


Java Programming

MCA-205

UNIT III


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 48



Learning Objectives

- **Event Handling:** Different Mechanism, the Delegation Event Model, Event Classes, Event Listener Interfaces, Adapter and Inner Classes, Working with windows, Graphics and Text, using AWT controls, Layout managers and menus, handling Image, animation, sound and video, Java Applet.
- **The Collection Framework:** The Collection Interface, Collection Classes, Working with Maps & Sets
- **JDBC:** Introduction to DBMS & RDBMS, JDBC API, JDBC Application Architecture, Obtaining a Connection, JDBC Models: Two Tier and Three Tier Model, ResultSet, Prepared Statement, Callable Statement.


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 49



Introduction to Event Handling

- Applets are event-driven programs. Thus, event handling is at the core of successful applet programming.
- Most events to which your applet will respond are generated by the user. These events are passed to your applet in a variety of ways, with the specific method depending upon the actual event.
- There are several types of events. The most commonly handled events are those generated by the mouse, the keyboard, and various controls, such as a push button.
- Events are supported by the **java.awt.event** package.
- The way in which events are handled by an applet changed significantly between the original version of Java (1.0) and modern versions of Java, beginning with version 1.1.
- The 1.0 method of event handling is still supported, but it is not recommended for new programs. Also, many of the methods that support the old 1.0 event model have been deprecated. The modern approach is the way that events should be handled by all new programs, including those written for Java 2


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 50



The Delegation Event Model

- The modern approach to handling events is based on the *delegation event model*, 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.
- 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.


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III, .d



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.
- 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.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III, .d




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.


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III, .d



Event Sources

- Some sources may allow only one listener to register. The general form of such a method is this:
- `public void addTypeListener(TypeListener el) throws java.util.TooManyListenersException`
- Here, *Type* is the name of the event and *el* is a reference to the event listener.
- When such an event occurs, the registered listener is notified. This is known as *unicasting* the event.
- A source must also provide a method that allows a listener to unregister an interest in a specific type of event. The general form of such a method is this:
- `public void removeTypeListener(TypeListener el)`
- Here, *Type* is the name of the event and *el* is a reference to the event listener. For example, to remove a keyboard listener, you would call `removeKeyListener()`.
- The methods that add or remove listeners are provided by the source that generates events. For example, the **Component** class provides methods to add and remove keyboard and mouse event listeners.


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III



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.


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III



Event Classes

- The classes that represent events are at the core of Java's event handling mechanism.
- Thus, we begin our study of event handling with a tour of the event classes. As you will see, they provide a consistent, easy-to-use means of encapsulating events.
- 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.


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III



Event Classes

- 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()`
- 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 several types of events that are generated by various user interface elements.


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason
U III. 48



Event Classes

Action Event	KeyAdapter
AdjustmentEvent	KeyEvent
ComponentAdapter	MouseAdapter
ComponentEvent	MouseEvent
ContainerEvent	MouseMotionAdapter
FocusAdapter	MouseWheelEvent
FocusEvent	PaintEvent
HierarchyEvent	TextEvent
InputEvent	WindowAdapter
ItemEvent	WindowEvent


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason
U III. 49



The ActionEvent 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 modifier)`
- 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.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason
U III. 50




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.

BLOCK_DECREMENT	Block decrement adjustment type
BLOCK_INCREMENT	Block increment adjustment type
TRACK	Absolute tracking adjustment type
UNIT_DECREMENT	Unit decrement adjustment type
UNIT_INCREMENT	Unit increment adjustment type

In addition, there is an integer constant, **ADJUSTMENT_VALUE_CHANGED**, that indicates that a change has occurred. Here is one **AdjustmentEvent** constructor:
AdjustmentEvent(Adjustable src, int id, int type, int data)

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason
U III. 48




The ComponentEvent Class

- A **ComponentEvent** is generated when the size, position, or visibility of a component is changed.
- There are four types of component events. The **ComponentEvent** class defines integer constants that can be used to identify them. The constants and their meanings are shown here:

COMPONENT_HIDDEN	Component was rendered invisible
COMPONENT_MOVED	Component's position changed
COMPONENT_RESIZED	Component's size changed
COMPONENT_SHOWN	Component was made visible

ComponentEvent has the constructor: **ComponentEvent(Component src, int id, int type, int data)**. **ComponentEvent** is the superclass either directly or indirectly of **ContainerEvent**, **FocusEvent**, **KeyEvent**, **MouseEvent**, and **WindowEvent**. The **getComponent()** method returns the component that generated the event. It is shown here: **Component getComponent()**


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason
U III. 49



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.
- ContainerEvent** is a subclass of **ComponentEvent** and has this constructor: **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 **comp**.
- You can obtain a reference to the container that generated this event by using the **getContainer()** method: **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()**


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason
U III. 50



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)`
- You can determine the other component by calling **getOppositeComponent()**, shown here.
- Component `getOppositeComponent()`
- The opposite component is returned. This method was added by Java 2, version 1.4. 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**.


