Txtfdaplc
    extends JPanel
    implements ActionListener
{
    JTextField nam, account, pinnumb, amount;
```

```

JLabel name, acno, pin, amt, welcome;
String names, accounts, pinnumbs, amounts, warning="";

Txtfdaplcn()
{
    welcome = new JLabel("          Welcome to SOMSON
                        e-Bank          ");
    name = new JLabel(" Name ");
    acno = new JLabel("Ac. No ");
    pin = new JLabel("PIN no ");
    amt = new JLabel("Amount ");
    nam = new JTextField("Type in your name", 25);
    account = new JTextField("Type in your Ac. No", 25);
    pinnumb = new JTextField(25);
    amount = new JTextField("0000", 25);
    add(welcome);
    add(name);
    add(nam);
    add(acno);
    add(account);
    add(pin);
    add(pinnumb);
    add(amt);
    add(amount);
    nam.addActionListener(this);
    account.addActionListener(this);
    pinnumb.addActionListener(this);
    amount.addActionListener(this);
}

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

public void paintComponent(Graphics gp)
{
    super.paintComponent(gp);
    names = nam.getText();
    accounts = account.getText();
    pinnumbs = pinnumb.getText();
    amounts = amount.getText();
    int amountint = Integer.parseInt(amounts);
    if (amountint > 5000)
    {
        warning = "Sorry, you cannot draw more than
                  Rs.5000";

        amount.setText("0000");
        amounts = "0000";
        gp.drawString(warning, 10, 320);
    }
}

```

```

        }

        gp.drawString("Name : " + names, 10, 230);
        gp.drawString("Your Ac.No :" + accounts, 10, 250);
        gp.drawString("Your PIN.No : " + pinnumbs, 10, 270);
        gp.drawString("Amount to draw:" + amounts, 10, 290);
    }
}

class Txtfdfr1
    extends JFrame
    {
        Container conpan;

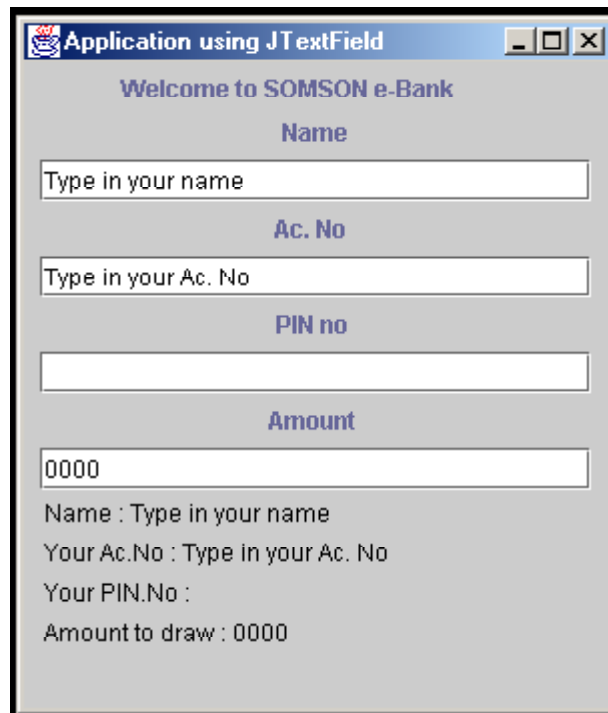
        Txtfdfr1(String str)
        {
            super(str);
            conpan = getContentPane();
            setSize(300, 350);
            conpan.add(new Txtfdaplcn());
            addWindowListener(new WindowAdapter()
                {
                    public void windowClosing(WindowEvent we)
                    {
                        System.exit(0);
                    }
                }
            ));
        }
    }

class JTxtfapl
    {
        public static void main(String args [])
        {
            JFrame txtaplcn = new Txtfdfr1("Application using
                                           JPasswordField");

            txtaplcn.setVisible(true);
        }
    }

```

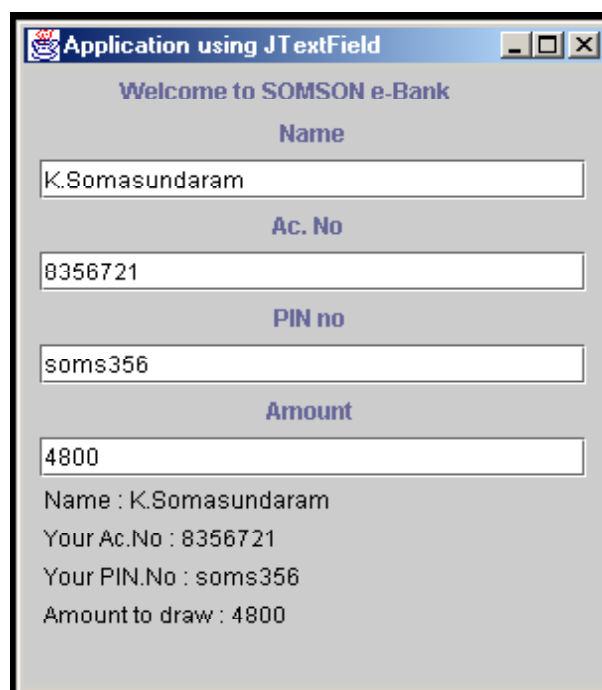
The above program gives the following output screen when executed:



The image shows a Java Swing window titled "Application using JTextField". Inside the window, the text "Welcome to SOMSON e-Bank" is displayed at the top. Below this, there are four labels: "Name", "Ac. No", "PIN no", and "Amount". Each label is followed by a text input field. The "Name" field contains the placeholder text "Type in your name". The "Ac. No" field contains the placeholder text "Type in your Ac. No". The "PIN no" field is empty. The "Amount" field contains the placeholder text "0000". At the bottom of the window, there is a summary of the input fields: "Name : Type in your name", "Your Ac.No : Type in your Ac. No", "Your PIN.No :", and "Amount to draw : 0000".

Fig.19.29 a) The Initial Output Screen for Program 19.26

The output screen after the items are filled up by the customer is given below:



The image shows the same Java Swing window as in Fig.19.29 a), but with the input fields filled with user data. The "Name" field now contains "K.Somasundaram". The "Ac. No" field now contains "8356721". The "PIN no" field now contains "soms356". The "Amount" field now contains "4800". At the bottom of the window, the summary text has been updated to reflect the new input: "Name : K.Somasundaram", "Your Ac.No : 8356721", "Your PIN.No : soms356", and "Amount to draw : 4800".

Fig.19.29 b) Output Screen After Editing the Fields for Program 19.26

19.12 JPasswordField

The **JPasswordField** creates a display for text field similar to **JTextField**. The only difference is that when text is displayed, the actual characters are replaced by * characters. This password field is useful where the text typed by a user is not to be seen by other people. **JPasswordField** is a subclass of **JTextComponent**. Like in **JTextField**, only one line of text, with one font and color can be used.

19.12.1 Creating JPasswordField

The constructors used for creating password field are given below:

JPasswordField()

Creates an empty password field

JPasswordField(int columns)

Creates an empty password field with the specified number of columns

JPasswordField(String text)

Creates a password field with the specified text

JPasswordField(String text, int columns)

Creates a password field with the specified text and specified number of columns

Methods

JPasswordField has a number of inherited methods and some of its own. Some of the methods defined for **JPasswordField** are:

boolean echoCharIsSet()

Returns true if an echo character has been set

char getEchoChar()

Returns the echo character set for this field

void setEchoChar(char c)

Sets the specified character as echo character for this field

19.12.2 Creating JPasswordField on JFrame

The **JPasswordField** is used whenever entries made into a text field are to be made in a secured way so that other persons viewing the screen are not able to read the characters typed. As an illustration, we will show in program 19.27, the creation of **JPasswordField** and placing it on **JFrame** window. By default, the echo character that is displayed as a substitute for the typed character is *. However, if a user wants a different echo character, it can be set using the **setEchoChar()** method.

Program 19.27

```

/* This program shows how to create a JPasswordField
   and add it on a JFrame window.
   somasundaramk@yahoo.com
*/

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

class JPswdframe
    extends JFrame
    {
        JPasswordField usrpswd;
        JTextField name;
        JLabel namelbl, pswdlbl;
        Container conpan;

        JPswdframe(String str)
        {
            super(str);
            conpan = getContentPane();
            setSize(300, 200);
            conpan.setLayout(new FlowLayout(FlowLayout.LEFT));
            namelbl = new JLabel("Name : ");
            pswdlbl = new JLabel("Password : ");
            name = new JTextField("Type your name", 20);
            usrpswd = new JPasswordField(10);
            conpan.add(namelbl);
            conpan.add(name);
            conpan.add(pswdlbl);
            conpan.add(usrpswd);
            addWindowListener(new WindowAdapter()
                {
                    public void windowClosing(WindowEvent we)
                    {
                        System.exit(0);
                    }
                });
        }
    }

class JPwdf
    {
        public static void main(String args [])
        {
            JFrame pswdf = new JPswdframe("JPassword on
                                           JFrame");

            pswdf.setVisible(true);
        }
    }

```

The above program gives the following output:



Fig.19.30 Output Screens for Program 19.27 The first window shows the initial screen display. The second one shows the screen after the entry has been made.

19.13 JTextArea

JTextArea is a subclass of **JTextComponent**. A **JTextArea** component displays multiple lines of text in one color and with one font. Text area text are displayed as such in the defined window. There is no scroll bar to view the text. If the text is large, then a **JScrollPane** has to be created using the text area component.

19.13.1 Creating JTextArea

JTextArea objects are created using the following constructors:

`JTextArea()`

Creates an empty text area

`JTextArea(int row, in column)`

Creates an empty text area with the specified number of rows and columns that are visible

`JTextArea(String text, int row, int column)`

Creates a text area with the specified text and with the specified rows and columns

Methods

A large number of methods are defined in **JTextArea** class. Some of them are :

`void append(String text)`

Adds the specified text at the end of the text area

`void copy()`

Copies the selected text into the system clipboard

`void cut()`

Cuts the selected text into the system clipboard; the selected text is removed from the text area.

`int getCaretPosition()`

Returns the current caret(cursor) position inside the text area

`int getColumns()`

Returns the number of columns set for the text area

`int getLineCount()`

Returns the number of lines in the text area

`boolean getLineWrap()`

Returns true if line wrap has been set for the text area, otherwise false

`int getRows()`

Returns the number of rows set for the text area

`String getSelectedText()`

Returns the selected text

`int getSelectionEnd()`

Returns the index next to the last character of the selected text

`int getSelectionStart()`

Returns the index of the first character of the selected text

`String getText()`

Returns the entire text of the text area

`void insert(String str, int pos)`

Inserts the specified string at the specified position pos

`boolean isEditable()`

Returns the boolean indicating whether the text area is editable or not

`void setLineWrap(boolean b)`

Sets the line wrap for the text area as specified by the boolean; wrapping is done as per the policy determined by the method `setWrapStyleWord`.

`void paste()`

Pastes the string from system clipboard at the current cursor position

`void replaceRange(String str, int start, int end)`

Replaces the string specified in the text area from start to end with the specified string

`void replaceSelection(String str)`

Replaces the selected text with the specified string

`void setEditable(boolean b)`

Sets the text to the editable or non-editable mode as specified by the boolean value

`void setSelectionEnd(int end)`

Sets the index at which the selection should end

`void setSelectionStart(int start)`

Sets the index at which the selection should start

`void setText(String text)`

Sets the text for the text area

`void setWrapStyleWord(boolean b)`

Sets the wrapping style; if set to true, line wrapping is done at the end of word boundaries. If set to false, line wrapping occurs at character boundaries.



JTextArea does not provide scroll bar. Hence, a JTextArea is to be manually inserted into a JScrollPane.

19.13.2 Creating JTextArea on JPanel

The following program 19.28 shows the creation of a **JTextArea** and placing it on a **JPanel**. The text area created has been set to display 6 rows of text with 25 character width. A **JScrollPane** is then attached to it so that the text area can be scrolled in a window. To illustrate the **append()** method, two additional strings are added to the text area created. The text in the text area is made to wrap at the end of each line of text using **setLineWrap()** method. The **setWrapStyleWord(true)** tells that the wrapping is to be done at the word boundaries. When the boolean is set to false, wrapping will occur at character boundaries.

Program 19.28

```
/* This program shows how to create a JTextArea
   and add it on a JPanel window.
   somasundaramk@yahoo.com
*/

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

class Txtapanel
    extends JPanel
    {
        JTextArea txta;
```

String testtext = "This is a demo text used in a JTextArea object. Text area can hold multiple lines with one font and one color, unlike the JTextField, which can contain only one line of text. Text area is useful for displaying a large text. Text area does not support ScrollPane. Hence, if need be, the text area object has to be passed into a ScrollPane constructor";

```

    Txtapanel()
    {
        Font fnt = new Font("Courier", Font.BOLD, 18);
        txta = new JTextArea(testtext, 6, 25);
        txta.setWrapStyleWord(true);
        txta.setLineWrap(true);
        txta.append("\nSecond text");
        txta.append("\nThird text ");
        txta.setFont(fnt);
        JScrollPane txtasp = new JScrollPane(txta);
        add(txtasp);
    }
}

class Txtaframe
    extends JFrame
    {
        Container conpan;

        Txtaframe(String str)
        {
            super(str);
            conpan = getContentPane();
            setSize(320, 200);
            conpan.add(new Txtapanel());
            addWindowListener(new WindowAdapter()
            {
                public void windowClosing(WindowEvent we)
                {
                    System.exit(0);
                }
            });
        }
    }

class JTxtap
{
    public static void main(String args [])
    {
        JFrame txtaf = new Txtaframe("JTextArea on JPanel");
        txtaf.setVisible(true);
    }
}

```

The above program gives the following output screen:

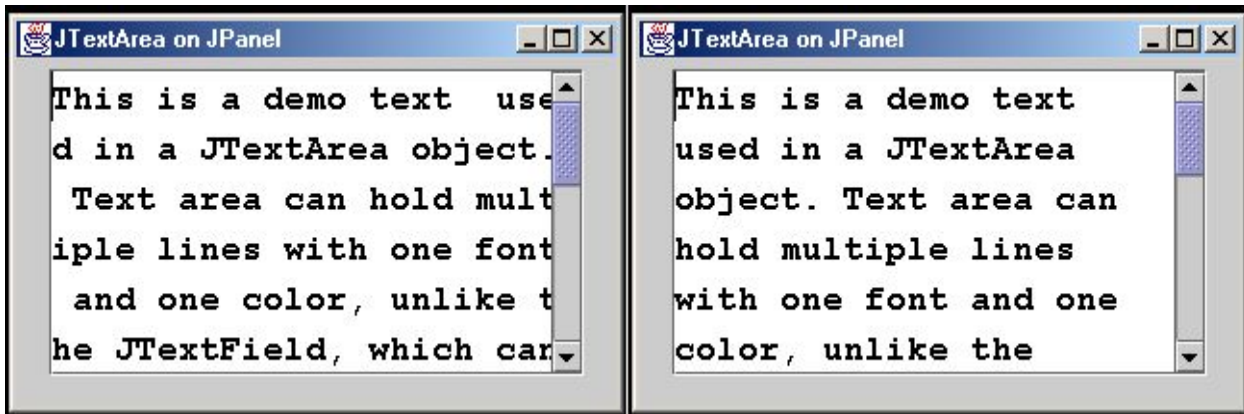


Fig.19.31 Output Screens for Program 19.28. The first window shows the line wrapping at character boundaries and the second shows the line wrapping at word boundaries.

19.13.3 Using JTextArea in Applications

In this section, we discuss and give an application program that makes use of **JTextArea**. **JTextArea** has a model-view structure, which is the basic concept in all Swing user-interface components. There is a **Document** interface for storing all attributes for the text. There are a large number of actions that can be done on text area and are defined in the class **EditorKit**. Text area is used for various applications. Various actions like selecting a text, cutting, copying, pasting, appending and inserting are possible. In the following program 19.29, we show how to select a particular string in the **JTextArea** component. The cursor is termed as caret. There is an interface **CaretListener**, which has an abstract method **caretUpdate()**, which is to be given a concrete method. To capture **CaretEvent**, the **addCaretListener()** method is called on the **JTextArea** object. We made use of the caret event to update the selections made in the text area.

Program 19.29

```
/* This program shows how to create a JTextArea
   and use it in an application.
   somasundaramk@yahoo.com
*/
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

class Txtaappanel
    extends JPanel
    implements CaretListener
{
```



```
JTextArea txta, txtsource;  
int caretindex = -1;  
String seltxt;  
String testtext = "This is a demo text  used in a  
JTextArea object. Text area can hold multiple lines with one  
font and one color, unlike the JTextField, which can contain  
only one line of text. Text area is useful for displaying a  
large text. Text area does not support ScrollPane. Hence, if  
need be, the text area object has to be passed into a  
ScrollPane constructor";
```

```
Txtaappanel()  
{  
    Font fnt = new Font("Courier", Font.BOLD, 18);  
    // create a JTextArea object  
    txta = new JTextArea(testtext, 6, 25);  
  
    // set the line wrapping is to be done with word  
    // boundary  
    txta.setWrapStyleWord(true);  
  
    // set line wrapping  
    txta.setLineWrap(true);  
  
    //add another line of text  
    txta.append("\nSecond text");  
    txta.append("\nThird  text ");  
  
    // set font for JTextArea  
    txta.setFont(fnt);  
  
