Unit-V

APPLETS

Applet

- Definition : Applets are small applications that are accessed on an Internet server, transported over the Internet, automatically installed, and run as part of a Web document.

Applet Example

```
import java.awt.*;
import
java.applet.*;
public class SimpleApplet extends Applet
{
public void paint(Graphics g)
{
g.drawString("A Simple Applet", 20, 20);
}
}
```

- An Applet begins with two import statements.

1The first imports the Abstract Window Toolkit (AWT) classes.

  - Applets interact with the user through the AWT, not through the consolebased I/O classes.
  - The AWT contains support for a windowbased, graphical interface.

2 The second import statement imports the applet package, which contains the class Applet. Every applet that you create must be a subclass of Applet.

After Import Stmts……

Class :

- The class must be declared as public, because it will be accessed by code that is outside the program.

Paint()

- Inside every class a paint() method is defined by the AWT and this must be overridden by the applet.
- paint( ) is called each time that the applet must redisplay its output.
- paint() is also called when the applet begins execution.
- Whatever the cause, whenever the applet must redraw its output, paint( ) is called.
- The paint( ) method has one parameter of type Graphics.
- This parameter contains the graphics context, which describes the graphics environment in which the applet is running.
- This context is used whenever output to the applet is required. drawString() :
- Inside paint() there is a call to drawString( ), which is a member of the Graphics class.
- This method outputs a string beginning at the specified X,Y location.
- It has the following general form:

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

- Here, message is the string to be output beginning at x,y. In a Java window, the upperleft corner is location 0,0.

Note

- Applet does not have a main( ) method.
- Unlike Java programs, applets do not begin execution at main( ).
- In fact, most applets don't even have a main( ) method.
- Instead, an applet begins execution when the name of its class is passed to an applet viewer or to a network browser.

Running an Applet Program

- Running SimpleApplet involves a different process.
- In fact, there are two ways in which you can run an applet:

■ Executing the applet within a Javacompatible Web browser.

■ Using an applet viewer, such as the standard SDK tool, appletviewer.

An applet viewer executes your applet in a window. This is generally the fastest and easiest way to test your applet.

HTML code

- To execute an applet in a Web browser, you need to write a short HTML text file that contains the appropriate APPLET tag.
- SampleApplet:

<applet code="SampleApplet" width=200 height=60> </applet>

• The width and height statements specify the dimensions of the display area used by the applet.

A Simple Applet Program

```
import java.awt.*;
import
java.applet.*;
/* <applet code="SimpleApplet" width=200 height=60>
</applet> */ public class SimpleApplet extends Applet
{

public void paint(Graphics g)
{
g.drawString("A Simple Applet", 20, 20);
}
}
```

Applet Architecture

- An applet is a windowbased program. As such, its architecture is different from the socalled normal, consolebased programs.

1First, applets are event driven.

- An applet waits until an event occurs. The AWT notifies the applet about an event by calling an event handler that has been provided by the applet.
- Once this happens, the applet must take appropriate action and then quickly return control to the AWT.
- This is a crucial point.

2 Second, the user initiates interaction with an applet—not the other way around.

- As you know, in a nonwindowed program, when the program needs input, it will prompt the user and then call some input method, such as readLine( ).
- This is not the way it works in an applet. Instead, the user interacts with the applet as he or she wants, when he or she wants.

- These interactions are sent to the applet as events to which the applet must respond.

An Applet Skeleton

- Applets override a set of methods that provides the basic mechanism by which the browser or applet viewer interfaces to the applet and controls its execution.

Applet Initialization and Termination

- It is important to understand the order in which the various methods shown in the skeleton are called.
- When an applet begins, the AWT calls the following methods, in this sequence:
    1. init( )
    2. start( )
    3. paint( )
- When an applet is terminated, the following sequence of method calls takes place:
    1. stop( )
    2. destroy( )

init( )

- The init( ) method is the first method to be called.
- This is where you should initialize variables.
- This method is called only once during the run time of your applet. start( )
- The start( ) method is called after init( ). It is also called to restart an applet after it has been stopped.
- Whereas init( ) is called once—the first time an applet is loaded,start( ) is called each time an applet's HTML document is displayed onscreen.
- So, if a user leaves a web page and comes back, the applet resumes execution at start( ). paint( )
- The paint( ) method is called each time your applet's output must be redrawn.
- This situation can occur for several reasons. The paint( ) method has one parameter of type Graphics.
- This parameter will contain the graphics context, which describes the graphics environment in which the applet is running.
- This context is used whenever output to the applet is required.

stop( )

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

destroy( )

- The destroy( ) method is called when the environment determines that your applet needs to be removed completely from memory.
- At this point, you should free up any resources the applet may be using.
- The stop( ) method is always called before destroy( ).