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason
U III. 48



The InputEvent Class

- The abstract class **InputEvent** is a subclass of **ComponentEvent** and is the superclass for component input events. Its subclasses are **KeyEvent** and **MouseEvent**.
- InputEvent** defines several integer constants that represent any modifiers, such as the control key being pressed, that might be associated with the event.
- To test if a modifier was pressed at the time an event is generated, use the **isAltDown()**, **isAltGraphDown()**, **isControlDown()**, **isMetaDown()**, and **isShiftDown()** methods. The forms of these methods are shown here:
 - `boolean isAltDown()`
 - `boolean isAltGraphDown()`
 - `boolean isControlDown()`
 - `boolean isMetaDown()`
 - `boolean isShiftDown()`


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason
U III. 49



The ItemEvent Class

- An **ItemEvent** is generated when a check box or a list item is clicked or when a checkable menu item is selected or deselected. There are two types of item events, which are identified by the following integer constants: **DESELECTED**, **SELECTED**
- 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 (i.e., **SELECTED** or **DESELECTED**) for the event. It is shown here:
 - `int getStateChange()`


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason
U III. 50



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 key presses result in characters. For example, pressing the SHIFT key does not generate a character.
- KeyEvent** is a subclass of **InputEvent**. Here are two of its constructors:
`KeyEvent(Component src, int type, long when, int modifiers, int code)`
`KeyEvent(Component src, int type, long when, int modifiers, int code, char ch)`
- `char getKeyChar()`
- `int getKeyCode()`


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason
U III. 48



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, **MOUSE_DRAGGED**, **MOUSE_ENTERED**, **MOUSE_EXITED**, **MOUSE_MOVED**, **MOUSE_PRESSED**, **MOUSE_RELEASED**, **MOUSE_WHEEL**
- 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)`
- The most commonly used methods in this class are **getX()** and **getY()**. These return the X and Y coordinates of the mouse when the event occurred. Their forms are shown here: `getX()`, `getY()`


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason
U III. 49



The MouseWheelEvent Class

- The **MouseWheelEvent** class encapsulates a mouse wheel event. It is a subclass of **MouseEvent** and was added by Java 2, version 1.4. 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.
- MouseWheelEvent** defines the following constructor.
`MouseWheelEvent(Component src, int type, long when, int modifiers, int x, int y, int clicks, boolean triggersPopup, int scrollHow, int amount, int count)`


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason
U III. 50



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)`
- 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. **Think of a text event notification as a signal to a listener that it should retrieve information from a specific text component.**


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. d



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 are :**WINDOW_ACTIVATED**,**WINDOW_CLOSED**,**WINDOW_CLOSING** **WINDOW_DEACTIVATED**,**WINDOW_DEICONIFIED**,**WINDOW_GAINED_FOCUS**,**WINDOW_ICONIFIED**,**WINDOW_LOST_FOCUS**,**WINDOW_OPENED**, **WINDOW_STATE_CHANGED**.
- **WindowEvent** is a subclass of **ComponentEvent**. It defines several constructors.
- `WindowEvent(Window src, int type)`
- `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)`


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. d



Sources of Events

- **Button**
- **Checkbox**
- **Choice**
- **List**
- **Menu Item**
- **ScrollBar**
- **Text components**
- **Window**


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. d



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. Ex:
- ActionListener
- AdjustmentListener
- ComponentListener
- ContainerListener
- FocusListener
- ItemListener
- KeyListener
- MouseListener

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 48



The ActionListener Interface

- This interface defines the **actionPerformed()** method that is invoked when an **action** event occurs. Its general form is :
- void actionPerformed(ActionEvent ae)


The AdjustmentListener Interface

- This interface defines the **adjustmentValueChanged()** method that is invoked **when** an adjustment event occurs. Its general form is :
- void adjustmentValueChanged(AdjustmentEvent ae)

The ComponentListener Interface

- This interface defines four methods that are invoked when a component is resized, moved, shown, or hidden. Their general forms are : void componentResized(ComponentEvent ce)
- void componentMoved(ComponentEvent ce)
- void componentShown(ComponentEvent ce)
- void componentHidden(ComponentEvent ce)

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 49




The ContainerListener Interface

- This interface contains two methods. When a component is added to a container, **componentAdded()** is invoked. **When a component is removed from a container, componentRemoved()** is invoked. Their general forms are :
- void componentAdded(ContainerEvent ce)
- void componentRemoved(ContainerEvent ce)

