Window Fundamentals
In this Unit, you will learn how to create and manage windows, manage fonts, output text, and utilize graphics. It also describes the various controls, such as scroll bars and push buttons, supported by the AWT. It also explains further aspects of Java’s event handling mechanism. This unit also examines the AWT’s imaging subsystem and animation. Although a common use of the AWT is in applets, it is also used to create stand-alone windows that run in a GUI environment, such as Windows. For the sake of convenience, most of the examples in the unit chapter are contained in applets. To run them, you need to use an applet viewer or a Java-compatible web browser. A few examples will demonstrate the creation of stand-alone, windowed programs.

AWT Classes[RGPV Dec 2014(3)]
The AWT classes are contained in the java.awt package. It is one of Java’s largest packages. Fortunately, because it is logically organized in a top-down, hierarchical fashion, it is easier to understand and use than you might at first believe. Table 23-1 lists some of the many AWT classes.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWTEvent</td>
<td>Encapsulates AWT events.</td>
</tr>
<tr>
<td>AWTEventMulticaster</td>
<td>Dispatches events to multiple listeners.</td>
</tr>
<tr>
<td>BorderLayout</td>
<td>The border layout manager. Border layouts use five components: North, South, East, West, and Center.</td>
</tr>
<tr>
<td>Button</td>
<td>Creates a push button control.</td>
</tr>
<tr>
<td>Canvas</td>
<td>A blank, semantics-free window.</td>
</tr>
<tr>
<td>CardLayout</td>
<td>The card layout manager. Card layouts emulate index cards, only the one on top is showing.</td>
</tr>
<tr>
<td>Checkbox</td>
<td>Creates a check box control.</td>
</tr>
<tr>
<td>CheckboxGroup</td>
<td>Creates a group of check box controls.</td>
</tr>
<tr>
<td>CheckboxMenuItem</td>
<td>Creates an on/off menu item.</td>
</tr>
<tr>
<td>Choice</td>
<td>Creates a popup list.</td>
</tr>
<tr>
<td>Color</td>
<td>Manages colors in a portable, platform-independent fashion.</td>
</tr>
<tr>
<td>Component</td>
<td>An abstract superclass for various AWT components.</td>
</tr>
<tr>
<td>Container</td>
<td>A subclass of Component that can hold other components.</td>
</tr>
<tr>
<td>Cursor</td>
<td>Encapsulates a bitmapped cursor.</td>
</tr>
<tr>
<td>Dialog</td>
<td>Creates a dialog window.</td>
</tr>
<tr>
<td>Dimension</td>
<td>Specifies the dimensions of an object. The width is stored in width, and the height is stored in height.</td>
</tr>
<tr>
<td>Event</td>
<td>Encapsulates events.</td>
</tr>
<tr>
<td>EventQueue</td>
<td>Queues events.</td>
</tr>
<tr>
<td>FileDialog</td>
<td>Creates a window from which a file can be selected.</td>
</tr>
<tr>
<td>FlowLayout</td>
<td>The flow layout manager. Flow layout positions components left to right, top to bottom.</td>
</tr>
<tr>
<td>Font</td>
<td>Encapsulates a type font.</td>
</tr>
<tr>
<td>FontMetrics</td>
<td>Encapsulates various information related to a font. This information helps you display text in a window.</td>
</tr>
<tr>
<td>Frame</td>
<td>Creates a standard window that has a title bar, resize corners, and a menu bar.</td>
</tr>
</tbody>
</table>
Window Fundamentals
The AWT defines windows according to a class hierarchy that adds functionality and specificity with each level. The two most common windows are those derived from Panel, which is used by applets, and those derived from Frame, which creates a standard application window. Much of the functionality of these windows is derived from their parent classes. Thus, a description of the class hierarchies relating to these two classes is fundamental to their understanding. Figure 23-1 shows the class hierarchy for Panel and Frame. Let’s look at each of these classes now.

Component[RGPV Dec 2014(3)]

At the top of the AWT hierarchy is the Component class. Component is an abstract class that encapsulates all of the attributes of a visual component. All user interface elements such as, buttons, check boxes, pop-up menus, text fields etc., which are displayed on the screen and that interact with the user are subclasses of Component. It defines over a hundred public methods that are responsible for managing events, such as mouse and keyboard input, positioning and sizing the window, and repainting. A Component object is responsible for remembering the current foreground and background colors and the currently selected text.
The `Container` class is a subclass of `Component`. It has additional methods that allow other `Component` objects to be nested within it. Other `Container` objects can be stored inside of a `Container` (since they are themselves instances of `Component`). This makes for a multileveled containment system. A container is responsible for laying out (that is, positioning) any components that it contains. It includes Applet Window, panels, dialog boxes, frames etc.

**Panel**
The `Panel` class is a concrete subclass of `Container`. It doesn’t add any new methods; it simply implements `Container`. A `Panel` may be thought of as a recursively nestable, concrete screen component. `Panel` is the superclass for `Applet`. When screen output is directed to an applet, it is drawn on the surface of a `Panel` object. In essence, a `Panel` is a window that does not contain a title bar, menu bar, or border. This is why you don’t see these items when an applet is run inside a browser. When you run an applet using an applet viewer, the applet viewer provides the title and border. Other components can be added to a `Panel` object by its `add()` method (inherited from `Container`). Once these components have been added, you can position and resize them manually using the `setLocation()`, `setSize()`, `setPreferredSize()`, or `setBounds()` methods defined by `Component`.

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

**Frame**
`Frame` encapsulates what is commonly thought of as a “window.” It is a subclass of `Window` and has a title bar, menu bar, borders, and resizing corners. If you create a `Frame` object from within an applet, it will contain a warning message, such as “Java Applet Window,” to the user that an applet window has been created. This message warns users that the window they see was started by an applet and not by software running on their computer. (An applet that could...
masquerade as a host-based application could be used to obtain passwords and other sensitive information without the user’s knowledge.) When a **Frame** window is created by a stand-alone application rather than an applet, a normal window is created.

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

<table>
<thead>
<tr>
<th>S.NO</th>
<th>RGPV QUESTIONS</th>
<th>YEAR</th>
<th>MARKS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q.1</td>
<td>What is AWT. Differentiate between component class &amp; container class.</td>
<td>Dec 2014</td>
<td>3</td>
</tr>
<tr>
<td>Q.2</td>
<td>Explain in Brief container and Components in AWT Package.</td>
<td>DEC 2009</td>
<td>10</td>
</tr>
<tr>
<td>Q.3</td>
<td>Using AWT classes, Create a Closable Window to receive the following inputs, i.e, username, sex, qualification, address with submit and reset button</td>
<td>DEC 2009</td>
<td>12</td>
</tr>
</tbody>
</table>
Working with Frame Windows

After the applet, the type of window you will most often create is derived from Frame. You will use it to create child windows within applets, and top-level or child windows for stand-alone applications. As mentioned, it creates a standard-style window.