Program 1 (Applet Skeleton)

```
import java.awt.*;
```

```
import java.applet.*;
/*<applet code="AppletSkel" width=300 height=100>
</applet> */ public class AppletSkel extends Applet
{
// Called first.
public void
init()
{
// initialization
}
/* Called second, after init(). Also called whenever the applet is restarted.
*/ public void start()
{
// start or resume execution

  }
  // Called when the applet is
  stopped. public void stop()
  {
  // suspends execution
  }
  /* Called when applet is terminated. This is the last method executed.
  */ public void destroy() {
  // perform shutdown activities
  }
  // Called when an applet's window must be
  restored. public void paint(Graphics g) {
  // redisplay contents of window
  }
  }
```

Simple Applet Display Methods

- To set the background color of an applet's window, use setBackground( ).
- To set the foreground color, use setForeground( ).
- These methods have the following general forms:

```
void setBackground(Color
newColor) void
setForeground(Color newColor)
```

- *newColor specifies the new color.*
- The class Color defines the constants shown here that can be used to specify colors:

Color.black	Color.magenta
Color.blue	Color.orange
Color.cyan	Color.pink

Color.darkGray	Color.red
Color.gray	Color.white
Color.green	Color.yellow
Color.lightGray	

Examples:

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

Repainting

- As a general rule, an applet writes to its window only when its update( ) or paint( ) method is called by the AWT.
- But, how can the applet itself cause its window to be updated when its information changes?
- It cannot create a loop inside paint( ) that repeatedly scrolls

- Solution is : whenever your applet needs to update the information displayed in its window, it simply calls repaint( ).
- The repaint( ) method is defined by the AWT.
- It causes the AWT runtime system to execute a call to your applet's update( ) method, which, in its default implementation, calls paint( ).
- Thus, for another part of your applet to output to its window, simply store the output and then call repaint( ).
- The repaint( ) method has four forms.

1 void repaint( )

- This version causes the entire window to be repainted.

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

- This version specifies a region that will be repainted
- Here, the coordinates of the upperleft corner of the region are specified by left and top, and the width and height of the region are passed in width and height.
- These dimensions are specified in pixels. You save time by specifying a region to repaint.
- Calling repaint( ) is essentially a request that your applet be repainted sometime soon.
- However, if your system is slow or busy, update( ) might not be called immediately.
- This can be a problem in many situations, including animation, in which a consistent update time is necessary.
- One solution to this problem is to use the following forms of repaint( ):

3 void repaint(long maxDelay)

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

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

Program 2

```
import
java.awt.*;
import
java.applet.*; /*
<applet code="SimpleBanner" width=300
height=50> </applet>
*/
public class SimpleBanner extends Applet implements Runnable
{ String msg = " A Simple Moving Banner.";
Thread t =
null; int state;
boolean stopFlag;
// Set colors and initialize
thread. public void init() {
setBackground(Color.cyan);
setForeground(Color.red);
}
// Start thread
public void start() {
t = new
Thread(this);
stopFlag = fals

t.start();
}
// Entry point for the thread that runs the banner.
public void run() {
char ch;
// Display banner
for( ; ; )
{ try {
repaint()
;
Thread.sleep(250);
ch =
msg.charAt(0);
msg = msg.substring(1,
msg.length()); msg += ch;
if(stopFlag)
break;
} catch(InterruptedException e) {}
}
}
// Pause the banner.
public void stop() {
stopFlag = true;
t = null;
}
// Display the banner.
```

```
public void paint(Graphics g)
{ g.drawString(msg, 50, 30);
}
}
```

Status Window

- In addition to displaying information in its window, an applet can also output a message to the status window of the browser or applet viewer on which it is running.
- To do so, call showStatus( ) with the string that you want displayed.
- The status window is a good place to give the user feedback about what is occurring in the applet, suggest options, or possibly report some types of errors. Program 3

```
import java.awt.*;
import
java.applet.*; /*
<applet code="StatusWindow" width=300
height=50> </applet>
*/
public class StatusWindow extends
Applet{ public void init() {
setBackground(Color.cyan);
}
// Display msg in applet
window. public void
paint(Graphics g) {
g.drawString("This is in the applet window.", 10, 20);

showStatus("This is shown in the status window.");
}
}
```

Passing Parameters to Applets

- APPLET tag in HTML allows you to pass parameters to your applet.
- To retrieve a parameter, use the getParameter() method.