    // add caret(cursor) listener. The JPanel is the  
    // listener  
    txta.addCaretListener(this);  
    JScrollPane txtasp = new JScrollPane(txta);  
    add(txtasp);  
}  
// whenever caret position changes the caretUpdate  
// is called  
public void caretUpdate(CaretEvent ce)  
{  
    txtsource = (JTextArea)ce.getSource();  
    seltxt = txtsource.getSelectedText();  
    caretindex = txtsource.getCaretPosition();  
    repaint();  
}  
  
public void paintComponent(Graphics gp)  
{  
    super.paintComponent(gp);
```

```

        int row = txta.getRows();
        boolean iswrap = txta.getLineWrap();
        String wrapped = iswrap ? "Yes" : "NO";
        gp.drawString("Line Wrapped:" + wrapped, 20, 200);
        gp.drawString("Rows:" + row, 20, 220);

        if (seltxt != null)
            gp.drawString("Selected String : " + seltxt,
                          20, 240);

        else
            gp.drawString(" No Text  Selected ", 20, 240);

        if (caretindex >= 0)
            gp.drawString("Caret Position : " +
                          caretindex, 20, 260);

        else
            gp.drawString("Caret Not Inside the Text
                          Area", 20, 260);
    }
}

class Txtaapframe
    extends JFrame
    {
        Container conpan;

        Txtaapframe(String str)
        {
            super(str);
            conpan = getContentPane();
            setSize(320, 300);
            conpan.add(new Txtaappanel());
            addWindowListener(new WindowAdapter()
            {
                public void windowClosing(WindowEvent we)
                {
                    System.exit(0);
                }
            });
        }
    }

class JTxtaap
    {
        public static void main(String args [])
        {
            JFrame txtaf = new Txtaapframe("Using JTextArea");
            txtaf.setVisible(true);
        }
    }

```

The above program gives the following output screen:

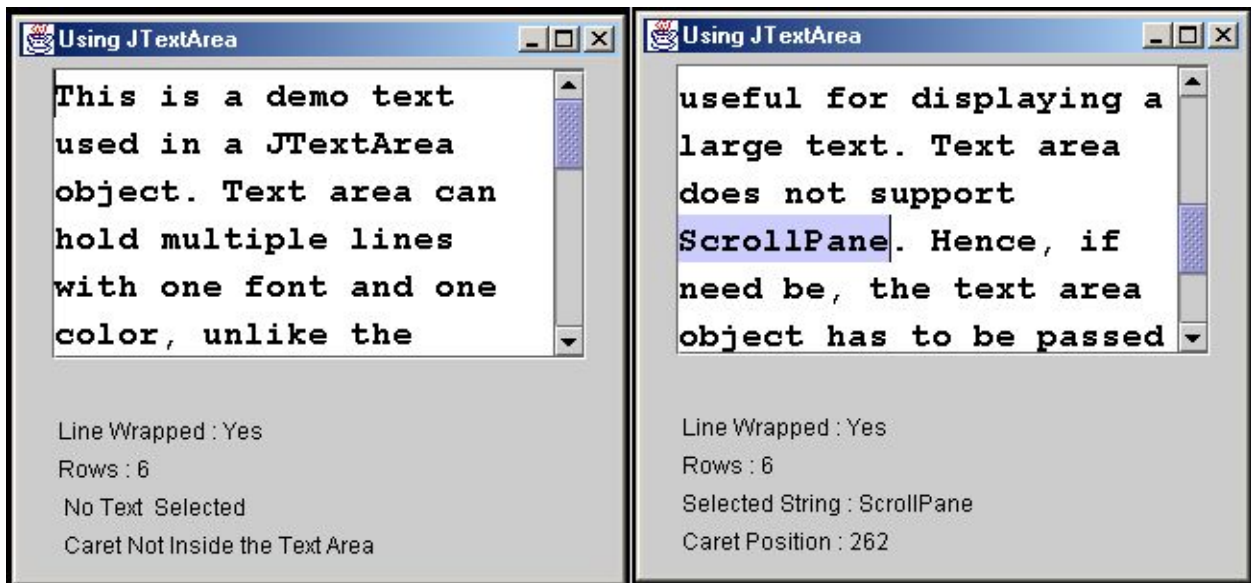


Fig.19.32 Output Screens for Program 19.29. The first is the unedited initial window. The second shows when a string is selected in the text area.

19.14 JComboBox

JComboBox is a subclass of **JComponent**. A combo box is a visual Swing graphical component that gives a popup list when clicked. It is the combination of **JList** and **JTextField**. In combo box, only one item is visible at a time. A popup menu displays the choices a user can select from. In **JList**, the items cannot be edited, but in combo box, the items can be edited by setting the **JComboBox** editable. **JComboBox** has a model-view structure. It has **DefaultComboBoxModel** class, which can be used to add more flexible methods to **JComboBox**.

19.14.1 Creating JComboBox

Combo boxes can be created using constructors defined in **JComboBox** class. Some of them are :

JComboBox()

Creates an empty combo box

JComboBox(ComboBoxModel model)

Creates a combo box using the items defined in the specified model

JComboBox(Object[] array)

Creates a combo box taking the items from the specified object array

JComboBox(Vector vec)

Creates a combo box taking the items from the specified Vector

Methods

JComboBox class has a number of methods. Some of them are:

void actionPerformed(ActionEvent ae)

This method is to be implemented when addActionListener is used

void addActionListener(ActionListener al)

Adds an action listener to the combo box

void addItem(Object obj)

Adds the specified object to the list

void addItemListener(ItemListener il)

Adds an item listener to the combo box

String getActionCommand()

Returns the action command

Object getItemAt(int index)

Returns item at the specified index in the list

int getItemCount()

Returns the number of items in the list

int getSelectedIndex()

Returns the number of items in the list

object getSelectedItem()

Returns the currently selected item

boolean isEditable()

Returns a boolean specifying whether the combobox items are editable or not

void removeAllItems()

Removes all items from the list

void removeItem(Object obj)

Removes the specified item from the list

void removeItemAt(int index)

Removes the item at the specified index from the list

void setActionCommand(String cmd)

Sets the specified string as the action command for the combo box

void setEditable(boolean edit)

Sets the boolean value indicating whether the items in the list are editable or not

```
void setEditor(ComboBoxEditor editor)
    Sets an editor for the combo box

void setModel(ComboBoxModel model)
    Sets a model for the combo box
```

19.14.2 Adding JComboBox on JPanel

JComboBox objects can be added on containers like **JApplet** window, **JFrame**, **JPanel** and other heavy-weight containers. In the following program 19.30, we will show how to create **JComboBox** objects and add them on to **JPanel**. We create two **String** arrays, monthname and countryname. They are then passed as argument to the **JComboBox** constructor to create two combo boxes. One of the combo boxes is made editable using **setEditable()** method. The two combo boxes are then added to a **JPanel**. The **JPanel** is then added to a **JFrame**. The **JFrame** is displayed through a **main()** method in another class.

Program 19.30

```
/*This program illustrates how to create JComboBox.
   JComboBoxes are then added to a JPanel.
   The JPanel is then added to a JFrame.
   somasundaramk@yahoo.com
*/

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

class Combopanel
    extends JPanel
    {
        JComboBox monthcombo, countrycombo;
        Combopanel()
        {
            // create array of String
            String monthname [] =
                {"January", "February", "March",
                "April", "May", "June", "July",
                "August", "September", "October"};

            String countryname [] =
                {"Australia", "Africa", "Bangladesh", "India",
                "Pakistan", "Singapore", "Russia", "USA"};

            // create JComboBox with array as argument
            monthcombo = new JComboBox(monthname);
            countrycombo = new JComboBox(countryname);
            countrycombo.setEditable(true);
```

```
        // add the combo boxes to the JPanel
        add(monthcombo);
        add(countrycombo);
    }
}

class Comboframe
    extends JFrame
    {
        Container conpan;

        Comboframe(String str)
        {
            super(str);
            conpan = getContentPane();
            conpan.add(new Combopanel());
            // using anonymous inner class
            addWindowListener(new WindowAdapter()
                {
                    public void windowClosing(WindowEvent e)
                    {
                        System.exit(0);
                    }
                }
            ));

            setSize(250, 250);
            setVisible(true);
        }
    }

class JCmbop
    {
        public static void main(String args [])
        {
            new Comboframe("JComboBox on JPanel");
        }
    }
```

The above program gives the following screen output:

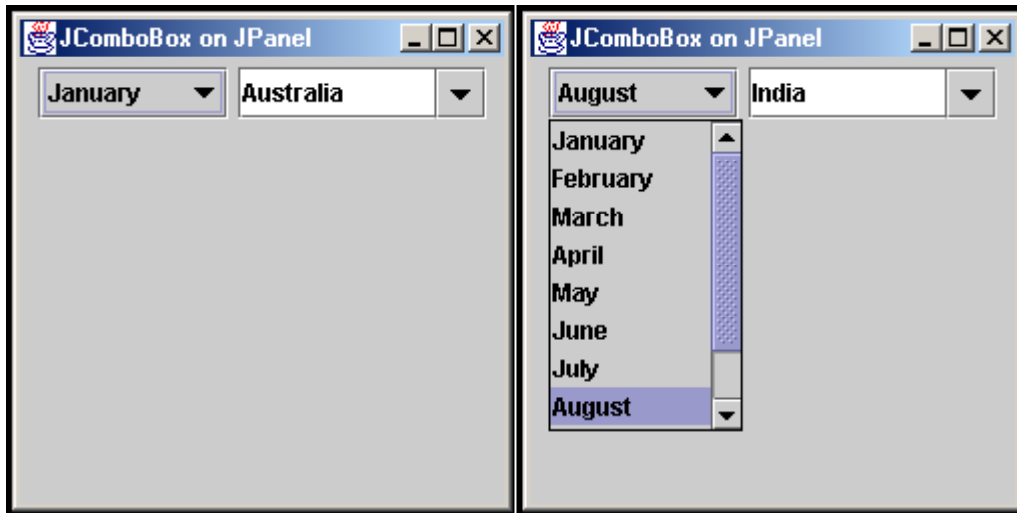


Fig.19.33 Output Screens for Program 19.30. The first window shows the initial display. The second window shows the screen after the users selection. Note the country display is in white, indicating that it is editable.



JComboBox is like the combination of JList and JTextField. Items in a JList cannot be edited, but items in a JComboBox can be set to editable.

19.14.3 Using JComboBox

In this section, we will discuss how to make use of a **JComboBox** in real-life problems with an example. The problem is to display the list of months and ask the user to select a month. The selected month is to be identified and displayed separately. Many applications require this kind of a situation. Displaying the selected month is an illustration of how to capture the selected item. Once the selection is identified, appropriate action can be carried out. In the following program 19.31, we will create a **JComboBox** containing the names of months. A **JLabel** object with a text instructing the user to select a month is created. The **JComboBox** is attached with an **ItemListener**. Whenever there is a change of state in the **JComboBox**, the **itemStateChanged()** method is executed. This method identifies the source of event. From the source, the selected item and its corresponding index are obtained. The **repaint()** method calls the **paintComponent()** method and updates the screen display.

Program 19.31

```

/* This program illustrates how to use JComboBox.

   somasundaramk@yahoo.com
*/

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

class Combopanel
    extends JPanel
    implements ItemListener
{
    JComboBox monthcombo;
    JComboBox combosource;
    String selectitem;
    int selectindex;

    public Combopanel()
    {
        String monthname [] =
            { "January", "February", "March",
              "April", "May", "June", "July", "August",
              "September", "October", "November", "December" };

        monthcombo = new JComboBox();

        // add elements to the model
        for (int i = 0; i < monthname.length; i++)
            monthcombo.addItem(monthname[i]);
        monthcombo.addItemListener(this);
        JLabel month = new JLabel("Select a month ");
        add(month);
        add(monthcombo);
    }

    public void itemStateChanged(ItemEvent ie)
    {
        combosource = (JComboBox)ie.getSource();
        selectitem = (String)combosource.getSelectedItem();

        selectindex = combosource.getSelectedIndex();
        repaint();
    }

    public void paintComponent(Graphics gp)
    {
        super.paintComponent(gp);

        if (combosource != null)
            {

```



```

        gp.drawString("No. of months in the list : "
            + combosource.getItemCount(), 5, 140);
        gp.drawString("Selected item      : " +
            selectitem, 5, 160);
        gp.drawString("Item index    : " +
            selectindex, 5, 180);
    }

    else
        gp.drawString("No selection is made", 5, 160);
    }
}

class Dfcomboframe
    extends JFrame
    {
        Container conpan;
        Dfcomboframe(String str)
        {
            super(str);
            conpan = getContentPane();
            setSize(250, 250);
            // using anonymous inner class
            addWindowListener(new WindowAdapter()
            {
                public void windowClosing(WindowEvent e)
                {
                    System.exit(0);
                }
            });
            conpan.add(new Combopanel());
            setVisible(true);
        }
    }

class Jcmbopl
    {
        public static void main(String args [])
        {
            new Dfcomboframe("Using JComboBox ");
        }
    }

```

The above program gives the following output screen:

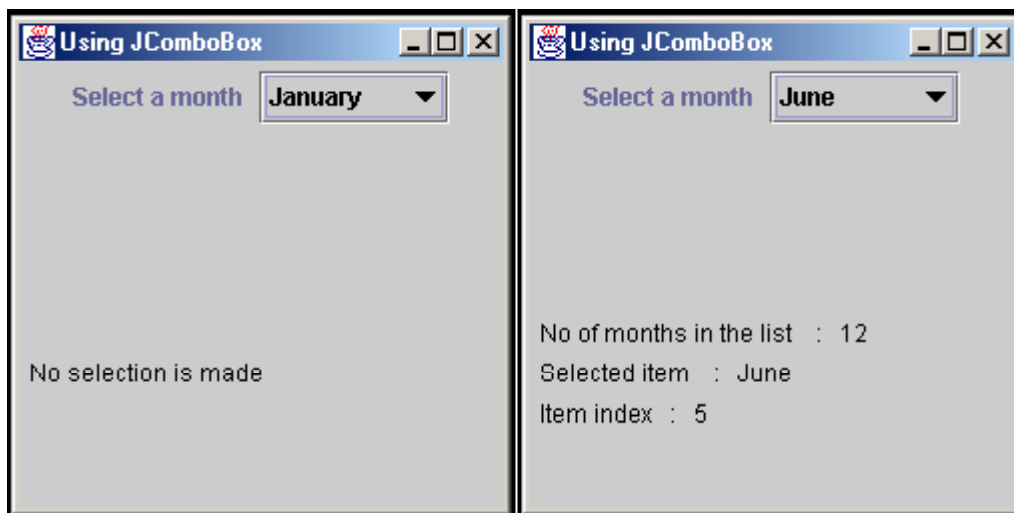


Fig.19.34 Output Screens for Program 19.31. The first window shows the initial display and the second shows the screen after the user's selection.

19.15 JMenuItem, JMenu and JMenuBar

A pull-down menu with several items can be created using **JMenuItem**, **JMenu** and **JMenuBar** objects. **JMenu** is a subclass of **JComponent** and **JMenuItem** is a subclass of **JMenu**. **JMenuBar** is a subclass of **JComponent**. A **JMenu** contains several **JMenuItem**. By clicking menu item, actions can be initiated. A **JMenuItem** is like a button. A **JMenuItem** is to be attached to a **JMenu** object. A **JMenu** is to be attached to a **JMenuBar**. A **JMenuBar** is to be attached to a **JFrame** or **JApplet** window. To create a menu window, the following steps are to be followed:

- Create JMenuItem (many).
- Create JMenu (many).
- Create JMenuBar.
- Create JFrame(or JApplet).
- Add all JMenuItem to JMenu.
- Add all JMenu to JMenuBar.
- Add JMenuBar to JFrame(or JApplet).

19.15.1 Creating JMenuItem

A **JMenuItem** is a part of **JMenu** displayed in a window. When a **JMenu** is clicked, a popup menu appears, displaying all menu items contained in it. Each menu item acts like a button. By clicking a menu item, actions can be initiated. In this section, we will see how to create **JMenuItem**.

The following constructors are used to create **JMenuItem**. A **JMenuItem** can contain a string or an image or both.

`JMenuItem()`

Creates an empty menu item

`JMenuItem(Icon img)`

Creates a menu item with the specified icon

`JMenuItem(String str)`

Creates a menu item with a text as specified in the string

`JMenuItem(String text, Icon img)`

Creates a menu item with the specified string and icon

Methods

The actions created by clicking the menu item are handled by several methods defined in the **JMenuItem** class. Some of them are :

`void addActionListener(ActionListener al)`

Adds action listener to the menu item

`void addMenuDragMouseListener(MenuDragMouseListener mdl)`

Adds menu drag mouse listener to this menu item

`String getActionCommand()`

Returns the action command set for this menu item

`Icon getIcon()`

Returns the icon of this menu item

`String getText()`

Returns text of this menu item

`void setActionCommand(String cmd)`

Sets the specified string as action command

`void setIcon(Icon img)`

Sets the icon for this menu item

`void setRolloverIcon(Icon img)`

Sets the roll over icon for this menu item

19.15.2 Creating JMenu

JMenu is a container for **JMenuItem**. A menu can be attached with several menu items. When a menu is clicked, a popup menu displays all the menu items. Several such menus can be attached to a menu bar. A **JMenu**

can also contain **JSeparator**, which displays a visual line separator between two menu items. In this section, we will see how to create **JMenu** objects.

The constructors used for creating **JMenu** are :

JMenu()

Creates an empty menu

JMenu(String str)

Creates a menu with the specified string

JMenu(String str, boolean tearoff)

Creates a menu with the specified string; the boolean specifies whether the menu is tear-off or not.

Methods

The **JMenu** class has a number of methods to manage **JMenu** objects. Some of them are:

Component add(Component com)

Appends a component to the end of the menu

Component add(Component com, int index)

Inserts the specified component at the specified location in the menu

JMenuItem add(JMenuItem mitem)

Adds the specified menu item at the end of the menu

JMenuItem add(String str)

Creates a new menu item with the specified string and appends it to the end of the menu

void addSeparator()

Appends a separator to the menu

int getItemCount()

Returns the total number of items, including the separator, in the menu

JMenuItem insert(JMenuItem mitem, int index)

Inserts the specified menu item at the specified index

void addActionListener(ActionListener al)

Adds action listener

19.15.3 Creating JMenuBar

JMenuBar is a subclass of **JComponent**. A menu bar holds many menus in its bar. A menu bar has to be attached to a **JFrame** window or

to a **JApplet** window. There is only one constructor to create a **JMenuBar**. It is:

```
JMenuBar()
```

Creates a menu bar

Methods

A menu bar can be managed using the methods defined in **JMenuBar** class. Some of them are:

```
JMenu add(JMenu jm)
```

Adds specified menu to the menu bar

```
JMenu getHelpMenu()
```

Returns the help menu

```
JMenu getMenu(int index)
```

Returns the menu at the specified location

```
int getMenuCount()
```

Returns the number of items in the menu bar

```
void setHelpMenu(JMenu jm)
```

Sets a help menu in the menu bar

19.15.4 Creating JMenu on JFrame

Having seen the functions of the **JMenuItem**, **JMenu** and **JMenuBar**, we will see in this section, how all these three components are integrated together to produce a user window with a menu. We will follow the steps mentioned in section 19.15. Program 19.32 shows how to create a window with two menus, cities and states, each having several items. First a **JMenuBar** is created using the constructor and **setMenuBar()** method. Then two menus, cities and states, are created. Several menu items for each of them are created and added to the respective menu using **add()** method. The menus are then added to the menu bar. Note that one of the menu items for cities is set with an image icon.

Program 19.32

```
/* This program illustrates the following:
   JMenuItem
   JMenu
   JMenuBar
   somasundaramk@yahoo.com
*/
```

```

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

// You need a JFrame window to set up a menu bar.
class Menufrmael
    extends JFrame
    {
        JMenuItem albd, blore, cbe, dhi, chni, dgl, mdu;
        JMenuItem ap, up, ka, kr, or, mh, bn, pb, hr, tn;

        Menufrmael(String title)
        {
            super(title);

            // create a menu bar
            JMenuBar mnubar = new JMenuBar();
            setJMenuBar(mnubar);

            // create menu
            JMenu city = new JMenu("Cities");

            //create menu items
            Icon laughimg = new ImageIcon("laugh.jpg");

            albd = new JMenuItem("Allahabad");
            blore = new JMenuItem("Bangalore");
            cbe = new JMenuItem("Coimbatore");
            dhi = new JMenuItem("Delhi");
            chni = new JMenuItem("Chennai");
            dgl = new JMenuItem("Dindigul");
            mdu = new JMenuItem("Madurai", laughimg);
            // add menu items to menu
            city.add(albd);
            city.add(blore);
            city.add(cbe);
            city.add(chni);
            city.add(dhi);
            city.add(dgl);
            city.add(mdu);
            // attach city menu to menu bar
            mnubar.add(city);

            // create a second menu
            JMenu states = new JMenu("States");

            // Create menu items for the second menu
            ap = new JMenuItem("Andhra Pradesh");
            up = new JMenuItem("Uttar Pradesh");
            ka = new JMenuItem("Karnataka");
            kr = new JMenuItem("Kerala");
            or = new JMenuItem("Orissa");

```

```
mh = new JMenuItem("Maharashtra");
bn = new JMenuItem("West Bengal");
pb = new JMenuItem("Punjab");
hr = new JMenuItem("Haryana");
tn = new JMenuItem("Tamil Nadu");

// add menu items to menu
states.add(ap);
states.add(hr);
states.add(ka);
states.add(kr);
states.add(mh);
states.add(or);
states.add(pb);
states.add(up);
states.add(tn);

//add the menu states to the menubar
mnuBar.add(states);
setSize(250, 250);
setVisible(true);
addWindowListener(new WindowAdapter()
{
    public void windowClosing(WindowEvent we)
    {
        System.exit(0);
    }
});
}
}
class JMenuf1
{
    public static void main(String args [])
    {
        new Menufrmae1("JMenu Example");
    }
}
```

The above program gives the following output screen:

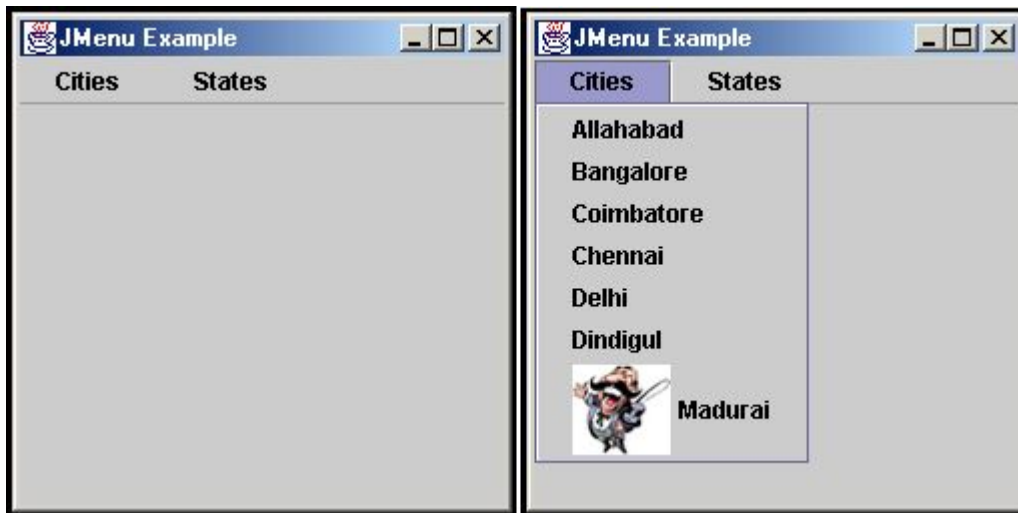


Fig.19.35 Output Screens for Program 19.32 The first window shows the initial display. The second window shows the display after the user clicked cities menu.

19.15.5 Creating JMenu on JApplet

A JApplet program which creates the same user window as that of the above program (19.32) is shown in program 19.33.

Program 19.33

```
/* This program illustrates how to use the following
   JMenuItem
   JMenu
   JMenuBar
   on JApplet window.
   somasundaramk@yahoo.com
   <applet code = JMenual width =250 height =250>
   </applet>
   */

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

public class JMenual
    extends JApplet
    {
        JMenuItem albd, blore, cbe, dhi, chni, dgl, mdu;
        JMenuItem ap, up, ka, kr, or, mh, bn, pb, hr, tn;
        JMenu city, states;

        public void init()
        {
            // create a menu bar
            JMenuBar mnubar = new JMenuBar();
            setJMenuBar(mnubar);
        }
    }

```



```
// create city menu
city = new JMenu("Cities");
//create menu items for city
Icon laughimg = new ImageIcon("laugh.jpg");
albd = new JMenuItem("Allahabad");
blore = new JMenuItem("Bangalore");
cbe = new JMenuItem("Coimbatore");
dhi = new JMenuItem("Delhi");
chni = new JMenuItem("Chennai");
dgl = new JMenuItem("Dindigul");
mdu = new JMenuItem("Madurai", laughimg);
// add menu items to menu
city.add(albd);
city.add(blore);
city.add(cbe);
city.add(chni);
city.add(dhi);
city.add(dgl);
city.add(mdu);
// attach city menu to menu bar
mnubar.add(city);
// create states menu
states = new JMenu("States");
// Create menu items for the states menu
ap = new JMenuItem("Andhra Pradesh");
up = new JMenuItem("Uttar Pradesh");
ka = new JMenuItem("Karnataka");
kr = new JMenuItem("Kerala");
or = new JMenuItem("Orissa");
mh = new JMenuItem("Maharashtra");
bn = new JMenuItem("West Bengal");
pb = new JMenuItem("Punjab");
hr = new JMenuItem("Haryana");
tn = new JMenuItem("Tamil Nadu");

// add menu items to menu
states.add(ap);
states.add(hr);
states.add(ka);
states.add(kr);
states.add(mh);
states.add(or);
states.add(pb);
states.add(up);
states.add(tn);
//add the menu states to the menubar
mnubar.add(states);
}
```

19.16 JDialog

JDialog object is a top-level window with a title and a border. **JDialog** is a subclass of **Dialog**. A **JDialog** is created by another window to enable the user to input data or to display a message. A dialog must have either a **Frame** or **Dialog** as its owner. There are two types of dialog windows, modal and non-modal. A modal dialog prevents input to other top-level windows. By default, all dialogs are non-modal. **JDialog** creates **WindowOpened**, **WindowClosing**, **WindowClosed**, **WindowActivated** and **WindowDeactivated** events.

19.16.1 Creating JDialog

JDialog objects are created as a result of some action performed on the user-interface components. Hence, there essentially is a owner for the **JDialog**. Whenever some process takes place, a **JDialog** is created as a response to the process. A **JDialog** is created using the following constructors:

JDialog(Dialog owner)

Creates a new dialog window with an empty title and the specified owner dialog; the dialog is invisible and non-modal.

JDialog(Dialog owner, String title)

Creates a new dialog window with the specified dialog as owner and the specified string as title

JDialog(Dialog owner, String title, boolean modal)

Creates a new dialog window with the specified dialog as owner and the specified string as title; the boolean value specifies whether the dialog is modal or non-modal.

JDialog(Frame owner)

Creates a new dialog window with the specified frame as owner without a title and is invisible

JDialog(Frame owner, boolean modal)

Creates a new dialog window with the specified frame as owner and without a title; the boolean value specifies whether the dialog is modal or non-modal.

JDialog(Frame owner, String title, boolean modal)

Creates a new dialog with the specified frame as owner and the specified string as title; the boolean value specifies whether the dialog is modal or non-modal.

Methods

JDialog class has a number of methods to handle the dialog windows. Some of them are :

```
void dispose()
    Disposes the dialog

String getTitle()
    Returns the title of the dialog

Container getContentPane()
    Returns the content pane

void hide()
    Hides the dialog

boolean isModal()
    Returns true, if the dialog is modal

void setTitle(String str)
    Sets the title for the dialog

void setJMenuBar(JMenuBar menu)
    Sets the menu bar for the dialog

void show()
    Makes the dialog visible
```

19.16.2 Using JDialog

JDialog windows enable the user to make a dialog-like interaction with the user, allowing them to input a data or get messages to proceed further. In the following program 19.34, we will illustrate how to create a menu and then create a dialog window as a response to the clicking of an item in a menu. The `JDialog` class sets up a dialog. The `JDialogGenerator` frame creates a menu select. The select menu has three menu items, Start, Continue and Exit. Clicking the first two items will not produce any response. When the menu item Exit is clicked, a dialog window is created. When the Close button on the dialog window is selected, it is hidden.

Program 19.34

```
/* This program illustrates the creation
   of JDialog on a JFrame.
   somasundaramk@yahoo.com
*/

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

```

import java.awt.event.*;
class JDial
    extends JDialog
    implements ActionListener
{
    Container conpan;
    JDial(Frame fm, String str)
    {
        super(fm, str);
        conpan = getContentPane();
        JButton b = new JButton(" Close");
        b.addActionListener(this);
        conpan.add(b);
        setSize(150, 100);
    }
    public void actionPerformed(ActionEvent ae)
    {
        dispose();
    }
}
class JDiagenerator
    extends JFrame
    implements ActionListener
{
    String cmd = "";
    JDial dl;
    Container conpan;
    JDiagenerator(String str)
    {
        super(str);
        conpan = getContentPane();
        addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent we)
            {
                System.exit(0);
            }
        });
        conpan.setLayout(new BorderLayout());
        JLabel mesg = new JLabel("Click each choice in the
                                   Select menu");
        conpan.add(mesg, BorderLayout.SOUTH);
        JMenuBar mbar = new JMenuBar();
        setJMenuBar(mbar);
        JMenu select = new JMenu("Select");
        JMenuItem start = new JMenuItem("Start");

```

```

JMenuItem cont = new JMenuItem("Continue");
JMenuItem stop = new JMenuItem("Exit");
select.add(start);
select.add(cont);
select.add(stop);
mbar.add(select);
conpan.add(mbar, BorderLayout.NORTH);
setSize(300, 200);
setVisible(true);
start.addActionListener(this);
cont.addActionListener(this);
stop.addActionListener(this);
}
public void actionPerformed(ActionEvent ae)
{
    cmd = ae.getActionCommand();
    if (cmd == "Exit")
    {
        d1 = new JDial(this, "JDialog Window");
        d1.setVisible(true);
        d1.setLocation(100, 50);
    }
    else
    {
        d1 = new JDial(this, " ");
        d1.setLocation(70, 50);
        d1.setVisible(false);
    }
}
}
class JDIALOGf
{
    public static void main(String args [])
    {
        new JDiagenerator("Main Window");
    }
}

```

The above program gives the following output screen:

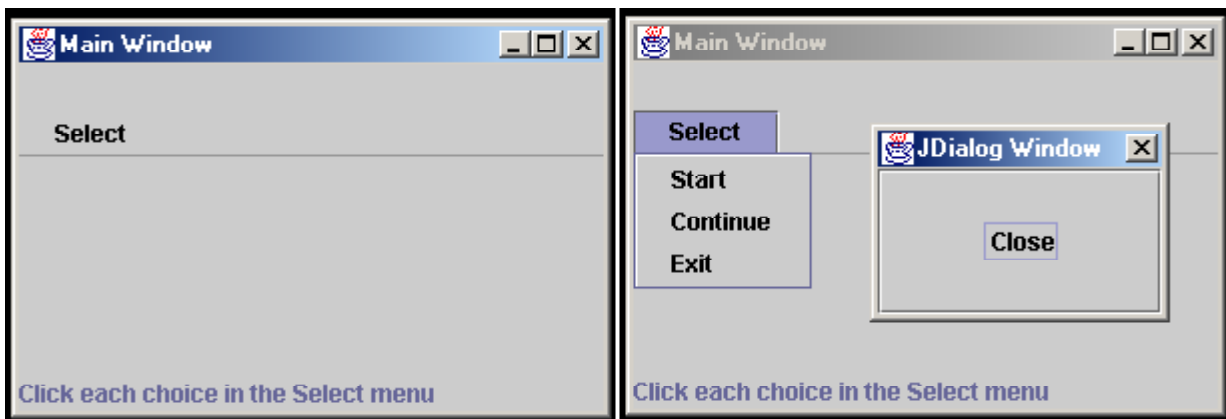


Fig.19.36 Output Screens for Program 19.34. The first window shows the initial display. The second window shows the screen after the user has clicked Exit item.



JDialog creates a top-level window. However, there should be a parent Component to which the dialog frame has to be attached. A JDialog can be owned by a JFrame or another JDialog.

19.17 JOptionPane

JOptionPane is used for creating dialogs in a user-interface. **JOptionPane** can display text messages and icons. **JOptionPane** class provides a large number of methods that can be used to create instant dialogs. **JOptionPane** has methods to display dialogs with different types of messages. Simple dialogs can be created easily than using **JDialog** objects. Dialog created by **JOptionPane** contains predefined buttons, which can be used for developing interactive applications. **JOptionPane** has a number of static methods that can be called directly to create dialogs. **JOptionPane** has static methods to create dialogs of the following types:

- | | | |
|----------------------|---|---|
| Message Dialogs | - | Gives some information |
| Confirmation Dialogs | - | Asks the user to confirm something that is going to take place like Yes/No/Cancel |
| Input Dialogs | - | Asks the user to give input |
| Option Dialogs | - | It is a dialog that combines all the above three. |

JOptionPane defines a number of constants that are used to create standard dialogs. The static methods and constructors have one or more of the following parameters as arguments:

ParentComponent	This is the component which has generated the dialog.
message	This is the message meant for the user to read. It can be an array of object or an Icon or a Component that is to be displayed.
messageType	<p>This defines the style of the displayed message. It always has a default icon that is displayed along with the message. This value is defined by the following int type constants:</p> <p>JOptionPane.ERROR_MESSAGE JOptionPane.INFORMATION_MESSAGE JOptionPane.WARNING_MESSAGE JOptionPane.QUESTION_MESSAGE JOptionPane.PLAIN_MESSAGE</p>
optionType	<p>This defines the buttons that appear on the dialog box. This value is defined by the following int type constants:</p> <p>JOptionPane.DEFAULT_OPTION JOptionPane.YES_NO_OPTION JOptionPane.YES_NO_CANCEL_OPTION JOptionPane.OK_CANCEL_OPTION JOptionPane.YES_OPTION</p>
icon	This is a decorative icon that is displayed in the dialog box. The default icon is defined by the messageType.
title	It is a string that is displayed as title for the dialog box.

19.17.1 Creating JOptionPane

JOptionPane has number of constructors to create dialogs. Some of them are:

`JOptionPane(Object message)`

Creates an option pane to display the specified message

`JOptionPane(Object message, int messageType)`

Creates an option pane to display the specified message with a style specified by messageType

`JOptionPane(Object message, int messageType, int optionType)`

Creates an option pane to display the specified message with a style specified by messageType and with buttons specified by optionType

`JOptionPane(Object message, int messageType, int optionType, Icon icon)`

Creates an option pane to display the specified message with a style

specified by `messageType` and with buttons specified by `optionType` with the specified icon instead of default icon defined by `messageType`

Methods

JOptionPane has a large number of methods to create dialog boxes and manage **JOptionPane** objects. Static methods defined in **JOptionPane** class can be used to create dialog boxes. Some of the methods are:

`JDialog createDialog(Component parentComponent, String title)`

Creates and returns a `JDialog` with the specified string as title for the dialog box

`Icon getIcon()`

Returns icon of the pane

`Object getMessage()`

Returns the message displayed in the pane.