Here are two of Frame’s constructors:

Frame()
Frame(String title)

The first form creates a standard window that does not contain a title. The second form creates a window with the title specified by title. Notice that you cannot specify the dimensions of the window. Instead, you must set the size of the window after it has been created.

Creating a Frame Window in an Applet

Creating a new frame window from within an applet is actually quite easy. First, create a subclass of Frame. Next, override any of the standard applet methods, such as init(), start(), and stop(), to show or hide the frame as needed. Finally, implement the windowClosing() method of the WindowListener interface, calling setVisible(false) when the window is closed. Once you have defined a Frame subclass, you can create an object of that class. This causes a frame window to come into existence, but it will not be initially visible. You make it visible by calling setVisible(). When created, the window is given a default height and width.

You can set the size of the window explicitly by calling the setSize() method. The following applet creates a subclass of Frame called SampleFrame. A window of this subclass is instantiated within the init() method of AppletFrame. Notice that SampleFrame calls Frame’s constructor. This causes a standard frame window to be created with the title passed in title. This example overrides the applet’s start() and stop() methods so that they show and hide the child window, respectively. This causes the window to be removed automatically when you terminate the applet, when you close the window, or, if using a browser, when you move to another page. It also causes the child window to be shown when the browser returns to the applet.

// Create a child frame window from within an applet.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="AppletFrame" width=300 height=50>
// Create a subclass of Frame.
class SampleFrame extends Frame {
    SampleFrame(String title) {
        super(title);
        // create an object to handle window events
        MyWindowAdapter adapter = new MyWindowAdapter(this);
        // register it to receive those events
        addWindowListener(adapter);
    }
    public void paint(Graphics g) {
        g.drawString("This is in frame window", 10, 40);
    }
}

class MyWindowAdapter extends WindowAdapter {
    SampleFrame sampleFrame;
    public MyWindowAdapter(SampleFrame sampleFrame) {
        this.sampleFrame = sampleFrame;
    }
    public void windowClosing(WindowEvent we) {
        sampleFrame.setVisible(false);
    }
}

// Create frame window.

public class AppletFrame extends Applet {
    Frame f;
    public void init() {
        f = new SampleFrame("A Frame Window");
        f.setSize(250, 250);
        f.setVisible(true);
    }
    public void start() {
        f.setVisible(true);
    }
    public void stop() {
        f.setVisible(false);
    }
    public void paint(Graphics g) {
        g.drawString("This is in applet window", 10, 20);
    }
}
Sample output from this program is shown here.

<table>
<thead>
<tr>
<th>S.NO</th>
<th>RGPF QUESTIONS</th>
<th>YEAR</th>
<th>MARKS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q.1</td>
<td>Explain the life cycle of applet.</td>
<td>Dec 2014</td>
<td>2</td>
</tr>
</tbody>
</table>

we don't take any liability for the notes correctness.
http://www.rgpvonline.com
Handling Events in a Frame Window

Since Frame is a subclass of Component, it inherits all the capabilities defined by Component. This means that you can use and manage a frame window just like you manage an applet’s main window. For example, you can override paint() to display output, call repaint() when you need to restore the window, and add event handlers. Whenever an event occurs in a window, the event handlers defined by that window will be called. Each window handles its own events. For example, the following program creates a window that responds to mouse events. The main applet window also responds to mouse events. When you experiment with this program, you will see that mouse events are sent to the window in which the event occurs.

```java
// Handle mouse events in both child and applet windows.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

/*
   <applet code="WindowEvents" width=300 height=50>
</applet>
*/

// Create a subclass of Frame.
class SampleFrame extends Frame implements MouseListener, MouseMotionListener {
    String msg = "";
    int mouseX=10, mouseY=40;
    int movX=0, movY=0;
    SampleFrame(String title) {
        super(title);
        // register this object to receive its own mouse events
        addMouseListener(this);
        addMouseMotionListener(this);
        // create an object to handle window events
        MyWindowAdapter adapter = new MyWindowAdapter(this);
        // register it to receive those events
        addWindowListener(adapter);
    }
    // Handle mouse clicked.
    public void mouseClicked(MouseEvent me) {
    }
    // Handle mouse entered.
    public void mouseEntered(MouseEvent evtObj) {
        // save coordinates
        mouseX = 10;
    }
    // Handle mouse moved.
    public void mouseMoved(MouseEvent me) {
    }
    // Handle mouse exited.
    public void mouseExited(MouseEvent evtObj) {
    }
    // Handle mouse dragged.
    public void mouseDragged(MouseEvent me) {
    }
    // Handle mouse pressed.
    public void mousePressed(MouseEvent me) {
    }
    // Handle mouse released.
    public void mouseReleased(MouseEvent me) {
    }
}
```

we don't take any liability for the notes correctness.

http://www.rgpvonline.com
mouseY = 54;
msg = "Mouse just entered child.";
repaint();
}
// Handle mouse exited.
public void mouseExited(MouseEvent evtObj) {
// save coordinates
mouseX = 10;
mouseY = 54;
msg = "Mouse just left child window."
repaint();
}
// Handle mouse pressed.
public void mousePressed(MouseEvent me) {
// save coordinates
mouseX = me.getX();
mouseY = me.getY();
msg = "Down";
repaint();
}
// Handle mouse released.
public void mouseReleased(MouseEvent me) {
// save coordinates
mouseX = me.getX();
mouseY = me.getY();
msg = "Up";
repaint();
}
// Handle mouse dragged.
public void mouseDragged(MouseEvent me) {
// save coordinates
movX = me.getX();
movY = me.getY();
msg = "*";
repaint();
}
// Handle mouse moved.
public void mouseMoved(MouseEvent me) {
// save coordinates
movX = me.getX();
movY = me.getY();
repaint(0, 0, 100, 60);
public class WindowEvents extends Applet implements MouseListener, MouseMotionListener {
    SampleFrame f;
    String msg = "";
    int mouseX = 0, mouseY = 10;
    int movX = 0, movY = 0;

    // Create a frame window.
    public void init() {
        f = new SampleFrame("Handle Mouse Events");
        f.setSize(300, 200);
        f.setVisible(true);
        // register this object to receive its own mouse events
        addMouseListener(this);
        addMouseMotionListener(this);
    }

    // Remove frame window when stopping applet.
    public void stop() {
        f.setVisible(false);
    }

    // Show frame window when starting applet.
    public void start() {
        f.setVisible(true);
    }

    // Handle mouse clicked.
    public void mouseClicked(MouseEvent me) {
    }

    // Handle mouse entered.
    public void mouseEntered(MouseEvent me) {
    }

    // Applet window.
    public class MyWindowAdapter extends WindowAdapter {
        SampleFrame sampleFrame;
        public MyWindowAdapter(SampleFrame sampleFrame) {
            this.sampleFrame = sampleFrame;
        }

        public void windowClosing(WindowEvent we) {
            sampleFrame.setVisible(false);
        }
    }

    // Public void paint(Graphics g) {
    // g.drawString(msg, mouseX, mouseY);
    // g.drawString("Mouse at ", movX, " + movY, 10, 40);
    // }

    public void paint(Graphics g) {
        g.drawString(msg, mouseX, mouseY);
        g.drawString("Mouse at " + movX + ", " + movY, 10, 40);
    }
}