The FocusListener Interface

- This interface defines two methods. When a component obtains keyboard focus, **focusGained()** is invoked. **When a component loses keyboard focus, focusLost()** is called. Their general forms are :
- void focusGained(FocusEvent fe)
- void focusLost(FocusEvent fe)

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 50




The ItemListener Interface

- This interface defines the **itemStateChanged()** method that is invoked when the state of an item changes. Its general form is:
- `void itemStateChanged(ItemEvent ie)`

The KeyListener Interface

- This interface defines three methods. The **keyPressed()** and **keyReleased()** methods are invoked when a key is pressed and released, respectively. The **keyTyped()** method is invoked when a character has been entered. For example, if a user presses and releases the A key, three events are generated in sequence: key pressed, typed, and released. If a user presses and releases the HOME key, two key events are generated in sequence: key pressed and released. The general forms of these methods are : `void keyPressed(KeyEvent ke)`
- `void keyReleased(KeyEvent ke)` , `void keyTyped(KeyEvent ke)`

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 48




The MouseListener Interface

- This interface defines five methods. The general forms of these methods are shown here:
- `void mouseClicked(MouseEvent me)`
- `void mouseEntered(MouseEvent me)`
- `void mouseExited(MouseEvent me)`
- `void mousePressed(MouseEvent me)`
- `void mouseReleased(MouseEvent me)`

The MouseMotionListener Interface

- This interface defines two methods. Their general forms are shown here:
- `void mouseDragged(MouseEvent me)`
- `void mouseMoved(MouseEvent me)`

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 49



The MouseWheelListener Interface

- This interface defines the **mouseWheelMoved()** method that is invoked when the mouse wheel is moved. Its general form is: `void mouseWheelMoved(MouseWheelEvent mwe)`
- MouseWheelListener was added by Java 2, version 1.4.**


The TextListener Interface

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

The WindowFocusListener Interface added by Java 2, version 1.4.

- This interface defines two methods, their general forms are: `void windowGainedFocus(WindowEvent we)`
- `void windowLostFocus(WindowEvent we)`


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 50



The WindowListener Interface

- This interface defines seven methods. The general forms of these methods are
- void windowActivated(WindowEvent we)
- void windowClosed(WindowEvent we)
- void windowClosing(WindowEvent we)
- void windowDeactivated(WindowEvent we)
- void windowDeiconified(WindowEvent we)
- void windowIconified(WindowEvent we)
- void windowOpened(WindowEvent we)


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III, .d



Using the Delegation Event Model

- Applet programming using the delegation event model is actually quite easy. Just follow these two steps:
- 1. Implement the appropriate interface in the listener so that it will receive the type of event desired.
- 2. Implement code to register and unregister (if necessary) the listener as a recipient for the event notifications.
- Remember that a source may generate several types of events. Each event must be registered separately. Also, an object may register to receive several types of events, but it must implement all of the interfaces that are required to receive these events.


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III, .d



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.
- The commonly used adapter classes in **java.awt.event** are:
- **ComponentAdapter, ContainerAdapter, FocusAdapter, KeyAdapter, MouseAdapter, MouseMotionAdapter, WindowAdapter**

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III, .d




Inner Classes

- The basics of inner classes were explained previously. Here we will see why they are important.
- // This applet does NOT use an inner class.
- import java.applet.*; import java.awt.event.*;

```
/*<applet code="MousePressedDemo" width=200 height=100>
</applet>*/
public class MousePressedDemo extends Applet {
    public void init() {addMouseListener(new MyMouseAdapter(this));}
    class MyMouseAdapter extends MouseAdapter {
        MousePressedDemo mousePressedDemo;
        public MyMouseAdapter(MousePressedDemo mousePressedDemo) {
            this.mousePressedDemo = mousePressedDemo;}
        public void mousePressed(MouseEvent me) {
            mousePressedDemo.showStatus("Mouse Pressed.");}}

```


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 48



```
// Inner class demo.
import java.applet.*;
import java.awt.event.*;
/*<applet code="InnerClassDemo" width=200 height=100>
</applet>*/
public class InnerClassDemo extends Applet {
    public void init() {
        addMouseListener(new MyMouseAdapter());}
    class MyMouseAdapter extends MouseAdapter {
        public void mousePressed(MouseEvent me) {
            showStatus("Mouse Pressed");}}

```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 48




Anonymous Inner Classes

- An *anonymous inner class* is one that is not assigned a name. This section illustrates how an anonymous inner class can facilitate the writing of event handlers.
- // Anonymous inner class demo.

```
import java.applet.*; import java.awt.event.*;
/*<applet code="AnonymousInnerClassDemo" width=200 height=100>
</applet> */
public class AnonymousInnerClassDemo extends Applet {
    public void init() {
        addMouseListener(new MouseAdapter() {
            public void mousePressed(MouseEvent me) {
                showStatus("Mouse Pressed");}});}

```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 48




Introduction to AWT

- The Abstract Window Toolkit (AWT) provides the basic support for applets. The AWT contains numerous classes and methods that allow you to create and manage windows.
- Although the main purpose of the AWT is to support applet windows, it can also be used to create stand-alone windows that run in a GUI environment, such as Windows.