`Object getValue()`

Returns the value the user had selected

`void setMessage(Object message)`

Sets the message for the pane

`void setOptionType(int optionType)`

Sets the `optionType` for the pane

`static int showConfirmDialog(Component parentComponent, Object message)`

Brings up a dialog box with the specified message with buttons Yes, No and Cancel

`static int showConfirmDialog(Component parentComponent, Object message, String title, int optionType)`

Brings up a dialog box with the specified message with the specified string as title for the dialog box and with buttons specified by `optionType`

`static int showConfirmDialog(Component parentComponent, Object message, String title, int optionType, int messageType)`

Brings up a dialog box with the specified message with the specified string as title for the dialog box with buttons specified by `optionType` and display style specified by `messageType`

`Static int showConfirmDialog(Component parentComponent, Object message, String title, int optionType, int messageType, Icon icon)`

Brings up a dialog box with the specified message with the specified string as title with buttons specified by `optionType` with display style specified by `messageType` and with the specified icon

`static int showMessageDialog(Component parentComponent, Object message)`
 Brings up a dialog box to display the specified message and with a default OK button

`static int showMessageDialog(Component parentComponent, Object message, String title, int messageType)`
 Brings up a dialog box to display the specified message with the specified title and style specified by `messageType`

`static int showMessageDialog(Component parentComponent, Object message, String title, int messageType, Icon icon)`
 Brings up a dialog box to display the specified message with the specified title with style specified by `messageType` and with specified icon

To create a dialog box using **JOptionPane**, any of the following two techniques may be followed:

1. Construct a `JOptionPane`
 ex : `JOptionPane job = new JOptionPane(.....);`
 Call the `createDialog` method
 ex : `JDialog jd = job.createDialog(.....);`
 Call the `show()` method
 ex : `jd.show()`
2. Call the static method directly
 ex : `JOptionPane.showMessageDialog(.....);`

19.17.2 Using JOptionPane

Dialog boxes can be created using **JOptionPane**'s constructors or by calling its static methods. In the following program 19.35, we show how the raw **JOptionPane** will look like. Generally, **JOptionPane** objects are created as a result of a user clicking a component in a user-interface screen. In this example, the **JOptionPane** objects are being displayed without any parent component for option pane. The program 19.35 generates output for four message types.

Program 19.35

```
/* This program illustrates the creation of JOptionPane.
   somasundaramk@yahoo.com
*/
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class Joption
    extends JFrame
    {
```

```

Container conpan;
Joption(String str)
{
    super(str);
    conpan = getContentPane();
    conpan.setLayout(new FlowLayout(FlowLayout.LEFT));
    JOptionPane jop1 = new JOptionPane("Information
        message ", JOptionPane.INFORMATION_MESSAGE);
    JOptionPane jop2 = new JOptionPane("Warning
        message", JOptionPane.WARNING_MESSAGE);
    JOptionPane jop3 = new JOptionPane("Question
        message", JOptionPane.QUESTION_MESSAGE);
    JOptionPane jop4 = new JOptionPane("Error
        message", JOptionPane.ERROR_MESSAGE);
    JLabel info = new JLabel("Default Icons for
        message types");
    addWindowListener(new WindowAdapter()
    {
        public void windowClosing(WindowEvent we)
        {
            System.exit(0);
        }
    });
    conpan.add(jop1);
    conpan.add(jop2);
    conpan.add(jop3);
    conpan.add(jop4);
    conpan.add(info);
    setSize(550, 250);
    setVisible(true);
}
}
class JOpan
{
    public static void main(String args [])
    {
        new Joption("JOptionPane ");
    }
}

```

The above program gives the following output:



Fig.19.37 Output Screen for Program 19.35. Note the default icons in the message.

In the following program 19.36, we show how **JOptionPane** can be used to create dialog in applications. First, we create a **JFrame** window. We then add three **JButtons** and a **JLabel**. The action listeners are registered with the buttons. Whenever the respective buttons are clicked by the user, action events are generated and the **actionPerformed()** method is called. To illustrate various message types, we create different option panes. The **oplangb** button, when clicked, generates a dialog with information message. The **corgb** button, when clicked, generates a dialog with warning message with Yes, No, Cancel options. The **mmb** button, when clicked, generates a dialog box with a warning message with an icon. The dialog box for **oplangb** button is created using the **JOptionPane** constructor, **createDialog()** and **show()** method. The dialog box for **corgb** button is created using the **JOptionPane**'s static method **showConfirmDialog()** method and for the **mmb** button, **showMessageDialog()** is used.

Program 19.36

```
/* This program illustrates the creation and use of
   JOptionPane.

   somasundaramk@yahoo.com
*/
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class Joption1
    extends JFrame
    {
        Container conpan;
```

```

JButton oplangb, corgb, mmb;
JLabel inform;

Joptionl(String str)
{
    super(str);
    conpan = getContentPane();
    conpan.setLayout(new FlowLayout(FlowLayout.LEFT));
    inform = new JLabel("Click the following to know
                        about them");
    oplangb = new JButton("OOP Language");
    corgb = new JButton("Computer Organization");
    mmb = new JButton("Multimedia");

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

    oplangb.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent ae)
        {
            String langs = "OOP Language deals with\n
                            Object-Oriented Languages like Java/
                            C++";
            JOptionPane jop = new JOptionPane(langs,
                                                JOptionPane.INFORMATION_MESSAGE);
            JDialog jdia = jop.createDialog(oplangb,
                                            "Infromation");
            Icon laugh = new ImageIcon("laugh.jpg");
            jdia.show();
        }
    });

    corgb.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent ae)
        {
            String corgs = "Computer Organization
                            deals with\n functioning of a computer";
            JOptionPane.showConfirmDialog(corgb,
                                           corgs, "Confirm Dialog",
                                           JOptionPane.YES_NO_CANCEL_OPTION,
                                           JOptionPane.WARNING_MESSAGE);
        }
    });
}

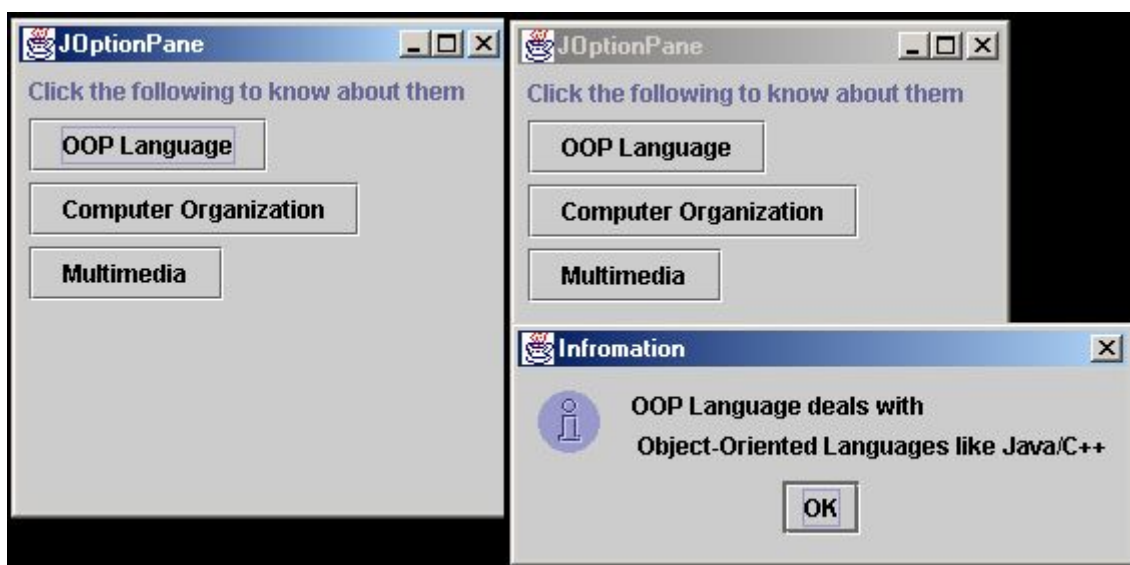
```

```

mmb.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent ae)
    {
        Icon laugh = new ImageIcon("laugh.jpg");
        String mms = "Multimedia deals with \n
            the combination of text,audio and video";
        JOptionPane.showMessageDialog(mmb, mms,
            "Warning", JOptionPane.WARNING_MESSAGE,
            laugh);
    }
});
conpan.add(inform);
conpan.add(oplangb);
conpan.add(corgb);
conpan.add(mmb);
setSize(250, 250);
setVisible(true);
}
}
class JOpan1
{
    public static void main(String args [])
    {
        new Joption1("JOptionPane ");
    }
}

```

The above program gives the following output screens:



(a)



(b)

Fig.19.38 Output Screens for Program 19.36

- a) The first one is the initial screen and the second one shows dialog box created after clicking the **oplangb** button.
- b) The first window shows dialog window created after clicking the **corgb** button and the second after clicking the **mmb** button. All the dialog windows are manually shifted to show both the main and dialog windows.

The dialogs are the after-effect of clicking a button or any component that can generate event. Event listeners are registered with the event source and as and when events occur, the event listeners carry out some process required for that application. For **JButton** objects, action listeners can be used to capture the event and generate dialog messages. In the previous program 19.36, action listeners were registered with the buttons and the **actionPerformed()** method was called through the anonymous inner class. This way of processing is straightforward and it is explicitly shown what action will be carried out for each button click. However, if there are more number of components in an application and for each of them only the dialog messages differ, then implementing the **actionPerformed()** method for each button appears to be repeating the same statements. An alternative way is to implement one common **actionPerformed()** method for action events received from different event sources. When one **actionPerformed()** method is used for several event sources (several buttons), it is necessary to identify from which source an event has originated. For that, **getSource()** and **getActionCommand()** methods can be used. We will show this technique in program 19.37.

Program 19.37

```
/* This program illustrates the creation and use of
   JOptionPane.

   somasundaramk@yahoo.com
*/

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class Joption2
    extends JFrame
    implements ActionListener
    {
        Container conpan;
        JOptionPane jop;
        JButton oplangb, corgb, mmb;

        Joption2(String str)
        {
            super(str);
            conpan = getContentPane();
            conpan.setLayout(new FlowLayout(FlowLayout.LEFT));
            JLabel inform = new JLabel("Click the following to
                                         know about them");

            oplangb = new JButton("OOP Language");
            corgb = new JButton("Computer Organization");
            mmb = new JButton("Multimedia");
            addWindowListener(new WindowAdapter()
            {
                public void windowClosing(WindowEvent we)
                {
                    System.exit(0);
                }
            });

            oplangb.addActionListener(this);
            corgb.addActionListener(this);
            mmb.addActionListener(this);
            oplangb.setActionCommand("oplang");
            corgb.setActionCommand("corg");
            mmb.setActionCommand("mm");
            conpan.add(inform);
            conpan.add(oplangb);
            conpan.add(corgb);
            conpan.add(mmb);
            setSize(250, 250);
            setVisible(true);
        }
    }
}
```

```

public void actionPerformed(ActionEvent ae)
{
    JButton buttonsource = (JButton)ae.getSource();
    String cmd = buttonsource.getActionCommand();
    String message = null;
    JDialog jdia;
    JOptionPane jop;

    if (cmd == "oplang")
    {
        message = "OOP Language deals with\n Object-
                    Oriented Languages like Java/C++";
        JOptionPane.showMessageDialog(buttonsource,
                                    message, "Information",
                                    JOptionPane.INFORMATION_MESSAGE);
    }

    if (cmd == "corg")
    {
        message = "Computer Organization deals with\n
                    functioning of a computer";
        JOptionPane.showMessageDialog(buttonsource,
                                    message, "Warning",
                                    JOptionPane.WARNING_MESSAGE);
    }

    if (cmd == "mm")
    {
        message = "Multimedia deals with \n the
                    combination of text,audio and video";
        Icon magesh = new ImageIcon("mag65.jpg");
        JOptionPane.showMessageDialog(buttonsource,
                                    message, "Plain", JOptionPane.PLAIN_MESSAGE,
                                    magesh);
    }
}

class JOpan2
{
    public static void main(String args [])
    {
        new Joption2("JOptionPane ");
    }
}

```

The above program gives the following output:



Fig.19.39 Output Screens for Program 19.37

The purpose of generating a dialog is to display a message and get the response from the user. In the case of displaying information message and warning message, press OK button in the dialog box. However, in the case of displaying confirmation message, the user has the option of selecting Yes, No or Cancel. In those occasions, it is necessary to know which button the user has selected in the dialog box. The **showConfirmDialog()** method returns an int value. This int value represents any of the following values, which can be used for the application:

JOptionPane.YES_OPTION
JOptionPane.NO_OPTION
JOptionPane.CANCEL_OPTION
JOptionPane.OK_OPTION
JOptionPane.CLOSED_OPTION

The last value **CLOSED_OPTION** represents that the user has closed the dialog window without selecting any option.

In the following program 19.38, we will show how to detect the selection made in the dialog window of a confirmation message. We create four **JButtons** and register action listener with each one of them. When a button is clicked, a dialog with a confirmation message with Yes and No button is displayed using **showConfirmDialog()** method. When the user clicks either Yes or No button, according to the message displayed, the user's selection is identified by taking the int value assigned to the variable result1. Accordingly, another dialog message appears, indicating whether the user has selected Yes or No button.

Program 19.38

```

/* This program illustrates how to create
   JOptionPane dialog and detect the
   option button selected by the user.
   somasundaramk@yahoo.com
*/
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class Joptionuse
    extends JFrame
    {
    Container conpan;
    JButton teleb, compb, zerob, sunb;
    JLabel inform1, inform2;
    int result1;
    Icon laugh = new ImageIcon("laugh.jpg");
    Joptionuse(String str)
        {
        super(str);
        conpan = getContentPane();
        conpan.setLayout(new FlowLayout(FlowLayout.LEFT));
        inform1 = new JLabel("Click the following and ");
        inform2 = new JLabel(" test the statements");
        teleb = new JButton("Telescope");
        compb = new JButton("Computer ");
        zerob = new JButton("Zero          ");
        sunb = new JButton("Sun           ");
        addWindowListener(new WindowAdapter()
            {
            public void windowClosing(WindowEvent we)
                {
                System.exit(0);
                }
            });
        teleb.addActionListener(new ActionListener()
            {
            public void actionPerformed(ActionEvent ae)
                {
                String teles = "Aristotle invented
                                Telescope.";
                result1 =
                    JOptionPane.showConfirmDialog(teleb,
                                                    teles, "Confirm Dialog",
                                                    JOptionPane.YES_NO_OPTION,
                                                    JOptionPane.WARNING_MESSAGE);
                String selection;
                switch (result1)

```

```

        {
        case JOptionPane.YES_OPTION:
            selection = "Sorry.\n Only Galileo
                        invented Telescope";
            break;
        case JOptionPane.NO_OPTION:
            selection = "You are RIGHT.\n
                        Only Galileo invented
                        Telescope";
            break;
        case JOptionPane.CLOSED_OPTION:
        default:
            selection = "Window closed without
                        answering";
        }
        JOptionPane.showMessageDialog(teleb, " "
            + selection, "Selection",
            JOptionPane.INFORMATION_MESSAGE,
            laugh);
    }
    });
    compb.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent ae)
        {
            String comps = "The concept of present-day
                            computer\n was given by Charless
                            Babbage ";

result1 =
            JOptionPane.showConfirmDialog(compb,
                comps, "Confirm Dialog",
                JOptionPane.YES_NO_OPTION,
                JOptionPane.WARNING_MESSAGE);
            String selection;
            switch (result1)
            {
                case JOptionPane.YES_OPTION:
                    selection = "Yes, You are RIGHT.";
                    break;
                case JOptionPane.NO_OPTION:
                    selection = "Sorry, Charless
                                Babbage gave the concept";
                    break;
                default:
                    selection = "Window closed without
                                answering";
            }
            JOptionPane.showMessageDialog(compb,
                selection, "Selection",

```

```

                                JOptionPane.INFORMATION_MESSAGE,
                                laugh);
                            }
                        });
zerob.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent ae)
    {
        String zeros = "The ancient Indian
            Mathematician \n gave the zero to
            the world";
        result1 =
            JOptionPane.showConfirmDialog(zerob,
                zeros, "Confirm Dialog",
                JOptionPane.YES_NO_OPTION,
                JOptionPane.WARNING_MESSAGE);
        String selection;
        switch (result1)
        {
            case JOptionPane.YES_OPTION:
                selection = "Yes, You are RIGHT.";
                break;
            case JOptionPane.NO_OPTION:
                selection = "Sorry, Take pride in
                    telling that\n ancient Indians
                    invented Zero";
                break;
            default:
                selection = "Window closed without
                    answering";
        }
        JOptionPane.showMessageDialog(zerob,
            selection, "Selection",
            JOptionPane.INFORMATION_MESSAGE,
            laugh);
    }
});
sunb.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent ae)
    {
        String suns = "The sun has a solid body";
        result1 =
            JOptionPane.showConfirmDialog(sunb,
                suns, "Confirm Dialog",
                JOptionPane.YES_NO_OPTION,
                JOptionPane.WARNING_MESSAGE);
        String selection;
        switch (result1)
        {

```

```

        case JOptionPane.YES_OPTION:
            selection = "Sorry.\n Sun has a
                        gaseous body";

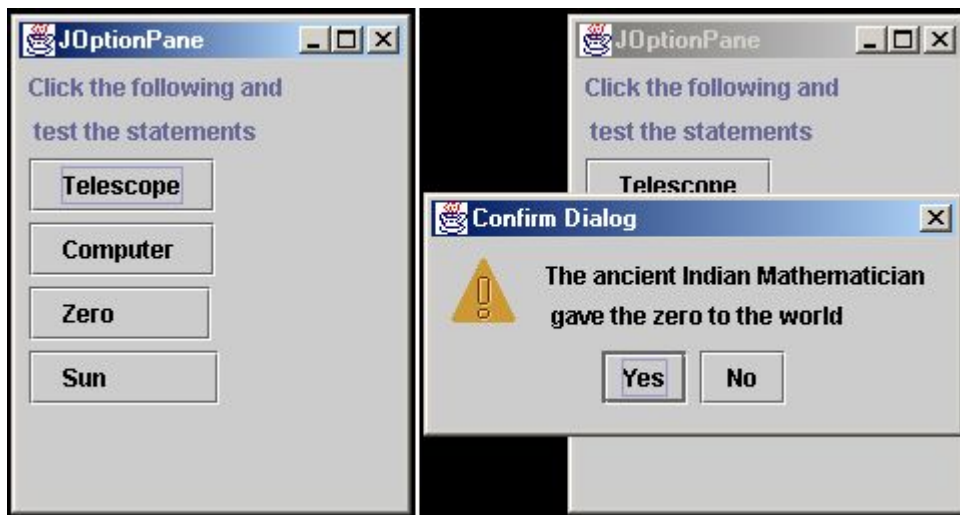
            break;
        case JOptionPane.NO_OPTION:
            selection = "Yes. Sun has only
                        gaseous body";

            break;
        default:
            selection = "Window closed without
                        answering";
    }
    JOptionPane.showMessageDialog(sunb,
        selection, "Selection",
        JOptionPane.INFORMATION_MESSAGE,
        laugh);
    });

    conpan.add(inform1);
    conpan.add(inform2);
    conpan.add(teleb);
    conpan.add(compb);
    conpan.add(zerob);
    conpan.add(sunb);
    setSize(200, 250);
    setVisible(true);
}
}
class JOpuse
{
    public static void main(String args [])
    {
        new Joptionuse("JOptionPane");
    }
}

```

The above program gives the following output screens:



(a)



(b)

Fig.19.40 Output Screens for Program 19.38

- (a) **First window shows the initial display. The second shows the dialog box after clicking the Zero button.**
- (b) **The window shows the response dialog when the No button is selected.**



For creating simple dialogs, use `JOptionPane`.
To detect which option is selected in a dialog, make use of the `int` value returned by the static methods.