Program 4

```
import
java.awt.*;
import
java.applet.*; /*
<applet code="ParamDemo" width=300
height=80> <param name=fontName
value=Courier>
<param name=fontSize
value=14> <param
name=leading value=2>
<param name=accountEnabled
value=true> </applet>
*/
public class ParamDemo extends
Applet{ String fontName;
```

```
int fontSize;
float leading;
boolean
active;
// Initialize the string to be
displayed. public void start() {
String param;
fontName =
getParameter("fontName");
if(fontName == null)
fontName = "Not Found";
param =
getParameter("fontSize"); try {
if(param != null) // if not found
fontSize =
Integer.parseInt(param); else
fontSize = 0;
} catch(NumberFormatException e)
{ fontSize = 1;
}
param =
getParameter("leading"); try {
if(param != null) // if not found
leading =
Float.valueOf(param).floatValue(); else
leading = 0;
} catch(NumberFormatException e)
{ leading = 1;
}


param =
getParameter("accountEnabled");
if(param != null)
active = Boolean.valueOf(param).booleanValue();
}
// Display parameters.
public void paint(Graphics g) {
g.drawString("Font name: " + fontName, 0,
10); g.drawString("Font size: " + fontSize, 0,
26); g.drawString("Leading: " + leading, 0,
42); g.drawString("Account Active: " +
active, 0, 58);
}
}
```

SWINGS

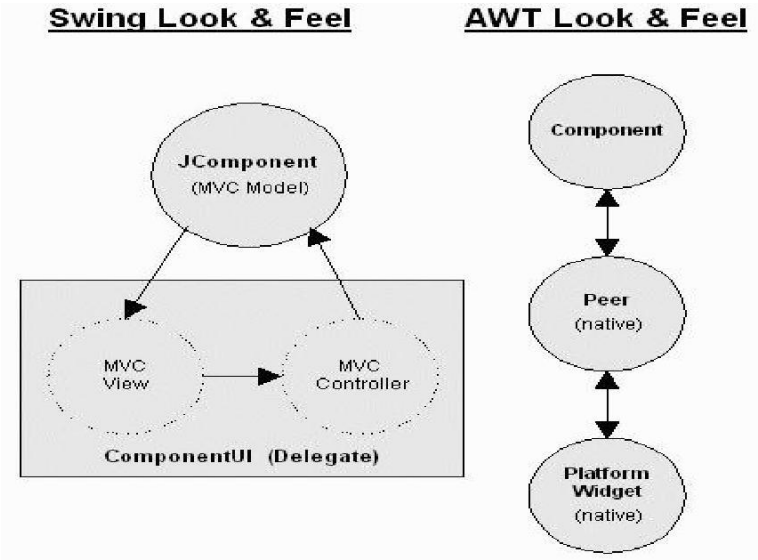
- Swing is a set of classes that provides more powerful and flexible components than are possible with the AWT.
- In addition to the familiar components, such as buttons, check boxes, and labels, Swing supplies several exciting additions, including tabbed panes, scroll panes, trees, and tables.
- Even familiar components such as buttons have more capabilities in Swing.
- For example, a button may have both an image and a text string associated with it. Also, the image can be changed as the state of the button changes.
- Unlike AWT components, Swing components are not implemented by platform-specific code.
- Instead, they are written entirely in Java and, therefore, are platform-independent.
- The term lightweight is used to describe such elements.

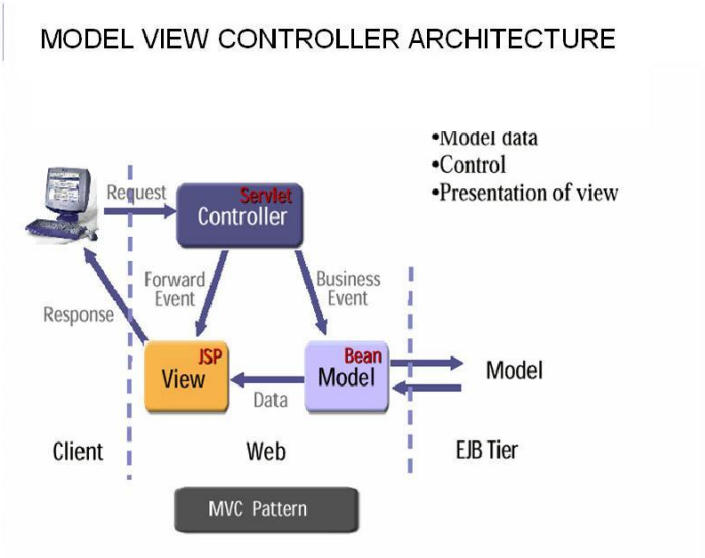
The Swing component are defined in javax.swing