AWT Classes

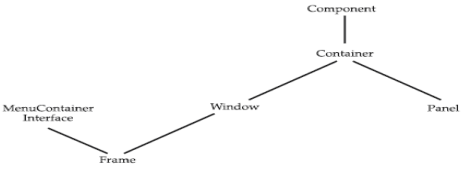
- The AWT classes are contained in the **java.awt package**. It is one of **Java's largest** packages. Fortunately, because it is logically organized in a top-down, hierarchical fashion, it is easier to understand and use.
- AWT Classes: AWTEvent, BorderLayout, Button, Canvas, CardLayout, Checkbox, CheckboxGroup, Choice, Color, Component, Container, Dialog, Dimension, Event, FlowLayout, Font, Frame, Graphics

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason
U III. 48



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 window.
- Much of the functionality of these windows is derived from their parent classes.



```

graph TD
    Component --> Container
    Container --> Window
    Container --> Panel
    Window --> MenuContainerInterface[MenuContainer Interface]
    Window --> Frame
  
```


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason
U III. 49



Component

- At the top of the AWT hierarchy is the **Component class**.
- Component is an abstract** class that encapsulates all of the attributes of a visual component. All user interface elements that are displayed on the screen and that interact with the user are subclasses of **Component**.
- It defines over a hundred public methods that are responsible for managing events, such as mouse and keyboard input, positioning and sizing the window, and repainting.
- A **Component object is responsible for remembering** the current foreground and background colors and the currently selected text font.


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason
U III. 50



Container

- The **Container** class is a subclass of **Component**.
- It has additional methods that allow other **Component** objects to be nested within it. Other **Container** objects can be stored inside of a **Container** (since they are themselves instances of **Component**).
- **This makes** for a multileveled containment system. A container is responsible for laying out (that is, positioning) any components that it contains.


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III, .d



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()**, or **setBounds()** methods defined by **Component**.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III, .d




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. When a **Frame** window is created by a program rather than an applet, a normal window is created.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III, .d




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.

Working with Frame Windows

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 48




Control Fundamentals

- The AWT supports the following types of controls:
 - Labels ■ Push buttons ■ Check boxes
 - Choice lists ■ Lists ■ Scroll bars ■ Text editing
- These controls are subclasses of **Component**.

Adding and Removing Controls

- To include a control in a window, you must add it to the window. To do this, you must first create an instance of the desired control and then add it to a window by calling **add()**, which is defined by **Container**.
- The **add()** method has several forms. One of the forms is :
- Component **add(Component compObj)**
- Here, **compObj** is an instance of the control that you want to add. Once a control has been added, it will automatically be visible whenever its parent window is displayed.


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 49



Removing Controls

- Sometimes you will want to remove a control from a window when the control is no longer needed. To do this, call **remove()** method 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()**.
- Responding to Controls
- Except for labels, which are passive controls, all controls generate events when they are accessed by the user.
- For example, when the user clicks on a push button, an event is sent that identifies the push button. In general, your program simply implements the appropriate interface and then registers an event listener for each control that you need to monitor.


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 50



Labels

- The easiest control to use is a label. A *label* is an object of type **Label**, and it contains a string, which it displays. Labels are passive controls that do not support any interaction with the user.
- Label** defines the following constructors:
 - `Label()`
 - `Label(String str)`
 - `Label(String str, int how)`
- The first version creates a blank label. The second version creates a label that contains the string specified by *str*. This string is left-justified. The third version creates a label that contains the string specified by *str* using the alignment specified by *how*. The value of *how* must be one of these three constants: **Label.LEFT**, **Label.RIGHT**, or **Label.CENTER**.


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason
U III. 48



Using Buttons

- 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()`
 - `Button(String str)`
- The first version creates an empty button. The second creates a button that contains *str* as a label.
- After a button has been created, you can set its label by calling **setLabel()**. You can retrieve its label by calling **getLabel()**. These methods are as follows:
 - `void setLabel(String str)`
 - `String getLabel()`
- Here, *str* becomes the new label for the button.