19.18 JFileChooser

The **JFileChooser** class helps to open a window and display files and directories in a specified path. **JFileChooser** class is a subclass of **JComponent**. Using **JFileChooser** object, one can create user-interface

windows for opening or saving files. The look and feel of **JFileChooser** windows are different from the conventional file Open/Save Window of windows-based computers. But, the display content and tools are almost the same. **JFileChooser** operates in three different modes, files only, directories and files and directories.

JFileChooser class provides three types of file choosers, which are associated with a dialog. The dialog types are open, save and custom. These dialogs provide options either to proceed with by clicking a button indicating the process or cancel the dialog by clicking cancel option. The methods that create the dialog return integer constants indicating what has been selected by the user. The int type constants are :

JFileChooser.APPROVE_OPTION

JFileChooser.CANCEL_OPTION

19.18.1 Creating JFileChooser

To create file chooser windows, one has to create an object of the type **JFileChooser** and call methods on the object. This class has several constructors. Some of the constructors are:

JFileChooser()

Creates a new file chooser using the user's default directory.

JFileChooser(File path)

Creates a file chooser using the specified file path

JFileChooser(String path)

Creates a file chooser using the file specified in the string

Methods

The **JFileChooser** has a large number of methods. There are methods to create user windows to open and save files. File filtering methods allow only specified file extensions to be displayed. Some of the methods defined in **JFileChooser** class are:

boolean accept(File f)

Returns true if the file is to be displayed

void addActionListener(ActionListener al)

Adds an action listener to the file chooser

JDialog createDialog(Component parent)

Creates a dialog with the specified component as its parent

String getApproveButtonText()

Returns the text contained in the ApproveButton in the dialog display

File getCurrentDirectory()

Returns the current directory

FileFilter getFileFilter()

Returns the currently selected file filter

String getName(File f)

Returns the file name

File[] getSelectedFiles()

Returns a list of selected files

void setCurrentDirectory(File dir)

Sets the specified directory as the current directory

void setFileFilter(FileFilter filter)

Sets the file filter

int showDialog(Component parent, String approveButton)

Pops up a custom file chooser dialog with the specified string as approveButton

int showOpenDialog(Component parent)

Pops up a “Open File” file chooser dialog

int showSaveDialog(Component parent)

Pops up a “Save File” file chooser dialog

19.18.2 Creating JFileChooser Dialog

Using constructors and methods discussed in the previous section, we now show how to create file chooser dialogs. Usually, a file chooser dialog is created when a user clicks a button to open a “Open File” dialog or “Save File” dialog. But, in the following example program 19.39, we create the file chooser dialogs without clicking any button, just to make the concept of opening a file chooser dialog simple.

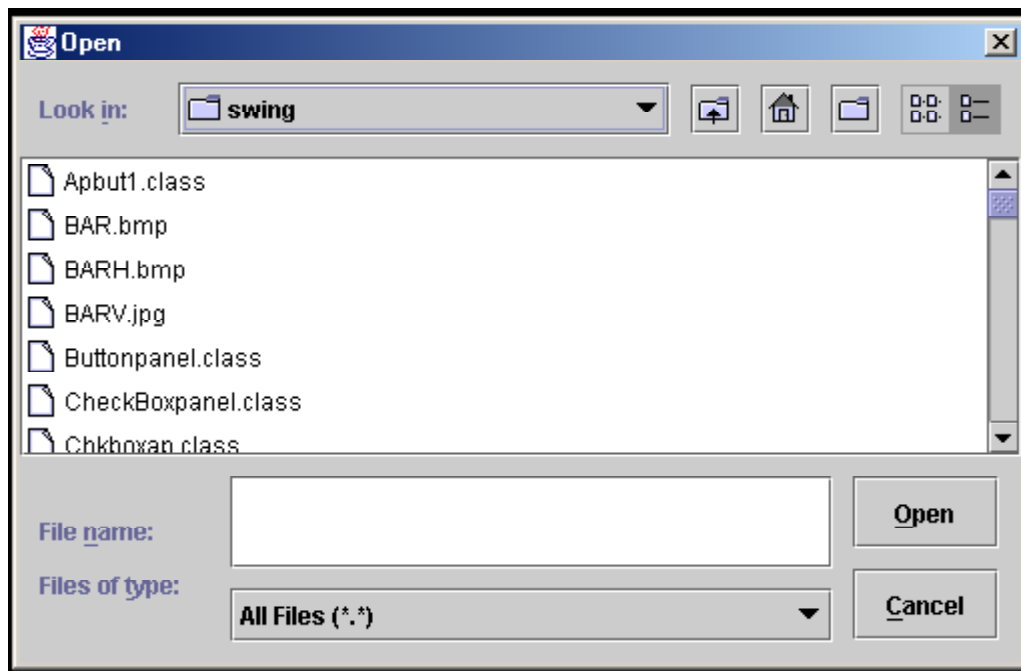
Program 19.39

```
/* This program illustrates the creation of JFileChooser.
   somasundaramk@yahoo.com
*/
import javax.swing.*;
import java.awt.*;
import java.io.*;
```



```
import java.awt.event.*;
class Filechl
    extends JFrame
    {
        Container conpan;
        JFileChooser fchoose;
        JLabel info;
        Filechl(String str)
        {
            super(str);
            conpan = getContentPane();
            conpan.setLayout(new FlowLayout(FlowLayout.LEFT));
            setSize(250, 250);
            info = new JLabel("Dialog created with
                               showOpenDialog");
            conpan.add(info);
            addWindowListener(new WindowAdapter()
            {
                public void windowClosing(WindowEvent we)
                {
                    System.exit(0);
                }
            });
            File myfile = new File("c:/jdk1.2.1/bin/Swing");
            fchoose = new JFileChooser(myfile);
            fchoose.showOpenDialog(this);
            setVisible(true);
        }
        public static void main(String args [])
        {
            new Filechl("File Chooser");
        }
    }
```

The above program gives the following output screens:



(a)



(b)

Fig.19.41 Output Screens for Program 19.39

- (a) The “Open File” file chooser
- (b) Screen display after cancel option is clicked by the user.

19.18.3 Using JFileChooser

In the previous section, we have shown how to create **JFileChooser**. In this section, we will show how to use the **JFileChooser** in an application. The example program 19.40 shows how to create a **JMenu** with File menu and use it like the familiar file handling menu with options, New, Open, Save, Save as and Exit. Menu items, new, open, save, save as and exit, are used to create a **JMenu**. Each of the menu item generates an **ActionEvent** and the

actionPerformed() method initiates dialogs generated by methods of **JFileChooser**. Click of the menu item “open” invokes **showOpenDialog()** method. From the **JFileChooser** object fch, the file selected by the user is obtained by calling the **getSelectedFile()** method, which gives a **File** type object. When the user clicks “open” in the open dialog, the **showOpenDialog()** returns an integer indicating that **JFileChooser.APPROVE_OPTION** is selected. From this, the file selected is displayed using **showConfirmDialog()** method. The user can choose either Yes or No option in the dialog, which ends the open menu item session. Similar process has been added to Save option. The class Respond is developed to create a dialog for the Exit menu item. A manual coding is done using **JDialog** to create the confirmation dialog Yes or No. (It could have been done more easily through **showConfirmDialog()** method). We did this only to show how to code **JDialog** to create interactive dialogs. NO option will return the control to the main window and YES option response to Exit menu item will close the main window.

Program 19.40

```

/* This program illustrates the creation and use
   of JFileChooser in a JMenu on a JFrame window.
   somasundaramk@yahoo.com
*/
import java.awt.*;
import java.io.*;
import javax.swing.*;
import java.awt.event.*;
class Fileframe
    extends JFrame
    implements ActionListener
    {
    Fileframe(String str)
        {
        super(str);
        JMenuBar mbar = new JMenuBar();
        setJMenuBar(mbar);
        JMenu file = new JMenu("File");
        JMenuItem neu = new JMenuItem("New");
        JMenuItem open = new JMenuItem("Open");
        JMenuItem save = new JMenuItem("Save");
        JMenuItem saveas = new JMenuItem("Save As");
        JMenuItem exit = new JMenuItem("Exit");
        neu.addActionListener(this);
        open.addActionListener(this);
        save.addActionListener(this);
        saveas.addActionListener(this);
        exit.addActionListener(this);
        file.add(neu);
        file.add(open);

```

```

        file.add(save);
        file.add(saveas);
        file.add(exit);
        mbar.add(file);
        setSize(250, 200);
        setVisible(true);
        addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent we)
            {
                System.exit(0);
            }
        });
    }
    public void actionPerformed(ActionEvent ae)
    {
        JFileChooser fch = new JFileChooser("c:/jdk1.2.1/
                                           bin/Swing");

        String cmd = " ";
        cmd = ae.getActionCommand();

        if (cmd == "New")
            fch.showDialog(this, "New");
        else if (cmd == "Open")
        {
            int odint = fch.showOpenDialog(this);
            File myfile = fch.getSelectedFile();
            if ((odint == JFileChooser.APPROVE_OPTION) &&
                (myfile != null))
                JOptionPane.showConfirmDialog(this, "File
                    to open : " + myfile.getPath(),
                    "Confirm the file Name",
                    JOptionPane.YES_NO_OPTION);
            else
                JOptionPane.showMessageDialog(this,
                    "Cancelled");
        }
        else if (cmd == "Save")
        {
            int sdint = fch.showSaveDialog(this);
            File myfile = fch.getSelectedFile();
            if ((sdint == JFileChooser.APPROVE_OPTION) &&
                (myfile != null))
                JOptionPane.showConfirmDialog(this, "Save
                    File : " + myfile.getPath(),
                    "Confirm the file Name",
                    JOptionPane.YES_NO_OPTION);
            else
                JOptionPane.showMessageDialog(this,

```

```

        "Cancelled");
    }
    else if (cmd == "Save As")
        fch.showDialog(this, "Save As");
    else if (cmd == "Exit")
        new Respond(this, "Confirm Exit");
    }
}

class Respond
implements ActionListener
{
    Container dconpan;
    JDialog dl;

    Respond(JFrame fm, String title)
    {
        dl = new JDialog(fm, title, true);
        JLabel mesg = new JLabel("Are you sure you want to
                                   Exit?");

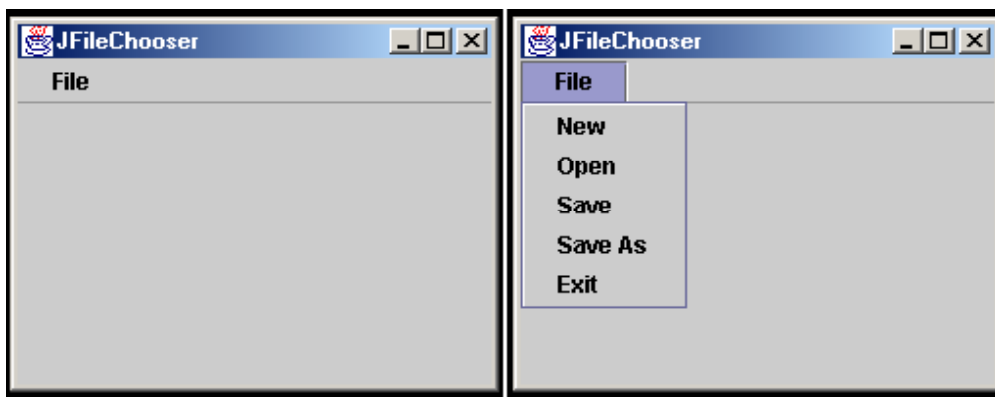
        JButton yes = new JButton("Yes");
        JButton no = new JButton("No");
        yes.addActionListener(this);
        no.addActionListener(this);
        dconpan = dl.getContentPane();
        dconpan.setLayout(new FlowLayout(FlowLayout.LEFT));
        dconpan.add(mesg);
        dconpan.add(yes);
        dconpan.add(no);
        dl.setSize(200, 100);
        dl.setVisible(true);
    }

    public void actionPerformed(ActionEvent ae)
    {
        String res = " ";
        res = ae.getActionCommand();
        if (res == "Yes")
            System.exit(0);
        else
            dl.dispose();
    }
}

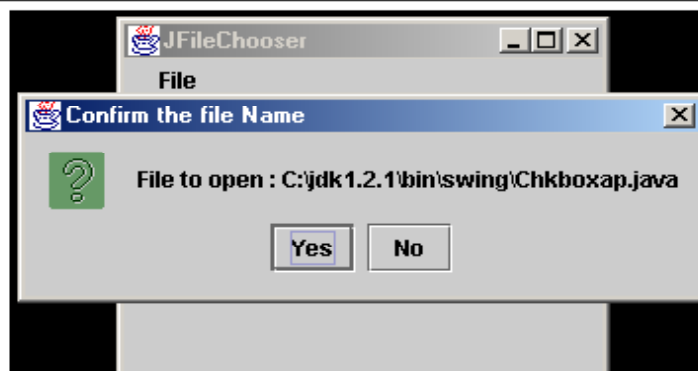
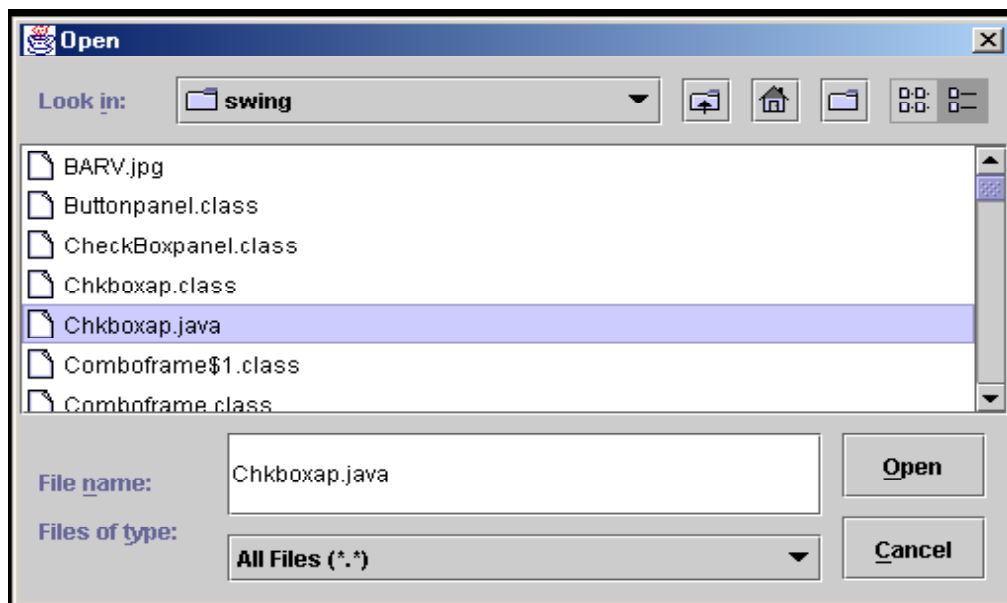
class JFilech2
{
    public static void main(String args [])
    {
        new Fileframe("JFileChooser");
    }
}

```

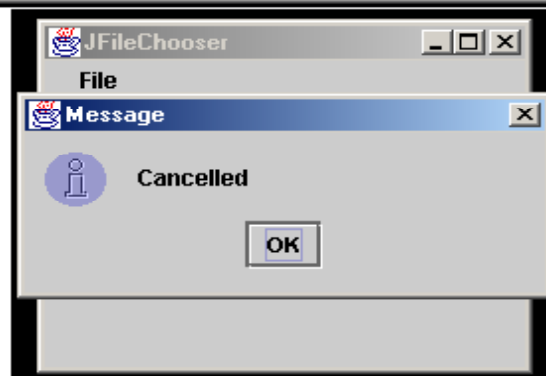
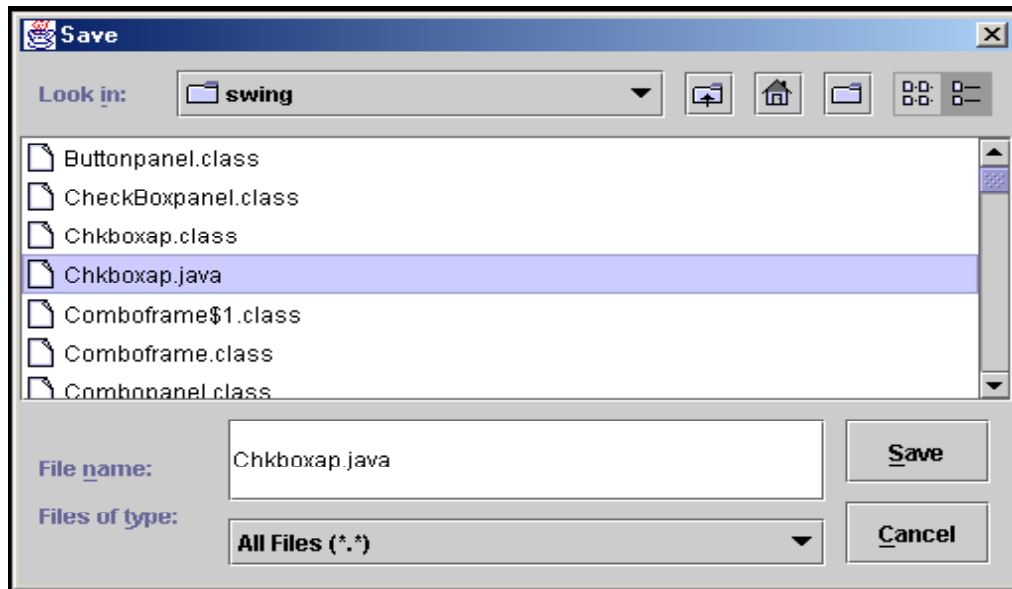
The above program gives the following output screens:



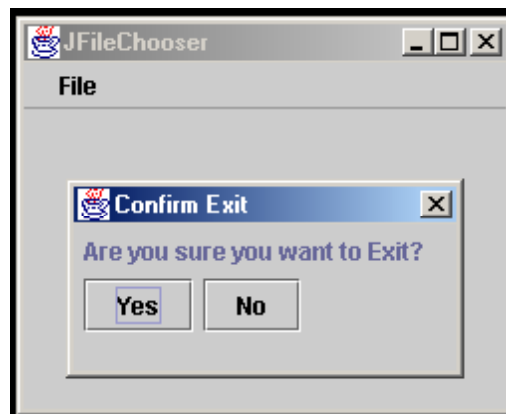
(a)



(b)



(c)



(d)

Fig.19.42 Output Screens for Program 19.40

- (a) The first window shows the initial display. The second shows the popup menu after clicking the File menu.
- (b) The top window shows the open dialog of file chooser. The second shows the message dialog after the Open option is selected.
- (c) The top window shows the save dialog of the file chooser. The second shows the message dialog after the Cancel option is selected.
- (d) The window shows the confirmation dialog. No option will take the control back to the menu and Yes option will close the menu window.