1. AbstractButton: Abstract superclass for Swing buttons.
2. ButtonGroup: Encapsulates a mutually exclusive set of buttons.
3. ImageIcon: Encapsulates an icon.
4. JApplet: The Swing version of Applet.
5. JButton: The Swing push button class.
6. JCheckBox: The Swing check box class.
7. JComboBox : Encapsulates a combo box (an combination of a drop-down list and text field).
8. JLabel: The Swing version of a label.
9. JRadioButton: The Swing version of a radio button.
10. JScrollPane: Encapsulates a scrollable window.
11. JTabbedPane: Encapsulates a tabbed window.
12. JTable: Encapsulates a table-based control.
13. JTextField: The Swing version of a text field.
14. JTree: Encapsulates a tree-based control.

Limitations of AWT

- AWT supports limited number of GUI components.
- AWT components are heavy weight components.
- AWT components are developed by using platform specific code.
- AWT components behaves differently in different operating systems.
- AWT component is converted by the native code of the operating system.
- Lowest Common Denominator
    - If not available natively on one Java platform, not available on any Java platform
- Simple Component Set
- Components Peer-Based
    - Platform controls component appearance
    - Inconsistencies in implementations
        - Interfacing to native platform error-prone





Model

- Model consists of data and the functions that operate on data
- Java bean that we use to store data is a model component
- EJB can also be used as a model component

View

- View is the front end that user interact.
- View can be a
  HTML
  JSP
  Struts ActionForm

Controller

- Controller component responsibilities
  1. Receive request fro m client
  2. Map request to spec ific business operation
  3. Determine the view to display based on the result of the business operation

Model- Delegate Architecture

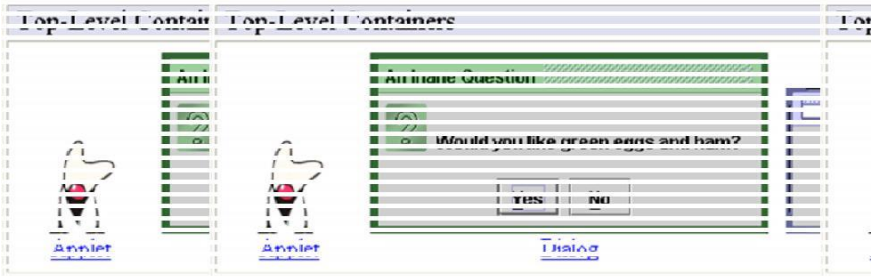
- Although the MVC architec ture and the principles behind it are conceptually sound, the high separation between the view and the controller was not benificial for Swing c omponents.
- Instead Swing uses a modifi ed version of MVC that combines the view and controller into a single logical entity called the "UI delegate" also called as "Model-Delegate Archit ecture"

Components

- Container
  - JComponent
    - AbstractBut ton
      - JButton
      - JMe nuItem
        - » JCheckBoxMenuItem
        - » JMenu
        - » JRadioButtonMenuItem
      - JTog gleButton
        - » JCheckBox
        - » JRadioButton
- JComponent
  - JTextComponent
    - JTextArea
    - JTextField
      - JPas swordField
    - JTextPane
      - JHTMLPane

Containers

- Top-Level Containers
- The components at the top of any Swing containment hierarchy

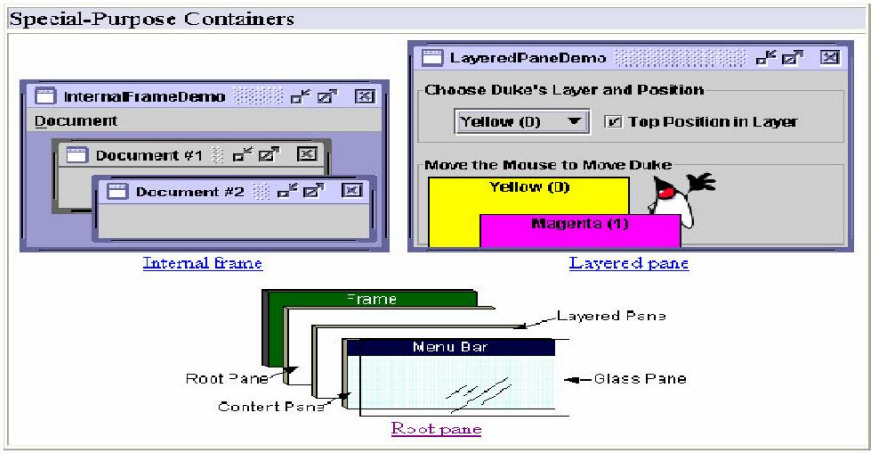


General Purpose Containers

- Intermediate containers that can be used under many different circumstances.