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason
U III. 49



Handling 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() { //Instantiate Button references
        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); }
}
```


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Anukiran Jain
U1. 48



Applying Check Boxes

- 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 check box that describes what option the box represents. You change the state of a check box by clicking on it. Check boxes can be used individually or as part of a group. Check boxes are objects of the **Checkbox** class.
- Checkbox** supports these constructors:
 - `Checkbox()`
 - `Checkbox(String str)`
 - `Checkbox(String str, boolean on)`
 - `Checkbox(String str, boolean on, CheckboxGroup cbGroup)`
 - `Checkbox(String str, CheckboxGroup cbGroup, boolean on)`
- `boolean getState()` //retrieve current state of a checkbox
- `void setState(boolean on)` //set checkbox state
- `String getLabel()` //current label with checkbox
- `void setLabel(String str)` //set label with a checkbox


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason
U III. .d



CheckboxGroup

- It is possible to create a set of mutually exclusive check boxes in which one and only one check box in the group can be checked at any one time.
- These check boxes are often called *radio buttons*.
- 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)`


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason
U III. .d



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.
- To add a selection to the list, call `add()`. It has this general form:
 - `void add(String name)`
- To determine which item is currently selected, you may call either `getSelectedItem()` or `getSelectedIndex()`. These methods are shown here:
 - `String getItem()`
 - `int getSelectedIndex()`
 - `int getItemCount()` //returns number of items in the list
 - `void select(int index)` //sets currently selected item
 - `void select(String name)`
 - `String getItem(int index)` //retrieves name associated with an item


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason
U III. .d



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


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason
U III. 48



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()/only one item to be selected at a time
 - List(int numRows)/number of entries always visible in a list
 - List(int numRows, boolean multipleSelect)
 - void add(String name)/to add a selection to a list
 - void add(String name, int index)
 - String getSelectedItem()/returns selected item for lists with single selection
 - int getSelectedIndex()


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason
U III. 48



Managing Scroll Bars

- Scroll bars are used to select continuous values between a specified minimum and maximum, they 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.
- Scrollbar defines the following constructors:**
 - Scrollbar(); Scrollbar(int style)
 - Scrollbar(int style, int initialValue, int thumbSize, int min, int max)


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason
U III. 48



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()`
- `TextField(int numChars)`
- `TextField(String str)`
- `TextField(String str, int numChars)`
- `String getText()`
- `void setText(String str)`


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason



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


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason



Understanding Layout Managers

- A layout manager automatically arranges your controls within a window by using some type of algorithm.
- 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)`
- Java has several predefined **LayoutManager** classes, you can use the layout manager that best fits your application.


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason



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. 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)`
- Valid values for *how* are as follows: `FlowLayout.LEFT`
`FlowLayout.CENTER`, `FlowLayout.RIGHT`


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. d



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)`
- **BorderLayout** defines the following constants that specify the regions:
- `BorderLayout.CENTER` `BorderLayout.SOUTH`
- `BorderLayout.EAST` `BorderLayout.WEST`
- `BorderLayout.NORTH`


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. d



Using Insets

- Sometimes you will want to leave a small amount of space between the container that holds your components and the window that contains it.
- To do this, override the **`getInsets()`** method that is defined by **Container**. This function returns an **Insets** object that contains the top, bottom, left, and right inset to be used when the container is displayed.
- These values are used by the layout manager to inset the components when it lays out the window. The constructor for **Insets** is shown here:
- `Insets(int top, int left, int bottom, int right)`
- The values passed in *top*, *left*, *bottom*, and *right* specify the amount of space between the container and its enclosing window.
- The **`getInsets()`** method has this general form:
- `Insets getInsets()`


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. d



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 shown here:
- `GridLayout()`
- `GridLayout(int numRows, int numColumns)`
- `GridLayout(int numRows, int numColumns, int horz, int vert)`


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 48



CardLayout

- 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:
- `CardLayout()`
- `CardLayout(int horz, int vert)`


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 49



Menu Bars and Menus

- A menu bar displays a list of top-level menu choices. Each choice is associated with a drop-down menu.
- This concept is implemented in Java by the following classes: **MenuBar**, **Menu**, and **MenuItem**.
- **In general, a menu bar contains one or more Menu objects. Each Menu object contains a list of MenuItem objects. Each MenuItem object represents something that can be selected by the user. Since Menu is a subclass of MenuItem, a hierarchy of nested submenus can be created. It is also possible to include checkable menu items.**
- These are menu options of type **CheckboxMenuItem** and will have a check mark next to them when they are selected.