To capture the options selected by the user in **showOpenDialog()**, **showOptionDialog()** and **showSaveDialog()** methods, use the integers returned by these methods. Test the integers with **JFileChooser.APPROVE_OPTION** or **JFileChooser.CANCEL_OPTION**.

19.19 JProgressBar

JProgressBar object gives the visual component that shows the amount of progress of a process that is taking place. It gives a continuously progressing shade, which is proportional to the percentage of work that has been carried out. **JProgressBar** is a subclass of **JComponent**. A **JProgressBar** can be used wherever a process that takes a long time to complete is to be informed to the user. The lowest point of the progress bar is called minimum value and the highest point is called the maximum value. Any point in between the two is called as value.

19.19.1 Creating JProgressBar

A **JProgressBar** can be created in two different orientations. The two orientations are specified by two integer constants:

JProgressBar.VERTICAL

JProgressBar.HORIZONTAL

which define the orientation of the progress bar in vertical or horizontal direction. Some of the constructors used to create a **JProgressBar** are:

JProgressBar()

Creates a horizontal progress bar that displays a border; the initial minimum value is 0 and the maximum value is 100.

JProgressBar(int orientation)

Creates a new progress bar in the specified orientation; the initial minimum value is 0 and the maximum is 100.

JProgressBar(int min, int max)

Creates a new horizontal progress bar with the specified minimum and maximum value

JProgressBar(int orientation, int min, int max)

Creates a progress bar with the specified orientation with the specified minimum and maximum value

Methods

The **JProgressBar** class has a number of methods to update and manage a progress bar. Some of the methods defined in **JProgressBar** class are :

`void addChangeListener(ChangeListener cl)`

Adds the specified change listener to the progress bar

`int getMaximum()`

Returns the maximum value of the progress bar

`int getMinimum()`

Returns the minimum value of the progress bar

`int getOrientation()`

Returns the orientation of the progress bar

`double getPercentComplete()`

Returns the percent complete for the progress bar

`String getString()`

Returns the current value of the progress string

`int getValue()`

Returns the current value of the progress bar

`void setMaximum(int max)`

Sets the maximum value for the progress bar

`void setMinimum(int min)`

Sets the minimum value for the progress bar

`void setString(String str)`

Sets the value of the progress string

`void setStringPainted(boolean b)`

Sets the `stringPainted` property that determines whether the progress bar should render a progress string or not

`void setValue(int value)`

Sets the progress bar's current value

19.19.2 Creating JProgressBar on JFrame

Progress bar can be created using the constructor methods of **JProgressBar**. Methods in the **JProgressBar** class can be used to make the progress bar useful to an application. After creating a progress bar, the minimum and maximum value of the progress has to be set. The interval between minimum and maximum will be converted to 100 units. The

percentComplete() method returns this value between 0 and 1. A progress bar will be continuously added on the progress bar. The **setValue()** method will automatically update the progress bar. There is no separate method required to update. We show in program 19.41, the technique of creating a horizontal progress bar, simulating a simple process, incrementing some value and making the progress bar propagate. A process like task is created by introducing a delay loop using **sleep()** method.

Program 19.41

```

/* This program illustrates the creation of a JProgressBar
   on a JFrame.
   somasundaramk@yahoo.com
*/

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

class JPbar1
    extends JFrame
    {
        Container conpan;
        int max = 1000;
        int min = 0;
        int value;
        int incr = (max - min) / 100;
        JProgressBar jpbar;

        JPbar1(String str)
        {
            super(str);
            conpan = getContentPane();
            conpan.setLayout(new FlowLayout());
            jpbar = new JProgressBar();
            jpbar.setMaximum(max);
            jpbar.setMinimum(min);
            jpbar.setValue(value);
            conpan.add(jpbar);
            setSize(250, 200);
            setVisible(true);
            addWindowListener(new WindowAdapter()
                {
                    public void windowClosing(WindowEvent we)
                    {
                        System.exit(0);
                    }
                }
            ));
        }
    }

```

```
// This delay is introduced to simulate a process
try
{
    for (int i = 0; i < 100; i++)
    {
        Thread.sleep(500);
        value = value + incr;
        jpbar.setValue(value);
    }
}
catch (InterruptedException e)
{
    ;
}

public static void main(String args [])
{
    new JPbar1("JProgressBar on JFrame ");
}
```

The above program gives the following output screens:

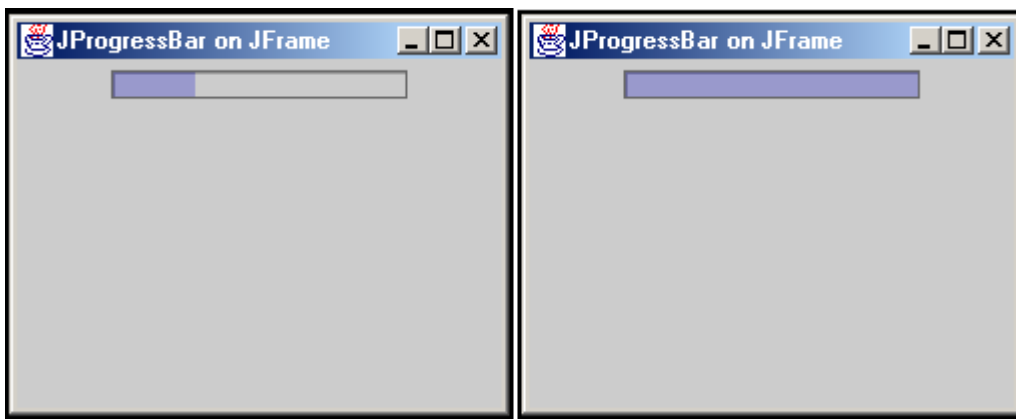


Fig.19.43 Output Screens for Program 19.41

The first window shows the progress bar status after progressing for about 20%. The second screen shows progress after 100%.



No separate method is needed to update a progress bar.

19.19.3 Using JProgressBar

Progress bars are needed wherever a time-consuming process takes place, to inform the user about the progress of the process. In the absence of such information, the user is not aware what is happening inside the program code.

In the following program 19.42, we show how a progress bar can be set up and the progress of the process is displayed on the progress bar. We first create a **JProgressBar**. By default, it is a horizontal progress bar. We then create a **JButton** with a text "Start". An empty info **JLabel** is created and added to the **contentPane** of **JFrame**. Another **JButton** with text Cancel is added to the **contentPane**. An action listener is registered with the start button. The time-consuming process and updation of the progress bar are placed in a separate thread and called in the **actionPerformed()** method. A separate thread is necessary as the action event thread will not process the action event until the time-consuming process is complete. When the user clicks Start button, the simulated time-consuming process and the progress bar updation take place. The start button is disabled from any more action on it. During the process, the Cancel button appears using which the process can be cancelled. At the completion of the process, the same Cancel button is made as Finish button.

Program 19.42

```

/* This program illustrates the use of JProgressBar.
   somasundaramk@yahoo.com
*/

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

class Pbarframe
    extends JFrame
    implements ActionListener
    {
        Container conpan;
        JButton begin, cancel;
        JLabel info;
        int max = 1000;
        int min = 0;
        int value;

        // divide the interval into 100 units
        int incr = (max - min) / 100;
        JProgressBar jpbar;
        Pbarframe(String str)
        {
            super(str);
            conpan = getContentPane();
            conpan.setLayout(new FlowLayout());
            info = new JLabel();
            begin = new JButton("Start ");
            cancel = new JButton("Cancel");
            jpbar = new JProgressBar();
            jpbar.setStringPainted(true);

```

```

conpan.add(begin);
conpan.add(jpbar);
conpan.add(info);
conpan.add(cancel);
cancel.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent ae)
    {
        System.exit(0);
    }
});
setSize(250, 200);
setVisible(true);
begin.addActionListener(this);
addWindowListener(new WindowAdapter()
{
    public void windowClosing(WindowEvent we)
    {
        System.exit(0);
    }
});
}
// This thread updates the progress bar values.
class Updatebar
    extends Thread
    {
    public void run()
    {
        begin.setEnabled(false);
        info.setText("Process in progress. Please
                                wait.....");

        jpbar.setMaximum(max);
        jpbar.setMinimum(min);
        jpbar.setValue(value);
        // This delay is introduced to simulate
        // a process
        try
        {
            for (int i = 0; i < 100; i++)
            {
                Thread.sleep(500);
                value = value + incr;
                jpbar.setValue(value);
            }
        }
        catch (Exception e)
        {
            ;
        }
        info.setText("Process complete");
    }
}

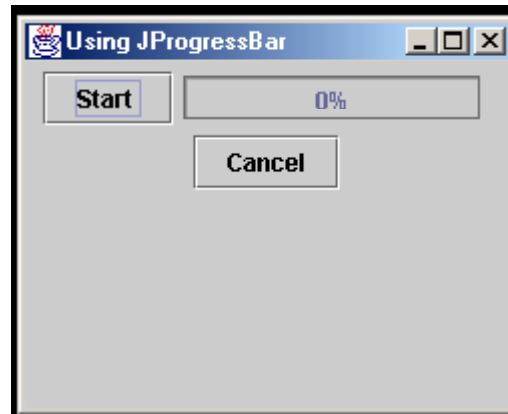
```

```

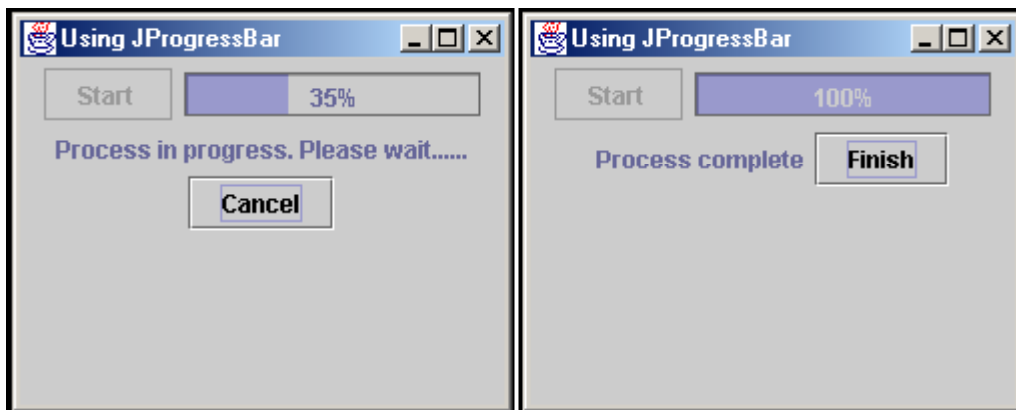
        cancel.setText("Finish");
    }
}
public void actionPerformed(ActionEvent ae)
{
    (new Updatebar()).start();
}
}
class JPbar2
{
    public static void main(String args [])
    {
        new Pbarframe("Using JProgressBar ");
    }
}

```

The above program gives the following output screens:



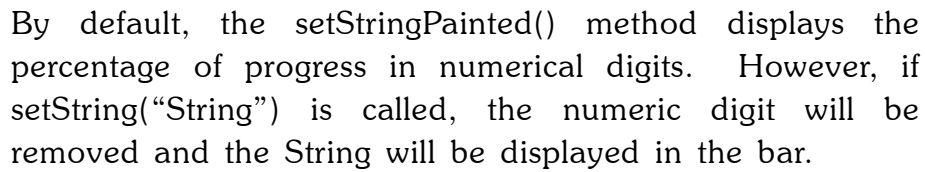
(a)



(b)

Fig.19.44 Output Screens for Program 19.42

- (a) The initial window
- (b) The first window shows the screen after 35% completion of the process. Note the Start button is disabled. The second window shows the screen after 100% completion of the process.



In the previous sections, we have seen how to add components on the container (like window). All the components like buttons, labels, check boxes were added at locations defined by default locations of the container. For placing the components in a container, layout managers are used. Each container has a default layout manager. Java defines several layout managers. A container can set the required layout manager using **setLayout()** method. Some of the layout managers are explained in this section.

BorderLayout class has a layout manager that divides its container into five regions and fits the components into it. The five regions are north, south, east, west, and center as given in fig. 19.45:



BorderLayout.CENTER	-	The center layout constraint (middle of container)
BorderLayout.EAST	-	The east layout constraint (right of container)

BorderLayout.NORTH	-	The north layout constraint (top of container)
BorderLayout.SOUTH	-	The south layout constraint (bottom of container)

Constructors

The **BorderLayout** class has the following constructors for creating border layout:

`BorderLayout()`

Creates a new border layout with no gap between components

`BorderLayout (int hgap, int vgap)`

Creates a border layout with the specified gaps between components; the hgap specifies the horizontal gap and vgap specifies the vertical gap.

Methods

Some of the methods defined in this class are given below.

`int getHgap()`

Returns the horizontal gap between components

`void setHgap(int hgap)`

Sets the horizontal gap between components

`public int getVgap()`

Returns the vertical gap between components

`void setVgap(int vgap)`

Sets the vertical gap between components

`void addLayoutComponent(Component comp, Object constraints)`

Adds the specified component to the layout using the specified constraint object; the constraints must be one of the constraints, NORTH, SOUTH, EAST, WEST or CENTER.

To add components to a container, the string constraints can be used as parameters. The following methods are defined in **Container** class:

`void add(Component comp, Object constraints)`

Adds the specified component to the end of the container

`void add(Component comp, Object constraints, int index)`

Adds the specified component to this container with the specified constraints at the specified index

In the following program 19.43, the use of border layout is given:

Program 19.43

```

/* This program illustrates the use of BorderLayout.
   somasundaramk@yahoo.com
   <applet code = Borderal width = 250 height = 150>
   </applet>
*/

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

public class Borderal
    extends JApplet
    {
        JButton north, south, east, west, center;
        Container conpan;
        public void init()
        {
            conpan = getContentPane();
            BorderLayout bl = new BorderLayout();
            conpan.setLayout(bl);
            north = new JButton("NORTH ");
            south = new JButton("SOUTH ");
            east = new JButton("EAST ");
            west = new JButton("WEST ");
            center = new JButton("CENTER ");
            conpan.add(north, BorderLayout.NORTH);
            conpan.add(south, BorderLayout.SOUTH);
            conpan.add(west, BorderLayout.WEST);
            conpan.add(east, BorderLayout.EAST);
            conpan.add(center, BorderLayout.CENTER);
            setVisible(true);
        }
    }

```

The output screen for program 19.43 is given below:

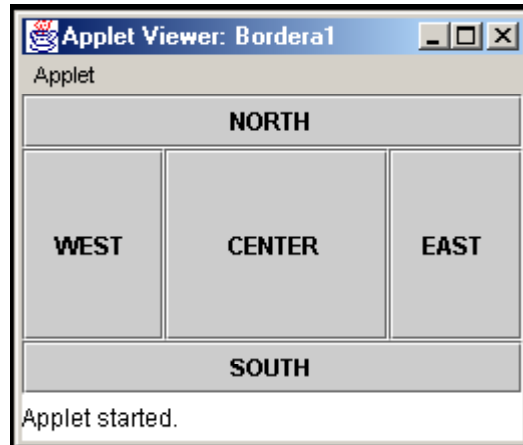


Fig.19.46 Output Screen for Program 19.43

In program 19.43, gaps can be introduced between components by inserting the following methods:

```
bl.setVgap(10);  
bl.setHgap(10);
```

After inserting the above statement, program 19.43 will give the following output:

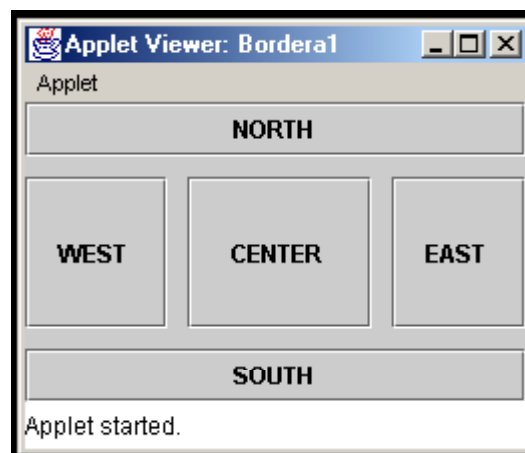


Fig.19.47 Output Screen for Program 19.43 with Vertical and Horizontal Gap

19.20.2 FlowLayout

In the **FlowLayout** manager, components are arranged in a left-to-right manner, like the flow of words in a line. When no more components fit in a line, it is taken to the next line. A small space is left between components. Components are arranged in a centered layout. It is the default layout for JApplet.