Special Purpose Container

- Intermediate containers that play specific roles in the UI.



LayoutManager

- Every 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)

Java has several predefined LayoutManager classes.

FlowLayout

- FlowLayout is the default layout manager.
- FlowLayout implements a simple layout style, which is similar to how words flow in a text editor.
- Components are laid out from the upper-left corner, left to right and top to bottom.
- When no more components fit on a line, the next one appears on the next line.
- A small space is left between each component, above and below, as well as left and right.

Here are the constructors for FlowLayout:

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

- The first form creates the default layout, which centers components and leaves five pixels of space between each component.
- The second form lets you specify how each line is aligned.

Valid values for *how are as follows:*

FlowLayout.LEFT
FlowLayout.CENTER
FlowLayout.RIGHT

These values specify left, center, and right alignment, respectively.

2 The third form allows you to specify the horizontal and vertical space left between components in horz and vert,
respectively. Program 1

```
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 Win98, winNT, solaris, mac;
public void init() {
// set left-aligned flow layout
setLayout(new FlowLayout(FlowLayout.LEFT));
Win98 = new Checkbox("Windows 98/XP", null,
true); winNT = new Checkbox("Windows NT/2000");
solaris = new Checkbox("Solaris");
mac = new Checkbox("MacOS");
add(Win98);
add(winNT);
add(solaris);
add(mac);
// register to receive item events
Win98.addItemListener(this);
winNT.addItemListener(this);
```

```
solaris.addItemListener(this);
mac.addItemListener(this);
}
// Repaint when status of a check box changes. public void itemStateChanged(ItemEvent ie) { repaint();
}
// Display current state of the check boxes. public void paint(Graphics g) {
msg = "Current state: ";
g.drawString(msg, 6, 80);
msg = " Windows 98/XP: " + Win98.getState();
g.drawString(msg, 6, 100);
msg = " Windows NT/2000: " + winNT.getState();
g.drawString(msg, 6, 120);
msg = " Solaris: " + solaris.getState();
g.drawString(msg, 6, 140);
msg = " Mac: " + mac.getState();
g.drawString(msg, 6, 160);
}
}
```

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.
- The constructors defined by BorderLayout are:

```
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 horizontal and vertical, respectively.

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

Program 2 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 = "Hii  " +
"this is JP Lab;\n" +
"And you are now doing Swing Programs "
+ "This is your final week.\n" +
"Prepare well for Lab Exam " +
"and for your JP External Exam.\n\n"
+ " – Have a nice Day..Byeee\n\n";
add(new TextArea(msg), BorderLayout.CENTER);
}
}
```

GridLayout

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

  GridLayout(int numRows, int numColumns) GridLayout(int numRows, int numColumns, int horz, int vert)
- The first form creates a single-column grid layout.
- The second form creates a grid layout with the specified number of rows and columns.
- The third form allows you to specify the horizontal and vertical space left between components in horz and vert,respectively.

Program 3 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));
}
}
}
}
```

CardLayout

- The CardLayout class is unique among the other layout managers in that it stores several different layouts.
- This can be useful for user interfaces with optional components that can be dynamically enabled and disabled upon user input.
- You can prepare the other layouts and have them hidden, ready to be activated when needed. CardLayout provides these two constructors:

CardLayout( ) CardLayout(int horz, int vert)

// The first form creates a default card layout.

// The second form allows you to specify the horizontal and vertical space left between components in horz and vert, respectively.

Program 4 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 Win98, winNT, 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 Win98 = new Checkbox("Windows 98/XP", null,
true); winNT = new Checkbox("Windows NT/2000");
solaris = new Checkbox("Solaris");
mac = new Checkbox("MacOS");
• add Windows check boxes to a
panel Panel winPan = new Panel();
winPan.add(Win98);
winPan.add(winNT);
• 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");
}
}
}
```

Event Handling

"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 received, the listener processes the event and then returns.
- 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.

Events

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

- 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.
- 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.

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.
- Its general form is:

  public void addTypeListener(TypeListener *el*)
- Here, Type is the name of the event and *el* is a reference to the event listener.

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.