// save coordinates
mouseX = 0;
m mouseY = 24;
msg = "Mouse just entered applet window.";
repaint();
}
// Handle mouse exited.
public void mouseExited(MouseEvent me) {
// save coordinates
mouseX = 0;
m mouseY = 24;
msg = "Mouse just left applet window.";
repaint();
}
// Handle button pressed.
public void mousePressed(MouseEvent me) {
// save coordinates
mouseX = me.getX();
m mouseY = me.getY();
msg = "Down";
repaint();
}
// Handle button released.
public void mouseReleased(MouseEvent me) {
// save coordinates
mouseX = me.getX();
m mouseY = me.getY();
msg = "Up";
repaint();
}
// Handle mouse dragged.
public void mouseDragged(MouseEvent me) {
// save coordinates
mouseX = me.getX();
m mouseY = me.getY();
movX = me.getX();
movY = me.getY();
msg = "*";
repaint();
}
// Handle mouse moved.
public void mouseMoved(MouseEvent me) {
// save coordinates
movX = me.getX();
movY = me.getY();
repaint(0, 0, 100, 20);
}

// Display msg in applet window.
public void paint(Graphics g) {
    g.drawString(msg, mouseX, mouseY);
g.drawString("Mouse at " + movX + ", " + movY, 0, 10);
}

Sample output from this program is shown here:

<table>
<thead>
<tr>
<th>S.NO</th>
<th>RGPV QUESTIONS</th>
<th>YEAR</th>
<th>MARKS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q.1</td>
<td>What is an event in java</td>
<td>Dec 2014</td>
<td>2</td>
</tr>
</tbody>
</table>
Controls are components that allow a user to interact with your application in various ways—for example, a commonly used control is the push button. A layout manager automatically positions components within a container. Thus, the appearance of a window is determined by a combination of the controls that it contains and the layout manager used to position them.

The AWT supports the following types of control.

- Labels
- Push Buttons
- Check Boxes
- Choice Lists
- Lists
- Scroll Bars
- Text Editing

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

```
Component add(Component compObj)
```

Here, `compObj` is an instance of the control that you want to add. A reference to `compObj` is returned. Once a control has been added, it will automatically be visible whenever its parent window is displayed. Sometimes you will want to remove a control from a window when the control is no longer needed. To do this, call `remove()`. This method is also defined by `Container`. It has this general form:

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

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

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

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

The first version creates a blank label. The second version creates a label that contains the string specified by str. This string is left-justified. The third version creates a label that contains the string specified by str using the alignment specified by how. The value of how must be one of these three constants: Label.LEFT, Label.RIGHT, or Label.CENTER. You can set or change the text in a label by using the setText( ) method. You can obtain the current label by calling getText( ). These methods are shown here:

void setText(String str)
String getText( )

For setText( ), str specifies the new label. For getText( ), the current label is returned. You can set the alignment of the string within the label by calling setAlignment( ). To obtain the current alignment, call getAlignment( ). The methods are as follows:

void setAlignment(int how)
int getAlignment( )

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

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

Button( ) throws HeadlessException
Button(String str) throws HeadlessException
The first version creates an empty button. The second creates a button that contains str as a label. After a button has been created, you can set its label by calling `setLabel()` and retrieve its label by calling `getLabel()`. These methods are as follows:

```java
void setLabel(String str)
String getLabel()
```

Here, str becomes the new label for the button.

### Handling Buttons

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

```java
Button( ) throws HeadlessException
Button(String str) throws HeadlessException
```

The first version creates an empty button. The second creates a button that contains str as a label. After a button has been created, you can set its label by calling `setLabel()`. You can retrieve its label by calling `getLabel()`. These methods are as follows:

```java
void setLabel(String str)
String getLabel()
```

Here, str becomes the new label for the button.

```
// Demonstrate Buttons
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="ButtonDemo" width=250 height=150>
</applet>
*/
public class ButtonDemo extends Applet implements ActionListener {
String msg = "";
Button yes, no, maybe;
public void init() {
yes = new Button("Yes");
no = new Button("No");
maybe = new Button("Undecided");
add(yes);
```
add(no);
add(maybe);
yes.addActionListener(this);
no.addActionListener(this);
maybe.addActionListener(this);
}
public void actionPerformed(ActionEvent ae) {
    String str = ae.getActionCommand();
    if(str.equals("Yes")) {
        msg = "You pressed Yes.";
    } else if(str.equals("No")) {
        msg = "You pressed No.";
    } else {
        msg = "You pressed Undecided.";
    }
    repaint();
}
public void paint(Graphics g) {
g.drawString(msg, 6, 100);
}
}

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

// Recognize Button objects.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
we dont take any liability for the notes correctness.
http://www.rgpvonline.com
*
In this version, the program stores each button reference in an array when the buttons are added to the applet window. (Recall that the add( ) method returns a reference to the button when it is added.) Inside actionPerformed( ), this array is then used to determine which button has been pressed.

<table>
<thead>
<tr>
<th>S.NO</th>
<th>RGPV QUESTION</th>
<th>YEAR</th>
<th>MARKS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q.1</td>
<td>What do you mean by AWT controls? What are the various controls supported by AWT?</td>
<td>DEC 2012</td>
<td>8</td>
</tr>
</tbody>
</table>
Applying Checkboxes

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

Checkbox supports these constructors:
- Checkbox() throws HeadlessException
- Checkbox(String str) throws HeadlessException
- Checkbox(String str, boolean on) throws HeadlessException
- Checkbox(String str, boolean on, CheckboxGroup cbGroup) throws HeadlessException
- Checkbox(String str, CheckboxGroup cbGroup, boolean on) throws HeadlessException

The first form creates a checkbox whose label is initially blank. The state of the checkbox is unchecked. The second form creates a checkbox whose label is specified by str. The state of the checkbox is unchecked. The third form allows you to set the initial state of the checkbox. If on is true, the checkbox is initially checked; otherwise, it is cleared. The fourth and fifth forms create a checkbox whose label is specified by str and whose group is specified by cbGroup. If this checkbox is not part of a group, then cbGroup must be null. (Checkbox groups are described in the next section.) The value of on determines the initial state of the checkbox. To retrieve the current state of a checkbox, call getState(). To set its state, call setState(). You can obtain the current label associated with a checkbox by calling getLabel(). To set the label, call setLabel(). These methods are as follows:

```java
boolean getState()
void setState(boolean on)
String getLabel()
void setLabel(String str)
```

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

The following program creates four checkboxes. The initial state of the first box is checked. The status of each checkbox is displayed. Each time you change the state of a checkbox, the status display is updated.

```java
// Demonstrate checkboxes.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
   we dont take any liability for the notes correctness.
http://www.rgpvonline.com
*/
```
<applet code="CheckboxDemo" width=250 height=200>
</applet>