The **FlowLayout** class defines the following integer constants representing the different orientations:

FlowLayout.CENTER	-	Indicates that each row of component should be centered
FlowLayout.LEFT	-	Indicates that each row of component should be left-justified
FlowLayout.RIGHT	-	Indicates that each row of component should be right-justified

Constructors

The **FlowLayout** class defines the following constructors to create flow layout managers:

`FlowLayout()`

Creates a new flow layout with a centered alignment and a default 5 pixel horizontal and vertical gap

`FlowLayout(int align)`

Creates a new flow layout with the specified alignment with default 5 pixel horizontal and vertical gap; the value of the alignment must be one of the values, `FlowLayout.CENTER`, `FlowLayout.LEFT` or `FlowLayout.RIGHT`.

`FlowLayout(int align, int hgap, int vgap)`

Creates a new flow layout with the specified alignment, horizontal gap and vertical gap; the value of the alignment must be one of the values, `FlowLayout.CENTER`, `FlowLayout.LEFT` or `FlowLayout.RIGHT`.

Methods

The **FlowLayout** class has the following methods:

`int getAlignment()`

Returns the alignment value for this layout

`void setAlignment(int align)`

Sets the alignment value for this layout

`int getHgap()`

Returns the horizontal gap between components

`void setHgap(int hgap)`

Sets the horizontal gap between components

`int getVgap()`

Returns the vertical gap between components

`void setVgap(int vgap)`

Sets the vertical gap between components

In the program 19.44, the use of **FlowLayout** is given. **JRadioButtons** are created and grouped. These buttons are placed on a **JFrame** window. **JLabel** component is added to make the application suitable for a quiz.

Program 19.44

```

/* This program illustrates the use of FlowLayout manager.
   The JRadioButtons are placed on JPanel window.
   The JRadioButtons are grouped using ButtonGroup.
   somasundaramk@yahoo.com
*/
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

class Flowpanel
    extends JPanel
    implements ItemListener, ActionListener
    {
        JButton ok;
        JRadioButton rb1, rb2, rb3;
        ButtonGroup rbxg;
        String cmd;
        boolean state;

        Flowpanel()
        {
            Font fnt = new Font("Courier", Font.BOLD, 14);
            setFont(fnt);
            rbxg = new ButtonGroup();
            setLayout(new FlowLayout(FlowLayout.LEFT));
            ok = new JButton("OK");
            rb1 = new JRadioButton("C++", true);
            rb2 = new JRadioButton("Java", false);
            rb3 = new JRadioButton("Pascal", false);
            add(rb1);
            add(rb2);
            add(rb3);
            add(ok);
            rbxg.add(rb1);
            rbxg.add(rb2);
            rbxg.add(rb3);
            ok.addActionListener(this);
            rb1.addItemListener(this);
            rb2.addItemListener(this);
            rb3.addItemListener(this);
        }

        public void actionPerformed(ActionEvent ae)
        {
            cmd = ae.getActionCommand();

```

```

        repaint();
    }

    //implemented to meet the interface requirement
    public void itemStateChanged(ItemEvent ie) { }

    public void paintComponent(Graphics gp)
    {
        super.paintComponent(gp);
        state = rb3.isSelected();

        gp.drawString("Which of the above is not an OOP
                        language?", 20, 70);
        gp.drawString("Press OK after selection", 20, 90);

        if (cmd != null)
        {
            if (state)
                gp.drawString("Yes, you are right", 20, 110);

            else
                gp.drawString("Sorry, you are wrong", 20, 110);
        }
    }
}

class Flowframe
    extends JFrame
    {
        Container conpan;

        Flowframe(String str)
        {
            super(str);
            conpan = getContentPane();
            conpan.add(new Flowpanel());
            setSize(375, 200);
            setVisible(true);
            addWindowListener(new WindowAdapter()
            {
                public void windowClosing(WindowEvent e)
                {
                    System.exit(0);
                }
            });
        }
    }

class Flowlof
    {
        public static void main(String args [])
        {

```

```

        new Flowframe("Quiz using JRadioButton ");
    }
}

```

The above program gives the following output:

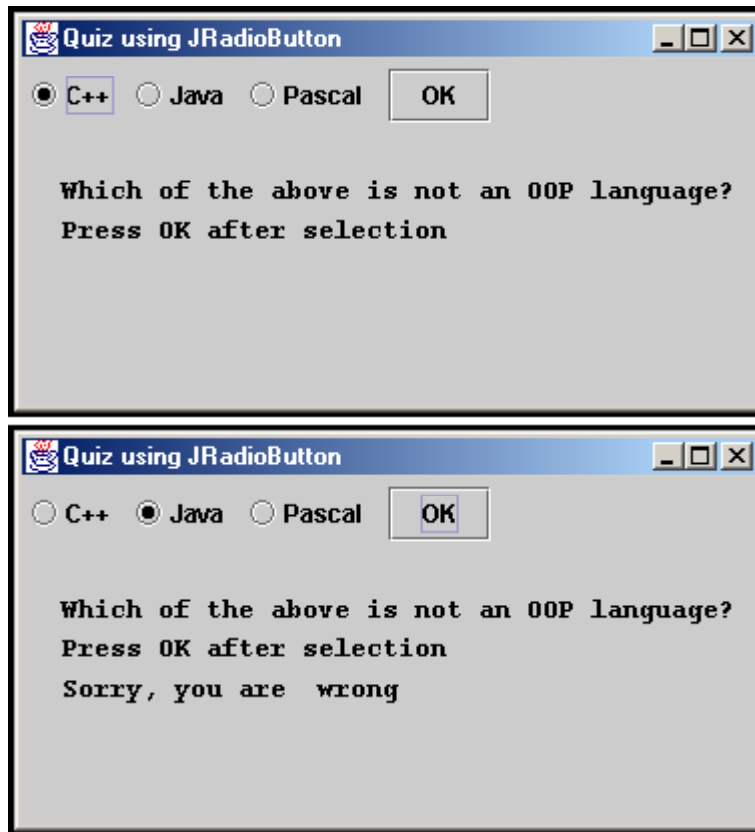


Fig.19.48 Output Screens for Program 19.44 The top screen shows the initial display. The bottom screen shows the screen after the user has selected an option.

19.20.3 GridLayout

The **GridLayout** class manager arranges the components into a rectangular, two-dimensional grid of rows and columns. The container is divided into equal-sized rectangles and one component is placed in each rectangle.

Constructors

The **GridLayout** class provides the following constructors to create grid layout:

`GridLayout()`

Creates a new grid layout with one column per component in a single row

`GridLayout(int rows, int columns)`

Creates a new grid layout with a specified number of rows and columns

`GridLayout(int rows, int columns, int hgap, int vgap)`

Creates a new grid layout with a specified number of rows and columns; the horizontal gap `hgap` is set between each column and at the left and right edges. The vertical gap `vgap` is set between each row and at the top and bottom.

Methods

The methods defined in this **GridLayout** class are given below:

`int getRows()`

Returns the number of rows in this layout

`void setRows(int rows)`

Sets the number of rows in this layout

`int getColumns()`

Returns the number of columns in this layout

`void setColumns(int Columns)`

Sets the number of columns in this layout

`int getHgap()`

Returns the horizontal gap between the components

`void setHgap(int hgap)`

Sets the horizontal gap between the components

`int getVgap()`

Returns the vertical gap between components

`void setVgap(int vgap)`

Sets the vertical gap between components

The following program 19.45 illustrates the use of **GridLayout** manager:

Program 19.45

```
/* This program illustrates the use of GridLayout manager.
   somasundaramk@yahoo.com
   <applet code = Gridapl width = 200 height = 150>
   </applet>
*/
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

class Gridpanel
    extends JPanel
    {
        Gridpanel()
    }
```

```
{
    setLayout(new GridLayout(2, 3));
    add(new JButton("A"));
    add(new JButton("B"));
    add(new JButton("C"));
    add(new JButton("D"));
    add(new JButton("E"));
    add(new JButton("F"));
}
}

class Gridl01
    extends JFrame
    {
        Container conpan;
        Gridl01(String str)
        {
            super(str);
            conpan = getContentPane();
            conpan.add(new Gridpanel());
            setSize(250, 200);
            setVisible(true);
            addWindowListener(new WindowAdapter()
            {
                public void windowClosing(WindowEvent e)
                {
                    System.exit(0);
                }
            });
        }
        public static void main(String args [])
        {
            new Gridl01("GridLayout");
        }
    }
```

The program 19.45 gives the following output:

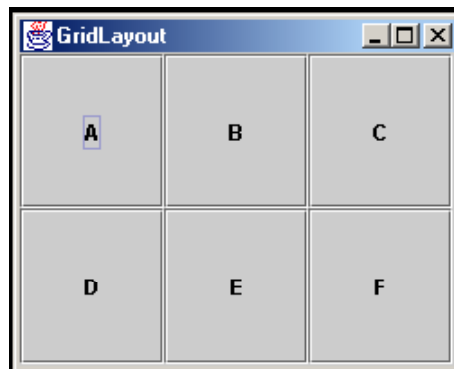


Fig.19.49 Output Screen for Program 19.45

Program 19.46 shows an application of grid layout. **JRadioButtons** are created and grouped using **ButtonGroup**. Whenever a user checks a button, the selection is displayed on the corresponding button on the second column. Checking a button creates an item event. The state of each button is obtained and the selection is displayed on the button.

Program 19.46

```
/* This program illustrates the use of GridLayout.
   JRadioButtons are placed on JPanel window.
   The JRadioButton components are arranged
   using the GridLayout manager.
   somasundaramk@yahoo.com
*/

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

class Gridap2
    extends JPanel
    implements ItemListener
    {
        JRadioButton rb1, rb2, rb3;
        ButtonGroup rbg;
        boolean st1, st2, st3;
        JButton pas, cpp, java;
        String pass, cpps, javas;

        Gridap2()
        {
            pass = "Pascal Selected ";
            cpps = " ";
            javas = " ";
            // make 4 rows and 2 columns layout
            GridLayout glo = new GridLayout(4, 2);
            setLayout(glo);
            rbg = new ButtonGroup();
            rb1 = new JRadioButton("C++", false);
            rb2 = new JRadioButton("Java", false);
            rb3 = new JRadioButton("Pascal", true);
            rb1.addItemListener(this);
            rb2.addItemListener(this);
            rb3.addItemListener(this);
            pas = new JButton(pass);
            cpp = new JButton(cpps);
            java = new JButton(javas);
            JLabel lbl = new JLabel("Check Your Option");
            rbg.add(rb1);
            rbg.add(rb2);
```

```

        rbg.add(rb3);
        add(rb1);
        add(cpp);
        add(rb2);
        add(java);
        add(rb3);
        add(pas);
        add(lbl);
    }

    // Update the screen when a check box state is changed
    public void itemStateChanged(ItemEvent ie)
    {
        st1 = rb1.isSelected();
        st2 = rb2.isSelected();
        st3 = rb3.isSelected();

        if (st1)
        {
            pass = "";
            cpps = "C++ Selected";
            javas = "";
            cpp.setText(cpps);
            java.setText(javas);
            pas.setText(pass);
        }

        if (st2)
        {
            pass = "";
            cpps = "";
            javas = "Java Selected";
            cpp.setText(cpps);
            java.setText(javas);
            pas.setText(pass);
        }

        if (st3)
        {
            pass = "Pascal Selected";
            cpps = "";
            javas = "";
            cpp.setText(cpps);
            java.setText(javas);
            pas.setText(pass);
        }
    }
}

class Gridlo2
    extends JFrame

```

```

{
Container conpan;
Gridlo2(String str)
{
    super(str);
    conpan = getContentPane();
    conpan.add(new Gridap2());
    setSize(300, 200);
    setVisible(true);
    addWindowListener(new WindowAdapter()
    {
        public void windowClosing(WindowEvent we)
        {
            System.exit(0);
        }
    });
}
public static void main(String args [])
{
    new Gridlo2("GridLayout Application");
}
}

```

The above program gives the following output:

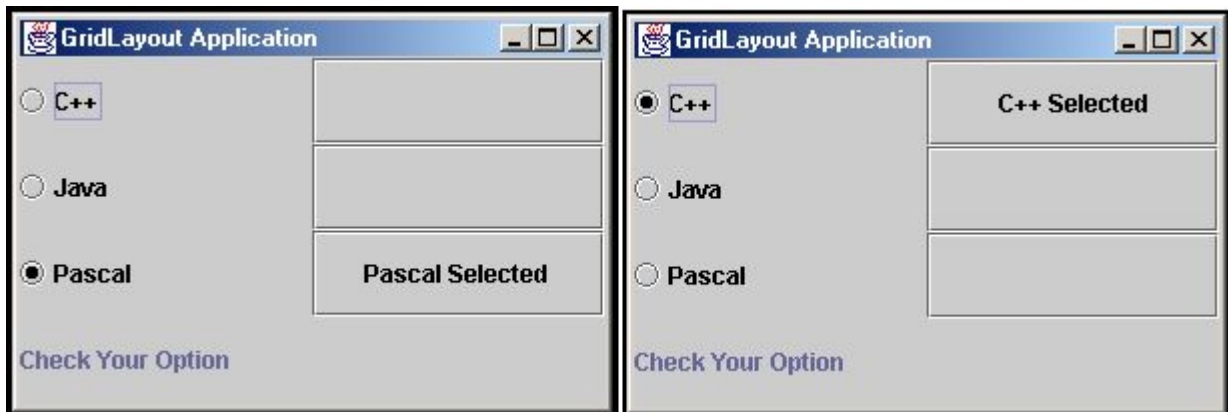


Fig.19.50 Output Screens for Program 19.46. The first window shows the initial screen. The second window shows the screen after the user has made a selection.

19.20.4 CardLayout

A card layout is a layout manager for a container. The components of the container are treated as a stack of cards. At any time, one card is visible. The first component added to the card is the visible component when the container is displayed (see fig.19.51).

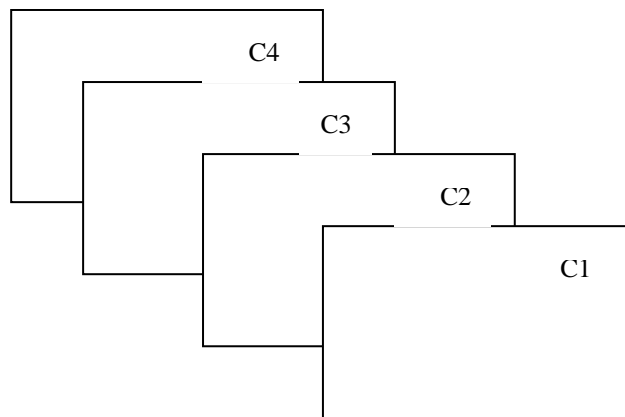


Fig.19.51 Card Layout

CardLayout provides methods to display a card of the user's choice. Cards can be displayed sequentially using suitable methods defined in **CardLayout**. To make a card, a **JPanel** is to be created and components are to be attached to it. Make the required number of such cards. Create a (master) **JPanel** and set that panel to have a **CardLayout** manager. Then add all the cards to the panel. Then add this main panel to **JApplet** window or **JFrame** window.

19.20.4.1 Creating CardLayout

CardLayout managers are created using constructors defined in it. The constructors are :

CardLayout()

Creates a new card layout with no gaps surrounding the card

CardLayout(int hgap, int vgap)

Creates a new card layout with the specified vertical and horizontal gaps; the horizontal gap appears at the top and bottom edges. The vertical gap appears at the left and right edges.

Methods

CardLayout class has several methods to handle the layout. Some of the methods are :

void addLayoutComponent(Component comp, Object Constraints)

Adds the specified component to this card layout's internal table of names; the Object specified must be a string. The card layout stores this string as a key-value pair that can be used for random access to a particular card. Using the show method, an application can display the component with the specified name.

```
void first(Container parent)
```

Shows the first card of the container

```
void next(Container parent)
```

Shows the next card of the specified container; after the last card, the first card is shown.

```
void previous(Container parent)
```

Shows the previous card of the specified container; the previous card after the first is the last card.

```
void last(Container parent)
```

Shows the last card of the container

In addition, the following methods defined in **Container** class are used to add components to container:

```
void add(Component comp, Object constraints)
```

Adds the specified component to the end of this container; the specified Object, usually a string, standing for the name of the component, is added to the layout manager's list

```
void add(Component comp, Object Constraints, int index)
```

Adds the specified component at the specified location; the object representing the name of the component is added to the layout manger's list.

19.20.4.2 Using CardLayout

To understand how to use the **CardLayout** manager and methods defined in it, we have developed the program 19.47 with two cards. Two cards, "fruits" and "animals", are prepared in a panel and attached to another panel set with **CardLayout** manager and added to the applet panel. By default, the first card will be displayed, whether the **show** method is called or not. Events registered on the button on the applet panel are listened and, for each action, the next card is called by using the next method.