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

Event Class	Description
ActionEvent	Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
AdjustmentEvent	Generated when a scroll bar is manipulated.
ComponentEvent	Generated when a component is hidden, moved, resized, or becomes visible.
ContainerEvent	Generated when a component is added to or removed from a container.
FocusEvent	Generated when a component gains or loses keyboard focus.
InputEvent	Abstract super class for all component input event classes.
ItemEvent	Generated when a check box or list is clicked; also occurs when a choice selection is made or a checkable menu is selected or deselected.
KeyEvent	Generated when input is received from the keyboard.
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.
MouseWheelEvent	Generated when the mouse wheel is moved. (Added by Java 2, version 1.4)
TextEvent	Generated when the value of a text area or text field is changed.
WindowEvent	Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Sources of Events

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

Event Listener Interfaces

- The delegation event model has two parts: sources and listeners.
- Listeners are created by implementing one or more of the interfaces defined by the java.awt.event package.
- When an event occurs, the event source invokes the appropriate method defined by the listener and provides an event object as its argument.

Interface	Description
ActionListener	Defines one method to receive action events.
AdjustmentListener	Defines one method to receive adjustment events.
ComponentListener	Defines two methods to recognize when a component is hidden, moved, resized, or shown
ContaineListener	Defines two methods to recognize when a component is added to or removed from a container
FocusListener	Defines two methods to recognize when a component gains or loses keyboard focus

ItemListener	Defines one method to recognize when the state of an item changes.

KeyListener	Defines three methods to recognize when a key is pressed, released, or typed.
MouseListener	Defines five methods to recognize when the mouse is clicked, enters a component, exists a component, is pressed, or is released.
MouseMotionListener	Defines two methods to recognize when the mouse is dragged or moved.
MouseWheelListener	Defines one method to recognize when the mouse wheel is moved. (Added by Java 2, version 1.4)
TextListener	Defines one method to recognize when a text value changes.
WindowFocusListener	Defines two methods to recognize when a window gains or looses input focus. (Added by Java 2, version 1.4)
WindowListener	Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Adapter Classes

- Java provides a special feature, called an adapter class, that can simplify the creation of event handlers in certain situations.
- An adapter class provides an empty implementation of all methods in an event listener interface.
- Adapter classes are useful when you want to receive and process only some of the events that are handled by a particular event listener interface.
- You can define a new class to act as an event listener by extending one of the adapter classes and implementing only those events in which you are interested.

Adapter Class	Listener Interface
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener

KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener

Example 1:

MyMouseAdapter implements the mouseClicked( ) method. The other mouse events are silently ignored by code inherited from the MouseAdapter class.

Example 2:

MyMouseMotionAdapter implements the mouseDragged( ) method. The other mouse motion event is silently ignored by code inherited from the MouseMotionAdapter class.

Using Push Buttons

- The JButton Class
- The JButton class provides the functionality of a push button.
- JButton allows an icon, a string, or both to be associated with the push button. Its constructors are:

JButton(Icon i)
JButton(String s)
JButton(String s, Icon i)

- Here, s and i are the string and icon used for the button.

Program 5

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

<applet code="JButtonDemo" width=250
height=300> </applet>
*/
public class JButtonDemo extends
JApplet implements ActionListener {
JTextField jtf;
public void init() {
// Get content pane
Container contentPane =
getContentPane();
contentPane.setLayout(new
FlowLayout()); // Add buttons to content
pane
ImageIcon france = new
ImageIcon("france.gif"); JButton jb = new
JButton(france);
jb.setActionCommand("France");
```

```
jb.addActionListener(this);
contentPane.add(jb);

ImageIcon germany = new
ImageIcon("germany.gif"); jb = new
JButton(germany);
jb.setActionCommand("Germany");
jb.addActionListener(this);
contentPane.add(jb);
ImageIcon italy = new
ImageIcon("italy.gif"); jb = new
JButton(italy);
jb.setActionCommand("Italy");
jb.addActionListener(this);
contentPane.add(jb);
ImageIcon japan = new
ImageIcon("japan.gif"); jb = new
JButton(japan);
jb.setActionCommand("Japan");
jb.addActionListener(this);
contentPane.add(jb);
// Add text field to content
pane jtf = new JTextField(15);

contentPane.add(jtf);
}
public void actionPerformed(ActionEvent ae)
{ jtf.setText(ae.getActionCommand());
}
}
```

Jtextfield

- JTextComponent class extends JComponent.
- It provides functionality that is common to Swing text components.
- One of its subclasses is JTextField, which allows you to edit one line of text. Its constructors are :

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

- Here, s is the string to be presented, and cols is the number of columns in the text field.

Program 6

```
import java.awt.*;
import
javax.swing.*; /*
<applet code="JTextFieldDemo" width=300
height=50> </applet>
*/
public class JTextFieldDemo extends JApplet
{ JTextField jtf;
public void init() {
// Get content pane
```