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 50



Menu Bars and Menus

- To create a menu bar, first create an instance of **MenuBar**. **This class only defines the default constructor.** Next, create instances of **Menu** that will **define the selections** displayed on the bar. Following are the constructors for **Menu**:
- `Menu()`
- `Menu(String optionName)`
- `Menu(String optionName, boolean removable)`
- Individual menu items are of type **MenuItem**. **It defines these constructors:**
- `MenuItem()`
- `MenuItem(String itemName)`
- `MenuItem(String itemName, MenuShortcut keyAccel)`
- You can disable or enable a menu item by using:
- `void setEnabled(boolean enabledFlag), boolean isEnabled()`


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason



Dialog Boxes

- Dialog boxes are primarily used to obtain user input. They are similar to frame windows, except that dialog boxes are always child windows of a top-level window.
- Also, dialog boxes don't have menu bars. In other respects, dialog boxes function like frame windows.
- Dialog boxes are of type **Dialog**.
- Two commonly used constructors are shown here:
- `Dialog(Frame parentWindow, boolean mode)`
- `Dialog(Frame parentWindow, String title, boolean mode)`


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason



FileDialog

- Java provides a built-in dialog box that lets the user specify a file. To create a file dialog box, instantiate an object of type **FileDialog**. **This causes a file dialog box to be displayed.**
- Usually, this is the standard file dialog box provided by the operating system. **FileDialog** provides these constructors:
- `FileDialog(Frame parent, String boxName)`
- `FileDialog(Frame parent, String boxName, int how)`
- `FileDialog(Frame parent)`
- FileDialog()** provides methods that allow you to determine the **name of the file** and its path as selected by the user. Here are two examples:
- `String getDirectory()`
- `String getFile()`


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason



Handling Events by Extending AWT Components

- The delegation event model was introduced and all of the programs so far have used that design. But Java also allows you to handle events by subclassing AWT components.
- Doing so allows you to handle events in much the same way as they were handled under the original 1.0 version of Java. Of course, this technique is discouraged, because it has the same disadvantages of the Java 1.0 event model, the main one being inefficiency.
- To extend an AWT component, you must call the **enableEvents()** method of Component. Its general form is shown here:
- protected final void enableEvents(long *eventMask*)

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. .#



Handling Events by Extending AWT Components

- The *eventMask* argument is a bit mask that defines the events to be delivered to this component. The **AWTEvent** class defines **int** constants for making this mask. Several are :
- ACTION_EVENT_MASK, KEY_EVENT_MASK, ADJUSTMENT_EVENT_MASK, MOUSE_EVENT_MASK, COMPONENT_EVENT_MASK, MOUSE_MOTION_EVENT_MASK, CONTAINER_EVENT_MASK, MOUSE_WHEEL_EVENT_MASK, FOCUS_EVENT_MASK, TEXT_EVENT_MASK, INPUT_METHOD_EVENT_MASK, WINDOW_EVENT_MASK, ITEM_EVENT_MASK
- You must also override the appropriate method from one of your superclasses in order to process the event.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. .#





Image Fundamentals: Creating, Loading, & Displaying

- The AWT's **Image** class and the **java.awt.image** package, together, they provide support for *imaging* (the display and manipulation of graphical images).
- An *image* is simply a rectangular graphical object. Images are a key component of web design.
- The **** tag was used to include an image *inline with the flow of hypertext*. Java expands upon this basic concept, allowing images to be managed under program control.
- There are a large number of imaging classes and interfaces defined by **java.awt.image** and it is **not** possible to examine them all.
- CropImageFilter MemoryImageSource FilteredImageSource
PixelGrabber ImageFilter RGBImageFilter
- These are few of the interfaces: ImageConsumer ImageObserver
ImageProducer


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. .#



File Formats

- Originally, web images could only be in GIF format. The GIF image format was created by CompuServe in 1987 to make it possible for images to be viewed while online, so it was well suited to the Internet. GIF images can have only up to 256 colors each.
- This limitation caused the major browser vendors to add support for JPEG images in 1995. The JPEG format was created by a group of photographic experts to store full-color spectrum, continuous-tone images. These images, when properly created, can be of much higher fidelity as well as more highly compressed than a GIF encoding of the same source image.
- In almost all cases, you will never care or notice which format is being used in your programs. The Java image classes abstract the differences behind a clean interface.


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason
U III. .d



Creating an Image Object

- Images must eventually be painted on a window to be seen, the **Image** class doesn't have enough information about its environment to create the proper data format for the screen.
- Therefore, the **Component class in java.awt** has a factory method called **createImage()** that is used to create **Image** objects.
- The **createImage()** method has the following two forms:
 - `Image createImage(ImageProducer imgProd)`
 - `Image createImage(int width, int height)`
- Ex: `Canvas c = new Canvas();`
- `Image test = c.createImage(200, 100);`
- This creates an instance of **Canvas** and then calls the **createImage()** method to actually make an **Image** object. At this point, the image is blank. Later you will see how to write data to it.


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason
U III. .d



Loading an Image

- The other way to obtain an image is to load one. To do this, use the **getImage()** method defined by the **Applet** class.
- `Image getImage(URL url)`
- `Image getImage(URL url, String imageName)`
- Displaying an Image
- Once you have an image, you can display it by using **drawImage()**, which is a member of the **Graphics** class. It has several forms.
- `boolean drawImage(Image imgObj, int left, int top, ImageObserver imgOb)`

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason
U III. .d