Program 19.47

```
/* This program illustrates the use of CardLayout.
   Two cards are created on JPanel and attached
   to another JPanel. The master JPanel, which contains
   the two cards, is then added to a JFrame is available
   for their display.
   Cards can be attached to Panel only.

   somasundaramk@yahoo.com
*/
```

```

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

class Card1
    extends JPanel
    implements ActionListener
    {
        JPanel fruits, animals;
        JPanel cardholder = new JPanel();
        CardLayout cl;

        Card1()
        {
            setLayout(new FlowLayout());
            cl = new CardLayout();
            cardholder.setLayout(cl);

            // create the first card
            fruits = new JPanel();
            fruits.setLayout(new GridLayout(2, 3));
            JButton apl = new JButton("Apple");
            JButton ban = new JButton("Banana");
            JButton grp = new JButton("Grapes");
            JButton jac = new JButton("Jackfruit");
            JButton man = new JButton("Mango");
            JButton lem = new JButton("Lemon");
            JButton ora = new JButton("Orange");
            fruits.add(apl);
            fruits.add(ban);
            fruits.add(grp);
            fruits.add(jac);
            fruits.add(man);
            fruits.add(ora);

            //create the second card
            animals = new JPanel();
            animals.setLayout(new GridLayout(2, 3));
            JButton ant = new JButton("Antelope");
            JButton bis = new JButton("Bison");
            JButton gor = new JButton("Gorilla");
            JButton ele = new JButton("Elephant");
            JButton cow = new JButton("Cow");
            JButton lio = new JButton("Lion");
            JButton tig = new JButton("Tiger");
            JButton cam = new JButton("Camel");
            animals.add(ant);
            animals.add(bis);
            animals.add(gor);
            animals.add(ele);

```

```

        animals.add(cow);
        animals.add(tig);
        animals.add(cam);
        // add the two cards to the main panel
        cardholder.add(fruits, "Card1");
        cardholder.add(animals, "Card2");

        // add the main panel to panel
        add(cardholder);

        //show the second card as initial screen
        cl.show(cardholder, "Card2");

        // add a Next Card Button to the applet panel
        JButton nxt = new JButton("Next card");
        add(nxt);
        nxt.addActionListener(this);
    }
    //show the next card for each press of the Next card
    //button
    public void actionPerformed(ActionEvent ae)
    {
        cl.next(cardholder);
    }
}
class Cardlo
    extends JFrame
    {
        Container conpan;
        Cardlo(String str)
        {
            super(str);
            conpan = getContentPane();
            conpan.add(new Card1());
            setSize(370, 150);
            addWindowListener(new WindowAdapter()
                {
                    public void windowClosing(WindowEvent we)
                    {
                        System.exit(0);
                    }
                });
        }
        public static void main(String args [])
        {
            JFrame clfrm = new Cardlo("CardLayout on JFrame");
            clfrm.setVisible(true);
        }
    }

```

The above program gives the following output:

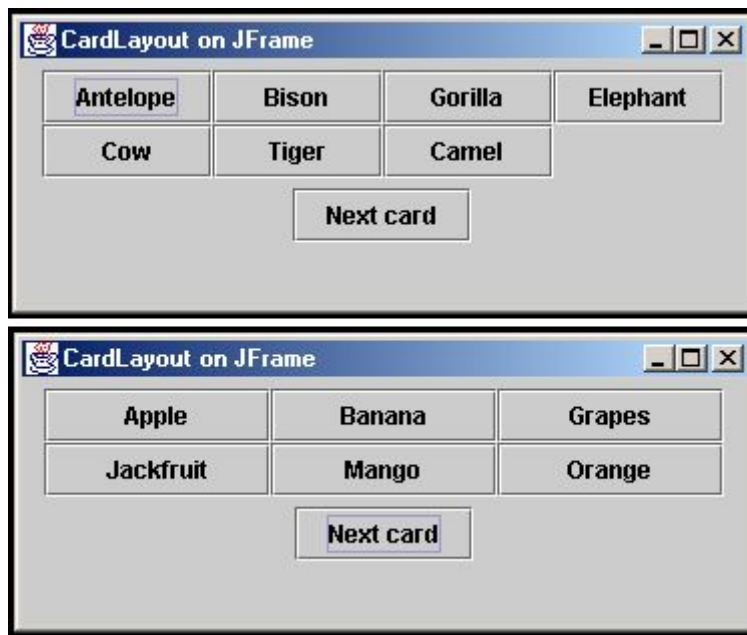


Fig.19.52 Output Screens for Program 19.47 The top window shows the initial display. The bottom window shows the screen after the user clicked Next button.

After reading this chapter, you should have learned the following concepts:

- ➞ Creation of various JComponents in Swing
 - ➞ Adding JComponents on JFrame, JApplet and JPanel
 - ➞ Registering various listeners to the JComponents
 - ➞ Carrying out the desired process when a component is clicked
 - ➞ Using different layout managers to layout the JComponent
 - ➞ Using JComponents in various applications
-

In the next chapter, the basics of networking is explained.

Worked Out Problem–19

19.1w. Design a screen to input the bill details and prepare an on-line bill.

Name of Customer	:	25 Characters - Key Input
Item	:	Selectable from 10 Items - Use Combo Box
Unit Price	:	Floating Point Automatically selected when Item is selected
Qty. Sold	:	Integer - Key Input
Amount	:	Unit Price x Qty. Sold
Tax Rate	:	Floating Point xx.xx Automatically selected when Item is selected.
Total Cost	:	$\text{Amount}(1 + \text{Tax Rate}/100)$

Program 19.1w

```

/* This program illustrates how to make data entry
   and prepare a bill in a company. Once a bill is
   prepared, it is written into a disk file.

   somasundaramk@yahoo.com
*/

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

class Combopanel
    extends JPanel
    implements ItemListener, ActionListener
    {
        JTextField namef, pricef, qtyf, taxf, amountf, costf;
        JTextField keysource;
        JLabel namel, iteml, qtyl, pricel, amountl, taxl,
            costl;

        JButton cal, clear;
        String itemnam [] = new String[10];
        double itemprice [] = new double[10];
        double taxtable [] = new double[10];
        JComboBox itemcombo;
        JComboBox combosource;
        String selectitem, selectprice;
    }

```

```

int selectindex;
int qtysold;
double taxrate, unitprice, amount, totalcost;

// date variables
int day, month, year;
String date;
JLabel date1, datefield;
ObjectOutputStream objos;

Combopanel()
{
    GregorianCalendar calendr = new
        GregorianCalendar();
    day = calendr.get(Calendar.DATE);
    month = calendr.get(Calendar.MONTH);
    month += 1; // January is 0 hence this conversion
    year = calendr.get(Calendar.YEAR);

    // create file for storing bill deatils in a file
    try
    {
        FileOutputStream fos = new
            FileOutputStream("bill.dat", true);
        objos = new ObjectOutputStream(fos);
    }
    catch (IOException ioe)
    {
        JOptionPane.showMessageDialog(this, "File
            opening problem");
    }

    setLayout(new GridLayout(10, 2));
    String itemname [] =
    {
        "Bolt", "Cutter", "Emmery Paper", "Hammer",
        "Latch", "Nut", "Paint", "Pipe", "Screw", "Rings"
    };

    double price [] = {25.75, 125.50, 8.50,
        200.0, 25.0, 2.0, 150.75, 300.0, 1.50, 4.50
    };

    double tax [] =
    {
        4.0, 8.5, 0, 5.0, 0, 0, 12.5,
        10.0, 0, 2.0
    };

    itemcombo = new JComboBox();
    for (int i = 0; i < itemname.length; i++)
    {

```

```

        itemcombo.addItem(itemname[i]);
        itemnam[i] = itemname[i];
        itemprice[i] = price[i];
        taxtable[i] = tax[i];
    }

    itemcombo.addItemListener(this);
    JLabel company=new JLabel("SOMSON HARDWARE MART ");
    JLabel adrs1 = new JLabel("21-New Street,Chennai-24");
    JLabel datel = new JLabel("Date      ");
    date = " " + day + "-" + month + "-" + year;
    datefield = new JLabel(date);
    JLabel namel = new JLabel("Customer Name");
    JTextField namef = new JTextField(25);
    JLabel iteml = new JLabel("Item ");
    JLabel pricel = new JLabel("Unit Price ");
    JTextField pricef = new JTextField(" " + price[0], 6);
    pricef.setEditable(false);
    JLabel qty1 = new JLabel("Quantity Sold ");
    JTextField qtyf = new JTextField("0", 3);
    JLabel amountl = new JLabel("Amount Rs.  ");
    JTextField amountf = new JTextField("0", 6);
    amountf.setEditable(false);
    JLabel taxl = new JLabel("Tax Rate  ");
    JTextField taxp = new JTextField("0", 5);
    taxp.setEditable(false);
    JLabel costl = new JLabel("Total Cost Rs.  ");
    JTextField costf = new JTextField("0", 6);
    costf.setEditable(false);
    JButton cal = new JButton("Calculate");
    cal.addActionListener(this);
    JButton clear = new JButton("Clear Entry");
    clear.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent ae)
        {
            namef.setText("");
            qtyf.setText("");
            taxp.setText("");
            amountf.setText("");
            costf.setText("");
        }
    });

    add(company);
    add(adrs1);
    add(datel);
    add(datefield);
    add(namel);
    add(namef);

```

```
        add(iteml);
        add(itemcombo);
        add(pricel);
        add(pricef);
        add(qtyl);
        add(qtyf);
        add(amountl);
        add(amountf);
        add(taxl);
        add(taxf);
        add(costl);
        add(costf);
        add(cal);
        add(clear);
    }

    public void itemStateChanged(ItemEvent ie)
    {
        combosource = (JComboBox)ie.getSource();
        selectitem = (String)combosource.getSelectedItem();
        selectindex = combosource.getSelectedIndex();
        pricef.setText(" " + itemprice[selectindex]);
        taxf.setText(" " + taxtable[selectindex]);
    }

    public void actionPerformed(ActionEvent ae)
    {
        try
        {
            qtysold = Integer.parseInt(qtyf.getText());
        }
        catch (NumberFormatException ne)
        {
            JOptionPane.showMessageDialog(this, "Quantity
                sold should be an integer", "Warning",
                JOptionPane.WARNING_MESSAGE);
        }

        unitprice = itemprice[selectindex];

        if (qtysold == 0)
            JOptionPane.showMessageDialog(this, "Quantity
                sold is zero. Please check the entry",
                "Warning", JOptionPane.WARNING_MESSAGE);

        amount = qtysold * unitprice;
        amountf.setText("" + amount);
        taxrate = taxtable[selectindex];
        totalcost = amount * (1 + taxrate / 100);
        costf.setText("" + totalcost);
    }
}
```

```

        // write the data to the file
        String custname = namef.getText();
        CreateRecord objrec = new CreateRecord(custname,
            date, selectitem, unitprice, qtysold,
            taxrate, amount);
        new AddRecord(objos, objrec, this);
    }
}

class CreateRecord
    implements Serializable
    {
        String customername, itemname, date;
        double itemcost, taxrate, amount;
        int qtysold;
        CreateRecord(String cnam, String dat, String itemnam,
            double itcost, int qsold, double tax, double amt)
        {
            customername = cnam;
            date = dat;
            itemname = itemnam;
            itemcost = itcost;
            qtysold = qsold;
            taxrate = tax;
            amount = amt;
        }
    }

class AddRecord
    {
        AddRecord(ObjectOutputStream ous, Object obj, JPanel jp)
        {
            try
            {
                ous.writeObject(obj);
            }
            catch (IOException ioe)
            {
                JOptionPane.showMessageDialog(jp, "File
                    writing problem");
            }
        }
    }

class Dfcomboframe
    extends JFrame
    {
        Container conpan;

        Dfcomboframe(String str)
    }

```

```

        {
            super(str);
            conpan = getContentPane();
            setSize(340, 250);
            addWindowListener(new WindowAdapter()
            {
                public void windowClosing(WindowEvent we)
                {
                    System.exit(0);
                }
            });

            conpan.add(new Combopanel());
            setVisible(true);
        }

class Probl91
{
    public static void main(String args [])
    {
        new Dfcomboframe("BILL PREPARATION ");
    }
}

```

The above program gives the following output screens:

The image shows two side-by-side screenshots of a Java Swing window titled "BILL PREPARATTION" for "SOMSON HARDWARE MART 21-New Street, Chennai -24".

Left Screenshot: The window contains a form with the following fields and values:

Date	4-2-2007
Customer Name	S.P.Kumar
Item	Hammer
Unit Price	200.0
Quantity Sold	2
Amount Rs.	400.0
Tax Rate	5.0
Total cost Rs.	420.0

At the bottom, there are two buttons: "Calculate" and "Clear Entry".

Right Screenshot: This screenshot shows the same window, but the "Item" field is now a list box containing the following items: Bolt, Cutter, Emmery Paper, Hammer (highlighted), Latch, Nut, Paint, and Pipe. The other fields and buttons remain the same.

(a)

BILL PREPARATION	
SOMSON HARDWARE MART 21-New Street, Chennai -24	
Date	4-2-2007
Customer Name	S.P.Kumar
Item	Hammer
Unit Price	200.0
Quantity Sold	2
Amount Rs.	400.0
Tax Rate	5.0
Total cost Rs.	420.0
Calculate	Clear Entry

(b)

Fig.19.53 Output Screens for Program 19.1W

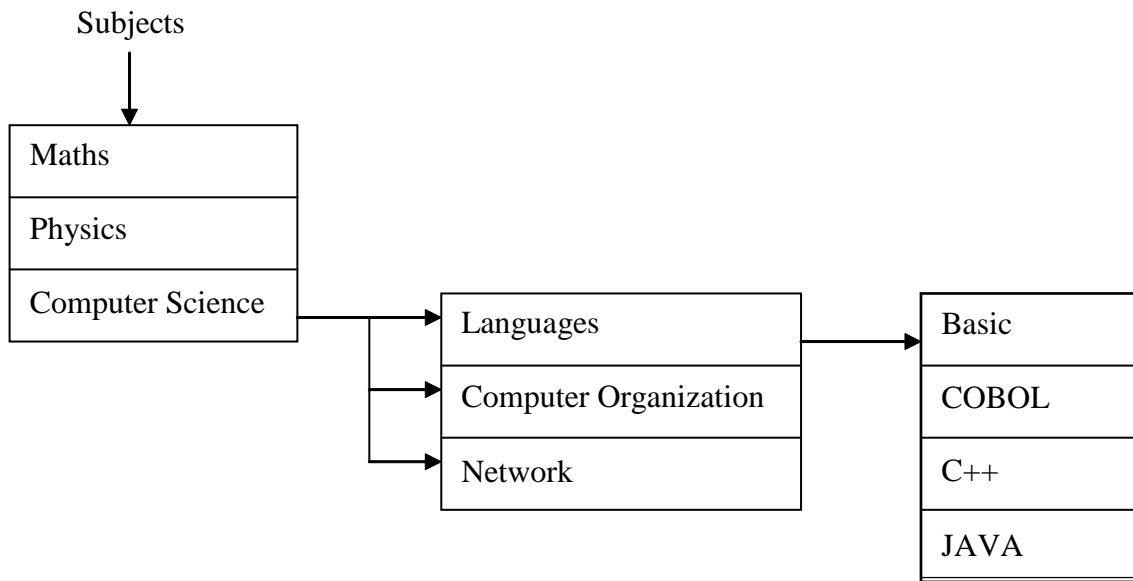
Exercise-19

I. Fill in the blanks

- 19.1 The Swing classes are part of _____ .
- 19.2 The top level windows in Swing are _____, _____, _____, _____.
- 19.3 JComponents are added on the _____ .
- 19.4 JFrame can be closed using Close Window icon. True/False
- 19.5 Images can be used in creating JButton. True/False
- 19.6 To arrange the components of a container, all top level containers are to be set with a _____ .
- 19.7 In JLabel _____ line(s) of text can be used as label.
- 19.8 JToggleButton is a _____ state button.
- 19.9 A JList displays a _____ which can be selected.
- 19.10 A ScrollPane is used to view _____ of a large text or image.
- 19.11 A JTextField can handle text with _____ font and _____ color.
- 19.12 A text in a JList _____ be edited while the text in a JTextField _____ be edited.
- 19.13 A JComboBox is like the combination of _____ .
- 19.14 A modal JDialog _____ input to other window.
- 19.15 Simple pre-configured dialogs can be created quickly using _____.
- 19.16 For creating dialogs for files and directories _____ is used.
- 19.17 The amount of work carried out by a process can be displayed using _____ .

II. Write programs for the following problems:

- 19.18 Create 4 JButtons with labels 1, 2, 3 and 4 and display them on a JFrame window. When you push a button, the number on it is to be displayed on a JLabel. When more number of buttons are pressed, the label on them is to be printed as a single sequence of digits on the JLabel.
- 19.19 An objective-type question has three choices. One of the choices is the correct answer. There are five objective-type questions. Each selection of correct answer is given 2 marks. A wrong answer is awarded - 1/2 mark. Each question is to be answered in a 30-second duration. At the end of the session, the total score is to be displayed. Write a Java program with the use of appropriate JCheckBox to implement the above problem.
- 19.20 It is required to get the name and date of birth of the user. Put up appropriate JTextField that enables the user to type in his/her name. Make use of separate JComboBox for year, month and date, which are to be selected. After the selection, the selected items are to be displayed on the screen. Write a program to implement the above requirement.
- 19.21 Set up a text area with a 2-line visible window. Set it as editable. Allow a user to type in 3 to 5 lines. Set up a second text field with a 2-line visible window and make it as non-editable. Select characters in the first text field and form valid English words to appear inside the second text area. Create 10 such words with a blank space between each word.
- 19.22 Write a program to simulate a calculator using JButton and JTextField.
- 19.23 Write a program to prepare an electronic spread sheet with as many functions as you can.
- 19.24 Prepare a monthly calendar for one year with one card per month using CardLayout.
- 19.25. Write JMenu for the following structure:



- 19.26. Set up CardLayout in a panel. Prepare cards to display one question as JLabel, with four JCheckBox. Allow the user to check the boxes. Whenever a choice is checked, a JDialog is to be displayed whether the choice is the correct answer or not. Repeat it for 5 cards and show the cards sequentially.

* * * * *