```
Container contentPane =
getContentPane();
contentPane.setLayout(new
FlowLayout()); // Add text field to content
pane
jtf = new
JTextField(15);
contentPane.add(jtf);
}
}
```

Jlabel

- Swing labels are instances of the JLabel class, which extends JComponent.
- It can display text and/or an icon.
- Its constructors are :

```
JLabel(Icon i)
Label(String
s)
JLabel(String s, Icon i, int align)
```

- Here, s and i are the text and icon used for the label.
- The align argument is either LEFT, RIGHT, CENTER, LEADING, or TRAILING.

ImageIcon

- In Swing, icons are encapsulated by the ImageIcon class, which paints an icon from an
- image.
- Two of its constructors are :

```
ImageIcon(String
filename) ImageIcon(URL
url)
```

- The first form uses the image in the file named filename.
- The second form uses the image in the resource identified by

url. Program 7

(Jlabel & Image Icon)

```
import java.awt.*;
import
javax.swing.*;
/*<applet code="JLabelDemo" width=250
height=150></applet>*/ public class JLabelDemo extends
JApplet {
public void init()
 {
// Get content pane
Container contentPane =
getContentPane(); // Create an icon
ImageIcon ii = new
ImageIcon("france.gif"); // Create a label
JLabel jl = new JLabel("France", ii,
JLabel.CENTER); // Add label to the content pane
contentPane.add(jl);
}
}
```

Swing Buttons

- Swing buttons provide features that are not found in the Button class defined by the AWT.

- Swing buttons are subclasses of the AbstractButton class, which extends JComponent.
- AbstractButton contains many methods that allow you to control the behavior of buttons, check boxes, and radio buttons.
- We can define different icons that are displayed for the component when it is "disabled", "pressed", or "selected". Another icon can be used as a "rollover" icon, which is displayed when the mouse is positioned over that component.
- The following are the methods that control this behavior:

  void setDisabledIcon(Icon
  di) void setPressedIcon(Icon
  pi) void
  setSelectedIcon(Icon si) void
  setRolloverIcon(Icon ri)
- Here, di, pi, si, and ri are the icons to be used for these different conditions.
- di = disabled,
- pi = pressed,
- si = selected,
- ri = rollover.

Trees

- A tree is a component that presents a hierarchical view of data.
- A user has the ability to expand or collapse individual subtrees in this display.
- Trees are implemented in Swing by the JTree class, which extends JComponent. Some of its constructors are shown here:

  JTree(Object obj[ ])
  JTree(TreeNode tn)
- Each element of the array obj is a child node in the first form.
- The tree node tn is the root of the tree in the second form.

Program 8

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import
javax.swing.tree.*;
/*<applet code="JTreeEvents" width=400
height=200></applet>*/ public class JTreeEvents extends
JApplet {
JTree tree;
JTextField jtf;
public void init() {
// Get content pane
Container contentPane = getContentPane();
// Set layout manager
contentPane.setLayout(new
BorderLayout());
// Create top node of tree
DefaultMutableTreeNode top = new
DefaultMutableTreeNode("Options"); // Create subtree of "A"
DefaultMutableTreeNode a = new
DefaultMutableTreeNode("A"); top.add(a);
```

```
DefaultMutableTreeNode a1 = new
DefaultMutableTreeNode("A1"); a.add(a1);
DefaultMutableTreeNode a2 = new
DefaultMutableTreeNode("A2"); a.add(a2);
// Create subtree of "B"
DefaultMutableTreeNode b = new
DefaultMutableTreeNode("B"); top.add(b);
DefaultMutableTreeNode b1 = new
DefaultMutableTreeNode("B1"); b.add(b1);
DefaultMutableTreeNode b2 = new
DefaultMutableTreeNode("B2"); b.add(b2);
DefaultMutableTreeNode b3 = new
DefaultMutableTreeNode("B3"); b.add(b3);
// Create tree
tree = new JTree(top);
// Add tree to a scroll pane
int v = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
int h = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;
JScrollPane jsp = new JScrollPane(tree, v, h);

// Add scroll pane to the content pane
contentPane.add(jsp,
BorderLayout.CENTER);
// Add text field to applet
jtf = new JTextField("", 20);
contentPane.add(jtf, BorderLayout.SOUTH);
// Anonymous inner class to handle mouse
clicks tree.addMouseListener(new
MouseAdapter() { public void
mouseClicked(MouseEvent me) {
doMouseClicked(me);
}
});
}
void doMouseClicked(MouseEvent me) {
TreePath tp = tree.getPathForLocation(me.getX(),
me.getY()); if(tp != null)
jtf.setText(tp.toString()
); else
jtf.setText("");
}
}
```

JMenuBar

- JMenuBar inherits Jcomponent.
- It has only one constructor, which is the default constructor.
- JMenuBar defines several methods, but often we only need to use one: add()
- The add() method adds Jmenu to the menu bar.