/*
public class CheckboxDemo extends Applet implements ItemListener {
    String msg = "";
    Checkbox winXP, winVista, solaris, mac;
    public void init() {
        winXP = new Checkbox("Windows XP", null, true);
        winVista = new Checkbox("Windows Vista");
        solaris = new Checkbox("Solaris");
        mac = new Checkbox("Mac OS");
        add(winXP);
        add(winVista);
        add(solaris);
        add(mac);
        winXP.addItemListener(this);
        winVista.addItemListener(this);
        solaris.addItemListener(this);
        mac.addItemListener(this);
    }
    public void itemStateChanged(ItemEvent ie) {
        repaint();
    }
    // Display current state of the check boxes.
    public void paint(Graphics g) {
        msg = "Current state: ";
        g.drawString(msg, 6, 80);
        msg = " Windows XP: " + winXP.getState();
        g.drawString(msg, 6, 100);
        msg = " Windows Vista: " + winVista.getState();
        g.drawString(msg, 6, 120);
        msg = " Solaris: " + solaris.getState();
        g.drawString(msg, 6, 140);
        msg = " Mac OS: " + mac.getState();
        g.drawString(msg, 6, 160);
    }
}
*/

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

You can determine which check box in a group is currently selected by calling `getSelectedCheckbox()`. You can set a check box by calling `setSelectedCheckbox()`.

These methods are as follows:
- `Checkbox getSelectedCheckbox()`
- `void setSelectedCheckbox(Checkbox which)`

Here, `which` is the check box that you want to be selected. The previously selected check box will be turned off. Here is a program that uses check boxes that are part of a group:

```java
// Demonstrate check box group.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
 <applet code="CBGroup" width=250 height=200>
 </applet>
 */
public class CBGroup extends Applet implements ItemListener {
```
String msg = "";
Checkbox winXP, winVista, solaris, mac;
CheckboxGroup cbg;
public void init() {
    cbg = new CheckboxGroup();
    winXP = new Checkbox("Windows XP", cbg, true);
    winVista = new Checkbox("Windows Vista", cbg, false);
    solaris = new Checkbox("Solaris", cbg, false);
    mac = new Checkbox("Mac OS", cbg, false);
    add(winXP);
    add(winVista);
    add(solaris);
    add(mac);
    winXP.addItemListener(this);
    winVista.addItemListener(this);
    solaris.addItemListener(this);
    mac.addItemListener(this);
}
public void itemStateChanged(ItemEvent ie) {
    repaint();
}
// Display current state of the check boxes.
public void paint(Graphics g) {
    msg = "Current selection: ";
    msg += cbg.getSelectedCheckbox().getLabel();
g.drawString(msg, 6, 100); }

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

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

To determine which item is currently selected, you may call either getSelectedItem( ) or
getSelectedIndex( ).

These methods are shown here:

String getSelectedItem( )
int getSelectedIndex( )

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

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

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

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

String getItem(int index)

Here, index specifies the index of the desired item.

Handling ChoiceList
Each time a choice is selected, an item event is generated. This is sent to any listeners that
previously registered an interest in receiving item event notifications from that component.
Each listener implements the ItemListener interface. That interface defines the
itemStateChanged( ) method. An ItemEvent object is supplied as the argument to this method.
Here is an example that creates two Choice menus. One selects the operating system. The
other selects the browser.

// Demonstrate Choice lists.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="ChoiceDemo" width=300 height=180>
</applet>
*/
public class ChoiceDemo extends Applet implements ItemListener {
    Choice os, browser;
    String msg = "";
    public void init() {
        os = new Choice();
        browser = new Choice();
        // add items to os list
        os.add("Windows XP");
        os.add("Windows Vista");
        os.add("Solaris");
        os.add("Mac OS");
        // add items to browser list
        browser.add("Internet Explorer");
        browser.add("Firefox");
        browser.add("Opera");
        // add choice lists to window
        add(os);
        add(browser);
        // register to receive item events
        os.addItemListener(this);
        browser.addItemListener(this);
    }
    public void itemStateChanged(ItemEvent ie) {
        repaint();
    }
    // Display current selections.
    public void paint(Graphics g) {
        msg = "Current OS: ";
        msg += os.getSelectedItem();
        g.drawString(msg, 6, 120);
        msg = "Current Browser: ";
        msg += browser.getSelectedItem();
        g.drawString(msg, 6, 140);
    }
}
we dont take any liability for the notes correctness.
Using Lists
The **List** class provides a compact, multiple-choice, scrolling selection list. Unlike the **Choice** object, which shows only the single selected item in the menu, a **List** object can be constructed to show any number of choices in the visible window. It can also be created to allow multiple selections. **List** provides these constructors:

- `List() throws HeadlessException`
- `List(int numRows) throws HeadlessException`
- `List(int numRows, boolean multipleSelect) throws HeadlessException`

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

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

- `void add(String name)`
- `void add(String name, int index)`

Here, `name` is the name of the item added to the list. The first form adds items to the end of the list. The second form adds the item at the index specified by `index`. Indexing begins at zero. You can specify `-1` to add the item to the end of the list. For lists that allow only single selection, you can determine which item is currently selected by calling either `getSelectedItem()` or `getSelectedIndex()`. These methods are shown here:

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

The `getSelectedItem()` method returns a string containing the name of the item. If more than one item is selected, or if no selection has yet been made, `null` is returned.

The `getSelectedIndex()` returns the index of the item. The first item is at index 0. If more than one item is selected, or if no selection has yet been made, `-1` is returned.

For lists that allow multiple selection, you must use either `getSelectedItems()` or `getSelectedIndexes()`, shown here, to determine the current selections:

- `String[] getSelectedItems()`
- `int[] getSelectedIndexes()`
getSelectedItems( ) returns an array containing the names of the currently selected items. 
getSelectedIndexes( ) returns an array containing the indexes of the currently selected items.

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

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

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

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

Here, index specifies the index of the desired item.

**Handling Lists**

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

ggetItemSelectable( ) returns a reference to the object that triggered this event. Here is an 
example that converts the Choice controls in the preceding section into List components, one 
multiple choice and the other single choice:

```java
// Demonstrate Lists.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
 <applet code="ListDemo" width=300 height=180>
 </applet>
 */
public class ListDemo extends Applet implements ActionListener {
 List os, browser;
 String msg = "";
 public void init() {
 os = new List(4, true);
 browser = new List(4, false);
 // add items to os list
 os.add("Windows XP");
```
Sample output generated by the ListDemo applet is shown in Figure given below.
Managing Scroll Bars

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

Scrollbar defines the following constructors:
Scrollbar( ) throws HeadlessException
Scrollbar(int style) throws HeadlessException
Scrollbar(int style, int initialValue, int.thumbSize, int min, int max) throws HeadlessException

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

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

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

The parameters have the same meaning as they have in the third constructor just described. To obtain the current value of the scroll bar, call getValue( ). It returns the current setting. To set the current value, call setValue( ). These methods are as follows:

int getValue()
void setValue(int newValue)

Here, newValue specifies the new value for the scroll bar. When you set a value, the slider box inside the scroll bar will be positioned to reflect the new value. You can also retrieve the minimum and maximum values via getMinimum( ) and getMaximum( ), shown here:

int getMinimum()
int getMaximum()

They return the requested quantity.
By default, 1 is the increment added to or subtracted from the scroll bar each time it is scrolled up or down one line. You can change this increment by calling `setUnitIncrement()`.