```

/*
 * <applet code="SimpleImageLoad" width=248 height=146>
 * <param name="img" value="seattle.jpg">
 */
import java.awt.*;
import java.applet.*;
public class SimpleImageLoad extends Applet
{ Image img;
  public void init() {
    img = getImage(getDocumentBase(), getParameter("img"));
  }
  public void paint(Graphics g) {
    g.drawImage(img, 0, 0, this);
  }
}

```


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 48



Playing Sound


- Simplest way to retrieve and play a sound is to use the play() method of the Applet class. It retrieves and plays the sound as soon as possible after it is called.
- play() method supports various sound formats, for ex: Windows wave file format(.wav), Sun audio file format(.au), Music and Instruments Digital Interface File Format(.midi or .mid).
- Syntax: play(getCodeBase(), "sound.au")
- To play a sound more than once or to start or stop the sound one must load the sound into an AudioClip object using the applet's newAudioClip() method. AudioClip is a part of the Applet class and must be imported into your program
- Syntax: AudioClip au=newAudioClip(getCodeBase(), "audi/event.au");

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 49



The Collection Framework


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 50



Collections Introduction

- A *collection* — sometimes called a container — is simply an object that groups multiple elements into a single unit
- used to store, retrieve, manipulate, and communicate aggregate data


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 48



Types of Collection

- Java supplies several types of **Collection**:
 - **Set**: cannot contain duplicate elements, order is not important
 - **SortedSet**: like a **Set**, but order is important
 - **List**: may contain duplicate elements, order is important
- Java also supplies some “collection-like” things:
 - **Map**: a “dictionary” that associates *keys* with values, order is not important
 - **SortedMap**: like a **Map**, but order is important
- While you can get all the details from the Java API, you are expected to learn (i.e. *memorize*):
 - The signatures of the “most important” methods in each interface
 - The most important implementations of each interface

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 49



Collections Framework

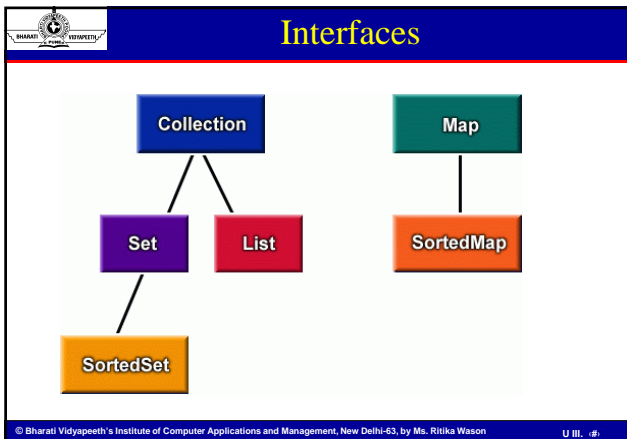
- unified architecture for representing and manipulating collecti
- **Interfaces**
 - abstract data types that represent collections.
 - allow collections to be manipulated independently of the details of their representation.
- **Implementations:**
 - concrete implementations of the collection interfaces.
 - reusable data structures.
- **Algorithms:**
 - methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces.
 - are said to be *polymorphic*: that is, the same method can be used on many different implementations of the appropriate collection interface.
 - In essence, algorithms are reusable functionality.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 50

Benefits of the Java Collections Framework

- Reduces programming effort
- Increases program speed and quality
- Allows interoperability among unrelated APIs
- Reduces effort to learn and to use new APIs
- Reduces effort to design new APIs
- Fosters software reuse


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason



Interfaces..

- all the core collection interfaces are generic. For example, this is the declaration of the Collection interface.
`public interface Collection<E>...`


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason



The Collection interface

- Collection is actually an interface
- Note that Java does not provide any direct implementations of *Collection*.
- Rather, concrete implementations are based on other interfaces which extend *Collection*, such as *Set*, *List*, etc.
- Still, the most general code will be written using *Collection* to type variables.


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 48



Creating a Collection

- All *Collection* implementations should have two constructors:
 - A no-argument constructor to create an empty collection
 - A constructor with another *Collection* as argument
- All the Sun-supplied implementations obey this rule, but--
- If you implement your own *Collection* type, this rule cannot be enforced, because an *Interface* cannot specify constructors


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 49



Operations Supported by Collection <E> interface

- The *Collection* interface specifies (among many other operations):
 - `boolean add(E o)`
 - `boolean contains(Object o)`
 - `boolean remove(Object o)`
 - `boolean isEmpty()`
 - `int size()`
 - `Object[] toArray()`
 - `Iterator<E> iterator()`
- You should learn *all* the methods of the *Collection* interface--all are important

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 50



Collection Interface