Jmenu add(Jmenu menu)

Here menu is a Jmenu instance that is added to the menu bar.

JMenu

- Jmenu encapsulates a menu, which is populated with JMenuItems.
- This enables one menu to be a submenu of another.

- Jmenu defines several constructors.
- The one that is mostly used is

Jmenu(String name) It creates a menu that has the title specified by name.

JMenuItem

- JMenuItem encapsulates an element in a menu.
- JMenuItem is derived from AbstractButton, and every item in a menu can be a thought of as a special kind of button.
- JMenuItem defines several constructors of which one is:

JMenuItem(String name) This creates a menu item with the name specified by name.

Creating a Main Menu

- To create a menu bar, first create an instance of JMenuBar.
- This class only defines the default constructor.
- Next, construct each menu that will be in the menubar.
- Following are the constructors for Menu:

  JMenu( )

  JMenu(String optionName)

  JMenu(String optionName, boolean removable)
- Here, optionName specifies the name of the menu selection.
- If removable is true, the pop-up menu can be removed and allowed to float free. Otherwise, it will remain attached to the menu bar.

Adding Mnemonics & Accelerators to Menu Items

- In real applications a menu usually includes support for keyboard shortcuts.
- These come in two forms 1 Mnemonics

  2 Accelerators
- 1. Mnemonis : As it applies to menus, a mnemonic defines a key that's lets you select an item from an active menu by typing a key.
- Thus, a mnemonic allows you to use the keyboard to select an item from a menu that is already

  being displayed.

A Mnemonic can be specified as follows

void setMnemonic(int mnem)

- 2. Accelerator : An Accelerator is a key that lets you select a menu item without having to activate the menu first. For example you might use "ctrl+s" to activate "save" function
- An Accelerator can be specified as follows by calling setAcceletor() as follows void setAccelerator(KeyStroke ks)

ShowMessageDialog

- ShowMessageDialog creates the simplest dialog that can be constructed.
- It displays a message and waits until the user presses "ok" button.
- Its version is shown here

static void showMessageDialog(Component parent,Object msg) throws HeadlessException Here parent specifies the component relative to which the dialog is displayed.

ShowConfirmDialog

- Another very common dialog type is one that requests a basic Yes/No response from the user.
- This is called "confirmation dialog".

- Its version is as follows:

  static int showConfirmDialog(Component parent, Object msg)throws HeadlessException

ShowInputDialog

- Although a Yes/No response is adequate for some simple dialogs, often other more flexible input is required.
- To handle this cases, JOptionPane provides two types of dialogs.
- Its simplest form is :

  Static String showInputDialog(Object msg) throws HeadlessException

ShowOptionDialog

- This method creates a dialog that contains the elements that we specify.
- Its method is specified as :

  static int showOptionDialog(Component parent,Object msg, String title, int optT,int msgT,Icon image,Object[] options,Object initVal) throws HeadlessException

JDialog

- Jdialog is the Swing class that creates Dialog.
- Jdialog is a top-level container that is not derived from Jcomponent.
- Jdialog defines many constructors.One of them is:

  JDialog(Frame parent,String title, boolean isModal)

- It creates a dialog whose owner is specified by parent. If isModal is true,then the dialog is modal.If isModal is false, then the dialog is modeless.

Creation of a Modeless Dialog

- JDialog creates a Modeless Dialog.
- Ex:

  JDialog(Frame parent, String title)

Here the owner is parent.The dialog has the title specified by “title”.

- To make the direction dialog modeless requires very few changes.
- First, change the call to the Jdialog constructor by removing the third argument, as shown: jdlg=new JDialog(jfrm, “Direction”);

Next,remove calls to setVisible(false) that are inside the event handlers for up and down buttons.

```
//Respond to the Up button in the dialog.
  jbtnUp.addActionListener(new
ActionListener()
  {
    public void actionPerformed(ActionEvent
    le)
    {
```

```
//Respond to the Down button in the dialog.
  jbtnDown.addActionListener(new
ActionListener()
  {

    public void actionPerformed(ActionEvent le)
    {
```

```
    jlab.setText("Direction is up");
   }
  } );
```

```
    jlab.setText("Direction is down");
   }
  } );
```