By default, page-up and page-down increments are 10. You can change this value by calling `setBlockIncrement()`. These methods are shown here:

```java
void setUnitIncrement(int newIncr)
void setBlockIncrement(int newIncr)
```

**Handling Scroll Bars**

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

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

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

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

public class SBDemo extends Applet
    implements AdjustmentListener, MouseMotionListener {  
String msg = "";
Scrollbar vertSB, horzSB;
public void init() {  
int width = Integer.parseInt(getParameter("width"));
int height = Integer.parseInt(getParameter("height"));
vertSB = new Scrollbar(Scrollbar.VERTICAL, 0, 1, 0, height);
horzSB = new Scrollbar(Scrollbar.HORIZONTAL, 0, 1, 0, width);
```
add(vertSB);
add(horzSB);
// register to receive adjustment events
vertSB.addAdjustmentListener(this);
horzSB.addAdjustmentListener(this);
addMouseMotionListener(this);
}
public void adjustmentValueChanged(AdjustmentEvent ae) {
  repaint();
}
// Update scroll bars to reflect mouse dragging.
public void mouseDragged(MouseEvent me) {
  int x = me.getX();
  int y = me.getY();
  vertSB.setValue(y);
  horzSB.setValue(x);
  repaint();
}
// Necessary for MouseMotionListener
public void mouseMoved(MouseEvent me) {
}
// Display current value of scroll bars.
public void paint(Graphics g) {
  msg = "Vertical: " + vertSB.getValue();
  msg += " , Horizontal: " + horzSB.getValue();
  g.drawString(msg, 6, 160);
  // show current mouse drag position
  g.drawString("*", horzSB.getValue(),
        vertSB.getValue());
}

Sample output from the SBDemo applet is shown in Figure given below.
## Using a TextField

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

- `TextField()` throws `HeadlessException`
- `TextField(int numChars)` throws `HeadlessException`
- `TextField(String str)` throws `HeadlessException`
- `TextField(String str, int numChars)` throws `HeadlessException`

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

`TextField` (and its superclass `TextComponent`) provides several methods that allow you to utilize a text field. To obtain the string currently contained in the text field, call `getText()`.

To set the text, call `setText()`. These methods are as follows:

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

Here, `str` is the new string.

The user can select a portion of the text in a text field. Also, you can select a portion of text under program control by using `select()`. Your program can obtain the currently selected text by calling `getSelectedText()`. These methods are shown here:

```java
String getSelectedText()
void select(int startIndex, int endIndex)
```

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

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

```java
boolean isEditable()
void setEditable(boolean canEdit)
```

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

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

Here, ch specifies the character to be echoed.

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

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

```java
// Demonstrate text field.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="TextFieldDemo" width=380 height=150>
</applet>
*/
public class TextFieldDemo extends Applet
implements ActionListener {
TextField name, pass;
public void init() {
Label namep = new Label("Name: ", Label.RIGHT);
Label passp = new Label("Password: ", Label.RIGHT);
name = new TextField(12);
pass = new TextField(8);
pass.setEchoChar('?');
add(namep);
add(name);
add(passp);
add(pass);
// register to receive action events
name.addActionListener(this);
```
pass.addActionListener(this);

// User pressed Enter.
public void actionPerformed(ActionEvent ae) {
    repaint();
}
public void paint(Graphics g) {
    g.drawString("Name: " + name.getText(), 6, 60);
    g.drawString("Selected text in name: " + name.getSelectedText(), 6, 80);
    g.drawString("Password: " + pass.getText(), 6, 100);
}
}

Sample output from the TextFieldDemo applet is shown in Figure given below.

Using a TextArea

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

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

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

- SCROLLBARS_BOTH
- SCROLLBARS_NONE
- SCROLLBARS_HORIZONTAL_ONLY
- SCROLLBARS_VERTICAL_ONLY

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

**TextArea** adds the following methods:

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

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

Text areas are almost self-contained controls. Your program incurs virtually no management overhead. Text areas only generate got-focus and lost-focus events. Normally, your program simply obtains the current text when it is needed. The following program creates a **TextArea** control:

```java
// Demonstrate TextArea.
import java.awt.*;
import java.applet.*;
/*
<applet code="TextAreaDemo" width=300 height=250>
</applet>
*/
public class TextAreaDemo extends Applet {
    public void init() {
        String val = "Java SE 6 is the latest version of the most
" + "widely-used computer language for Internet programming.
" + "Building on a rich heritage, Java has advanced both
" + "the art and science of computer language design.
" + "One of the reasons for Java's ongoing success is its
" + "constant, steady rate of evolution. Java has never stood
" + "still. Instead, Java has consistently adapted to the
" + "rapidly changing landscape of the networked world.
" + "Moreover, Java has often led the way, charting the
" + "course for others to follow."
;
        TextArea text = new TextArea(val, 10, 30);
```
add(text);
}
}

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

![TextAreaDemo Applet](image)

We don't take any liability for the notes correctness.

http://www.rgpvonline.com
Layout Managers

All of the components that we have shown so far have been positioned by the default layout manager. As we mentioned at the beginning of this chapter, a layout manager automatically arranges your controls within a window by using some type of algorithm. If you have programmed for other GUI environments, such as Windows, then you are accustomed to laying out your controls by hand. While it is possible to lay out Java controls by hand, too, you generally won’t want to, for two main reasons. First, it is very tedious to manually layout a large number of components. Second, sometimes the width and height information is not yet available when you need to arrange some control, because the native toolkit components haven’t been realized. This is a chicken-and-egg situation; it is pretty confusing to figure out when it is okay to use the size of a given component to position it relative to another. Each Container object has a layout manager associated with it. A layout manager is an instance of any class that implements the LayoutManager interface. The layout manager is set by the setLayout() method. If no call to setLayout() is made, then the default layout manager is used. Whenever a container is resized (or sized for the first time), the layout manager is used to position each of the components within it.

The setLayout() method has the following general form:

void setLayout(LayoutManager layoutObj)

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

FlowLayout

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

FlowLayout: [RGPV Dec 201473])
FlowLayout( )
FlowLayout(int how)
FlowLayout(int how, int horz, int vert)

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

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

These values specify left, center, right, leading edge, and trailing edge alignment, respectively. The third constructor allows you to specify the horizontal and vertical space left between components in horz and vert, respectively.

// Use left-aligned flow layout.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="FlowLayoutDemo" width=250 height=200>
</applet>
*/
public class FlowLayoutDemo extends Applet
implements ItemListener {
String msg = "";
Checkbox winXP, winVista, solaris, mac;
public void init() { 
// set left-aligned flow layout
setLayout(new FlowLayout(FlowLayout.LEFT));
winXP = new Checkbox("Windows XP", null, true);
winVista = new Checkbox("Windows Vista");
solaris = new Checkbox("Solaris");
mac = new Checkbox("Mac OS");
add(winXP);
add(winVista);
add(solaris);
add(mac);
// register to receive item events
winXP.addItemListener(this);
winVista.addItemListener(this);
solaris.addItemListener(this);
mac.addItemListener(this);
}
// Repaint when status of a check box changes.
public void itemStateChanged(ItemEvent ie) {
repaint();


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

**Applet Viewer: FlowLa...**

- **Applet**
  - ✅ Windows XP
  - ✅ Windows Vista
  - ✗ Solaris
  - ✗ Mac OS

**Current state:**
- Windows XP: true
- Windows Vista: true
- Solaris: false
- Mac: false

**Applet started.**

**BorderLayout**

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

- `BorderLayout( )`
- `BorderLayout(int horz, int vert)`

The first form creates a default border layout. The second allows you to specify the horizontal and vertical space left between components in `horz` and `vert`, respectively. **BorderLayout**
defines the following constants that specify the regions:

<table>
<thead>
<tr>
<th>BorderLayout.CENTER</th>
<th>BorderLayout.SOUTH</th>
</tr>
</thead>
<tbody>
<tr>
<td>BorderLayout.EAST</td>
<td>BorderLayout.WEST</td>
</tr>
<tr>
<td>BorderLayout.NORTH</td>
<td></td>
</tr>
</tbody>
</table>

// Demonstrate BorderLayout.
import java.awt.*;
import java.applet.*;
import java.util.*;
/*
<applet code="BorderLayoutDemo" width=400 height=200>
</applet>
*/
public class BorderLayoutDemo extends Applet {
public void init() {
setLayout(new BorderLayout());
add(new Button("This is across the top."), BorderLayout.NORTH);
add(new Label("The footer message might go here."), BorderLayout.SOUTH);
add(new Button("Right"), BorderLayout.EAST);
add(new Button("Left"), BorderLayout.WEST);
String msg = "The reasonable man adapts " +
"himself to the world:\n" +
"the unreasonable one persists in " +
"trying to adapt the world to himself.\n" +
"Therefore all progress depends " +
"on the unreasonable man.\n\n" +
" - George Bernard Shaw\n";
add(new TextArea(msg), BorderLayout.CENTER);
}
}

GridLayout[RGPV Dec 2014(7)]
GridLayout lays out components in a two-dimensional grid. When you instantiate a GridLayout, you define the number of rows and columns. The constructors supported by GridLayout are shown here:

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

Here is a sample program that creates a 4\times4 grid and fills it in with 15 buttons, each labeled
with its index:

```java
// Demonstrate GridLayout
import java.awt.*;
import java.applet.*;
/*
<applet code="GridLayoutDemo" width=300 height=200>
</applet>
*/
public class GridLayoutDemo extends Applet {
static final int n = 4;
public void init() {
setLayout(new GridLayout(n, n));
setFont(new Font("SansSerif", Font.BOLD, 24));
for(int i = 0; i < n; i++) {
for(int j = 0; j < n; j++) {
int k = i * n + j;
if(k > 0)
add(new Button("" + k));
}
}
}

Following is the output generated by the \texttt{GridLayoutDemo} applet:
```
The `CardLayout` class is unique among the other layout managers in that it stores several different layouts. Each layout can be thought of as being on a separate index card in a deck that can be shuffled so that any card is on top at a given time. This can be useful for user interfaces with optional components that can be dynamically enabled and disabled upon user input. You can prepare the other layouts and have them hidden, ready to be activated when needed.

**CardLayout** provides these two constructors:

```java
cardLayout( )
cardLayout(int horz, int vert)
```

The first form creates a default card layout. The second form allows you to specify the horizontal and vertical space left between components in `horz` and `vert`, respectively. Use of a card layout requires a bit more work than the other layouts. The cards are typically held in an object of type `Panel`. This panel must have `CardLayout` selected as its layout manager. The cards that form the deck are also typically objects of type `Panel`. Thus, you must create a panel that contains the deck and a panel for each card in the deck. Next, you add to the appropriate panel the components that form each card. You then add these panels to the panel for which `CardLayout` is the layout manager. Finally, you add this panel to the window. Once these steps are complete, you must provide some way for the user to select between cards. One common approach is to include one push button for each card in the deck. When card panels are added to a panel, they are usually given a name. Thus, most of the time, you will use this form of `add( )` when adding cards to a panel:

```java
void add(Component panelObj, Object name)
```

Here, `name` is a string that specifies the name of the card whose panel is specified by `panelObj`. After you have created a deck, your program activates a card by calling one of the following
methods defined by CardLayout:

void first(Container deck)
void last(Container deck)
void next(Container deck)
void previous(Container deck)
void show(Container deck, String cardName)

Here, deck is a reference to the container (usually a panel) that holds the cards, and cardName is the name of a card. Calling first( ) causes the first card in the deck to be shown. To show the last card, call last( ). To show the next card, call next( ). To show the previous card, call previous( ). Both next( ) and previous( ) automatically cycle back to the top or bottom of the deck, respectively. The show( ) method displays the card whose name is passed in cardName.

// Demonstrate CardLayout.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="CardLayoutDemo" width=300 height=100>
</applet>
*/
public class CardLayoutDemo extends Applet
implements ActionListener, MouseListener {
    Checkbox winXP, winVista, solaris, mac;
    Panel osCards;
    CardLayout cardLO;
    Button Win, Other;
    public void init() {
        Win = new Button("Windows");
        Other = new Button("Other");
        add(Win);
        add(Other);
        cardLO = new CardLayout();
        osCards = new Panel();
        osCards.setLayout(cardLO); // set panel layout to card layout
        winXP = new Checkbox("Windows XP", null, true);
        winVista = new Checkbox("Windows Vista");
        solaris = new Checkbox("Solaris");
        mac = new Checkbox("Mac OS");
        // add Windows check boxes to a panel
        Panel winPan = new Panel();
        winPan.add(winXP);
        winPan.add(winVista);
// add other OS check boxes to a panel
Panel otherPan = new Panel();
otherPan.add(solaris);
otherPan.add(mac);
// add panels to card deck panel
osCards.add(winPan, "Windows");
osCards.add(otherPan, "Other");
// add cards to main applet panel
add(osCards);
// register to receive action events
Win.addActionListener(this);
Other.addActionListener(this);
// register mouse events
addMouseListener(this);

// Cycle through panels.
public void mousePressed(MouseEvent me) {
    cardLO.next(osCards);
}

// Provide empty implementations for the other MouseListener methods.
public void mouseClicked(MouseEvent me) {
}
public void mouseEntered(MouseEvent me) {
}
public void mouseExited(MouseEvent me) {
}
public void mouseReleased(MouseEvent me) {
}

public void actionPerformed(ActionEvent ae) {
    if(ae.getSource() == Win) {
        cardLO.show(osCards, "Windows");
    } else {
        cardLO.show(osCards, "Other");
    }
}
Q.1. What are the different types of Layout in Java?

DEC 2012 8

Q-2. Explain the Layout Manager in Java. Also, describe the concept of menus.

June-2011 8

Q-3. How many Layouts are available in AWT Package?

Dec-2009 10
The Delegation Event Model

The modern approach to handling events is based on the delegation event model, which defines standard and consistent mechanisms to generate and process events. Its concept is quite simple: a source generates an event and sends it to one or more listeners. In this scheme, the listener simply waits until it receives an event. Once an event is received, the listener processes the event and then returns. The advantage of this design is that the application logic that processes events is cleanly separated from the user interface logic that generates those events.

A user interface element is able to “delegate” the processing of an event to a separate piece of code. In the delegation event model, listeners must register with a source in order to receive an event notification. This provides an important benefit: notifications are sent only to listeners that want to receive them. This is a more efficient way to handle events than the design used by the old Java 1.0 approach. Previously, an event was propagated up the containment hierarchy until it was handled by a component. This required components to receive events that they did not process, and it wasted valuable time. The delegation event model eliminates this overhead.

Events

In the delegation model, an event is an object that describes a state change in a source. It can be generated as a consequence of a person interacting with the elements in a graphical user interface. Some of the activities that cause events to be generated are pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse. Many other user operations could also be cited as examples. Events may also occur that are not directly caused by interactions with a user interface. For example, an event may be generated when a timer expires, a counter exceeds a value, a software or hardware failure occurs, or an operation is completed. You are free to define events that are appropriate for your application.

Event Sources

A source is an object that generates an event. This occurs when the internal state of that object changes in some way. Sources may generate more than one type of event.

A source must register listeners in order for the listeners to receive notifications about a specific type of event. Each type of event has its own registration method. Here is the general form:

public void addTypeListener(TypeListener el)

Here, Type is the name of the event, and el is a reference to the event listener. For example, the method that registers a keyboard event listener is called addKeyListener(). The method that
registers a mouse motion listener is called `addMouseMotionListener()`. When an event occurs, all registered listeners are notified and receive a copy of the event object. This is known as *multicasting* the event. In all cases, notifications are sent only to listeners that register to receive them.

**Event Listeners**

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

The methods that receive and process events are defined in a set of interfaces found in `java.awt.event`. For example, the `MouseMotionListener` interface defines two methods to receive notifications when the mouse is dragged or moved. Any object may receive and process one or both of these events if it provides an implementation of this interface.

**Event Classes**

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

```
EventObject(Object src)
```

Here, `src` is the object that generates this event. `EventObject` contains two methods: `getSource()` and `toString()`.

The `getSource()` method returns the source of the event. Its general form is shown here:

```
Object getSource()
```

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

The class `AWTEvent`, defined within the `java.awt` package, is a subclass of `EventObject`. It is the superclass (either directly or indirectly) of all AWT-based events used by the delegation event model. Its `getID()` method can be used to determine the type of the event. The signature of this method is shown here:

```
int getID()
```

Additional details about `AWTEvent` are provided at the end of Chapter 24. At this point, it is important to know only that all of the other classes discussed in this section are subclasses of `AWTEvent`.

To summarize:

- `EventObject` is a superclass of all events.
- `AWTEvent` is a superclass of all AWT events that are handled by the delegation event model.

The package `java.awt.event` defines many types of events that are generated by various user
interface elements.

**Action Event Class**

An **ActionEvent** is generated when a button is pressed, a list item is double-clicked, or a menu item is selected. The **ActionEvent** class defines four integer constants that can be used to identify any modifiers associated with an action event: **ALT_MASK**, **CTRL_MASK**, **META_MASK**, and **SHIFT_MASK**. In addition, there is an integer constant, **ACTION_PERFORMED**, which can be used to identify action events.

**ActionEvent** has these three constructors:

```
ActionEvent(Object src, int type, String cmd)
ActionEvent(Object src, int type, String cmd, int modifiers)
ActionEvent(Object src, int type, String cmd, long when, int modifiers)
```

Here, `src` is a reference to the object that generated this event. The type of the event is specified by `type`, and its command string is `cmd`. The argument `modifiers` indicates which modifier keys (ALT, CTRL, META, and/or SHIFT) were pressed when the event was generated. The `when` parameter specifies when the event occurred.

You can obtain the command name for the invoking **ActionEvent** object by using the `getActionCommand()` method, shown here:

```
String getActionCommand()
```

For example, when a button is pressed, an action event is generated that has a command name equal to the label on that button.

The `getModifiers()` method returns a value that indicates which modifier keys (ALT, CTRL, META, and/or SHIFT) were pressed when the event was generated. Its form is shown here:

```
int getModifiers()
```

**The AdjustmentEvent Class**

An **AdjustmentEvent** is generated by a scroll bar. There are five types of adjustment events. The **AdjustmentEvent** class defines integer constants that can be used to identify them. The constants and their meanings are shown here:
In addition, there is an integer constant, `ADJUSTMENT_VALUE_CHANGED`, that indicates that a change has occurred.

Here is one `AdjustmentEvent` constructor:

```java
AdjustmentEvent(Adjustable src, int id, int type, int data)
```

Here, `src` is a reference to the object that generated this event. The `id` specifies the event. The type of the adjustment is specified by `type`, and its associated data is `data`. The `getAdjustable()` method returns the object that generated the event. Its form is shown here:

```java
Adjustable getAdjustable()
```

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

```java
int getAdjustmentType()
```

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

```java
int getValue()
```

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

---

**The ContainerEvent Class**

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

```java
ContainerEvent(Component src, int type, Component comp)
```

Here, `src` is a reference to the container that generated this event. The type of the event is specified by `type`, and the component that has been added to or removed from the container is
You can obtain a reference to the container that generated this event by using the `getContainer()` method, shown here:

```
Container getContainer()
```

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

```
Component getChild()
```

### The FocusEvent Class

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

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

- `FocusEvent(Component src, int type)`
- `FocusEvent(Component src, int type, boolean temporaryFlag)`
- `FocusEvent(Component src, int type, boolean temporaryFlag, Component other)`

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

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

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

```
Component getOppositeComponent()
```

The opposite component is returned.

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

```
boolean isTemporary()
```

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

### The InputEvent Class

The abstract class `InputEvent` is a subclass of `ComponentEvent` and is the superclass for
component input events. Its subclasses are \texttt{KeyEvent} and \texttt{MouseEvent}.

\texttt{InputEvent} defines several integer constants that represent any modifiers, such as the control key being pressed, that might be associated with the event. Originally, the \texttt{InputEvent} class defined the following eight values to represent the modifiers:

\begin{verbatim}
ALT_MASK BUTTON2_MASK META_MASK
ALT_GRAPH_MASK BUTTON3_MASK SHIFT_MASK
BUTTON1_MASK CTRL_MASK
\end{verbatim}

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

\begin{verbatim}
ALT_DOWN_MASK BUTTON2_DOWN_MASK META_DOWN_MASK
ALT_GRAPH_DOWN_MASK BUTTON3_DOWN_MASK SHIFT_DOWN_MASK
BUTTON1_DOWN_MASK CTRL_DOWN_MASK
\end{verbatim}

When writing new code, it is recommended that you use the new, extended modifiers rather than the original modifiers. To test if a modifier was pressed at the time an event is generated, use the \texttt{isAltDown( )}, \texttt{isAltGraphDown( )}, \texttt{isControlDown( )}, \texttt{isMetaDown( )}, and \texttt{isShiftDown( )} methods. The forms of these methods are shown here:

\begin{verbatim}
boolean isAltDown( )
boolean isAltGraphDown( )
boolean isControlDown( )
boolean isMetaDown( )
boolean isShiftDown( )
\end{verbatim}

You can obtain a value that contains all of the original modifier flags by calling the \texttt{getModifiers( )} method. It is shown here:

\begin{verbatim}
int getModifiers( )
\end{verbatim}

You can obtain the extended modifiers by calling \texttt{getModifiersEx( )}, which is shown here:

\begin{verbatim}
int getModifiersEx( )
\end{verbatim}

\textbf{The ItemEvent Class}

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

\begin{verbatim}
DESELECTED The user deselected an item.
SELECTED The user selected an item.
\end{verbatim}
In addition, ItemEvent defines one integer constant, ITEM_STATE_CHANGED, that signifies a change of state.

ItemEvent has this constructor:
ItemEvent(ItemSelectable src, int type, Object entry, int state)

Here, src is a reference to the component that generated this event. For example, this might be a list or choice element. The type of the event is specified by type. The specific item that generated the item event is passed in entry. The current state of that item is in state. The getItem( ) method can be used to obtain a reference to the item that generated an event. Its signature is shown here:
Object getItem( )

The getItemSelectable( ) method can be used to obtain a reference to the ItemSelectable object that generated an event. Its general form is shown here:
ItemSelectable getItemSelectable( )
Lists and choices are examples of user interface elements that implement the ItemSelectable interface.

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

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

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

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

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

KeyEvent is a subclass of InputEvent. Here is one of its constructors:
KeyEvent(Component src, int type, long when, int modifiers, int code, char ch)
Here, src is a reference to the component that generated the event. The type of the event is specified by type. The system time at which the key was pressed is passed in when. The modifiers argument indicates which modifiers were pressed when this key event occurred. The virtual key code, such as VK_UP, VK_A, and so forth, is passed in code. The character equivalent (if one exists) is passed in ch. If no valid character exists, then ch contains CHAR_UNDEFINED.

For KEY_TYPED events, code will contain VK_UNDEFINE.
The KeyEvent class defines several methods, but the most commonly used ones are getKeyChar(), which returns the character that was entered, and getKeyCode(), which returns the key code. Their general forms are shown here:
char getKeyChar()
int getKeyCode()

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

The MouseEvent Class
There are eight types of mouse events. The MouseEvent class defines the following integer constants that can be used to identify them:
MOUSE_CLICKED The user clicked the mouse.
MOUSE_DRAGGED The user dragged the mouse.
MOUSE_ENTERED The mouse entered a component.
MOUSE_EXITED The mouse exited from a component.
MOUSE_MOVED The mouse moved.
MOUSE_PRESSED The mouse was pressed.
MOUSERELEASED The mouse was released.
MOUSE_WHEEL The mouse wheel was moved.

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

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

Here, src is a reference to the component that generated the event. The type of the event is specified by type. The system time at which the mouse event occurred is passed in when. The modifiers argument indicates which modifiers were pressed when a mouse event occurred. The coordinates of the mouse are passed in x and y. The click count is passed in clicks. The triggersPopup flag indicates if this event causes a pop-up menu to appear on this platform. Two commonly used methods in this class are getX() and getY(). These return the X and Y coordinates of the mouse within the component when the event occurred. Their forms are
shown here:

int getX( )
int getY( )

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

Point getPoint( )
It returns a `Point` object that contains the X,Y coordinates in its integer members: x and y. The `translatePoint( )` method changes the location of the event. Its form is shown here:

void translatePoint(int x, int y)
Here, the arguments x and y are added to the coordinates of the event.
The `getClickCount( )` method obtains the number of mouse clicks for this event. Its signature is shown here:
int getClickCount( )
The `isPopupTrigger( )` method tests if this event causes a pop-up menu to appear on this platform. Its form is shown here:
boolean isPopupTrigger( )

Also available is the `getButton( )` method, shown here:
int getButton( )
It returns a value that represents the button that caused the event. The return value will be one of these constants defined by `MouseEvent`:

NOBUTTON BUTTON1 BUTTON2 BUTTON3

The `NOBUTTON` value indicates that no button was pressed or released.
Java SE 6 added three methods to `MouseEvent` that obtain the coordinates of the mouse relative to the screen rather than the component. They are shown here:

Point getLocationOnScreen( )
int getXOnScreen( )
int getYOnScreen( )

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

The `MouseWheelEvent` Class

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

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

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

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

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

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

```java
int getWheelRotation()
```

It returns the number of rotational units. If the value is positive, the wheel moved counterclockwise. If the value is negative, the wheel moved clockwise.

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

```java
int getScrollType()
```

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

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

```java
int getScrollAmount()
```

### The TextEvent Class

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

The one constructor for this class is shown here:
TextEvent(Object src, int type)

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

The TextEvent object does not include the characters currently in the text component that generated the event. Instead, your program must use other methods associated with the text component to retrieve that information. This operation differs from other event objects discussed in this section. For this reason, no methods are discussed here for the TextEvent class. Think of a text event notification as a signal to a listener that it should retrieve information from a specific text component.

The WindowEvent Class

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

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

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

WindowEvent(Window src, int type)

Here, src is a reference to the object that generated this event. The type of the event is type. The next three constructors offer more detailed control:

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

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

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

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

<table>
<thead>
<tr>
<th>S.NO</th>
<th>RGPV QUESTION</th>
<th>YEAR</th>
<th>MARKS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q.1</td>
<td>What are the major components of event delegation model? Discuss with suitable example.</td>
<td>DEC 2014</td>
<td>7</td>
</tr>
<tr>
<td>Q.2</td>
<td>Explain the following with the help of small program. a) Form Layout b). Flow Layout c). Grid Layout</td>
<td>DEC 2014</td>
<td>7</td>
</tr>
<tr>
<td>Q.4</td>
<td>What is Event? Explain different components of an Event</td>
<td>DEC 2010</td>
<td>8</td>
</tr>
</tbody>
</table>

**References:**

<table>
<thead>
<tr>
<th>Book</th>
<th>Author</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java Programming</td>
<td>Herbertt Schield</td>
<td>1</td>
</tr>
<tr>
<td>Java</td>
<td>E Balaguruswamy</td>
<td>2</td>
</tr>
<tr>
<td>Java</td>
<td>Khalid Mugal</td>
<td>3</td>
</tr>
</tbody>
</table>