```

public interface Collection<E> extends Iterable<E> { // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    // optional
    boolean add(E element);
    // optional
    boolean remove(Object element);
    Iterator<E> iterator();
    // Bulk operations
    boolean containsAll(Collection<?> c);

    // optional
    boolean addAll(Collection<? extends E> c);
    // optional
    boolean removeAll(Collection<?> c);
    // optional
    boolean retainAll(Collection<?> c);
    // optional
    void clear();

    // Array operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
  
```


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason
U III. 48



Collection Interface Bulk Operations

- **containsAll**
 - returns true if the target Collection contains all of the elements in the specified Collection.
- **addAll**
 - adds all of the elements in the specified Collection to the target Collection.
- **removeAll**
 - removes from the target Collection all of its elements that are also contained in the specified Collection.
- **retainAll**
 - removes from the target Collection all its elements that are *not* also contained in the specified Collection.
- **Clear:** removes all elements from the Collection.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason
U III. 49



Traversing Collections

- **for-each Construct**
for (Object o : collection)
System.out.println(o);
- **Iterators**
 - An object that enables you to traverse through a collection and to remove elements from the collection selectively, if desired

```

public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); //optional
}
  
```

- The remove method may be called only once per call to next and throws an exception if this rule is violated.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason
U III. 50

The Iterator interface

- An iterator is an object that will return the elements of a collection, one at a time
- interface `Iterator<E>`
 - `boolean hasNext()`
 - ✓ Returns true if the iteration has more elements
 - `E next()`
 - ✓ Returns the next element in the iteration
 - `void remove()`
 - ✓ Removes from the underlying collection the last element returned by the iterator (optional operation)

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 48

Using an Iterator

- Two ways to iterate over Collections:
 - Iterator

```
static void loopThrough(Collection col){
    for (Iterator <E> iter = col.iterator(); iter.hasNext()) {
        Object obj=iter.next();
    }
}
```
 - For-each


```
static void loopThrough(Collection col){
    for (Object obj: col) {
        //access object
    }
}
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 49

Concrete Collections

concrete collection	implements	description
HashSet	Set	hash table
TreeSet	SortedSet	balanced binary tree
ArrayList	List	resizable-array
LinkedList	List	linked list
Vector	List	resizable-array
HashMap	Map	hash table
TreeMap	SortedMap	balanced binary tree
Hashtable	Map	hash table

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 50




Some implementations

- class `HashSet<E>` implements `Set`
- class `TreeSet<E>` implements `SortedSet`
- class `ArrayList<E>` implements `List`
- class `LinkedList<E>` implements `List`
- class `Vector<E>` implements `List`
 - class `Stack<E>` extends `Vector`
 - ✓ Important methods: `push`, `pop`, `peek`, `isEmpty`
- class `HashMap<K, V>` implements `Map`
- class `TreeMap<K, V>` implements `SortedMap`

• All of the above provide a no-argument constructor

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason
U III. 48




Set interface

- contains three general-purpose `Set` implementations
 - `HashSet`, `TreeSet`, and `LinkedHashSet`
- suppose you have a `Collection`, `c`, and you want to create another `Collection` containing the same elements but with all duplicates eliminated.

`Collection<Type> noDups = new HashSet<Type>(c);`

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason
U III. 48



Using Set

```
import java.util.*;
public class Test {
    public static void main(String[] args) {
        Set<String> ss = new LinkedHashSet<String>();
        for (int i = 0; i < args.length; i++)
            ss.add(args[i]);
        Iterator i = ss.iterator();
        while (i.hasNext())
            System.out.println(i.next());
    }
}
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason
U III. 48

Using Set . .

Adding unique objects to a collection

```
import java.util.*;
public class FindDups {
    public static void main(String[] args) {
        Set<String> s = new HashSet<String>();
        for (String a : args)
            if (!s.add(a))
                System.out.println("Duplicate detected: " + a);

        System.out.println(s.size() + " distinct words: " + s);
    }
}
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason

Set

- The following program takes the words in its argument list and prints out any duplicate words, the number of distinct words, and a list of the words with duplicates eliminated.

```
import java.util.*;
public class FindDups {
    public static void main(String[] args) {
        Set<String> s = new HashSet<String>();
        for (String a : args)
            if (!s.add(a))
                System.out.println("Duplicate detected: " + a);
        System.out.println(s.size() + " distinct words: " + s);
    }
}
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason

The SortedSet interface

- A **SortedSet** is a **Set** for which the order of elements is important
- interface **SortedSet<E>**
implements **Set**, **Collection**, **Iterable**
- Two of the **SortedSet** methods are:
 - E first()**
 - E last()**
- More interestingly, only **Comparable** elements can be added to a **SortedSet**, and the set's **Iterator** will return these in sorted order
- The **Comparable** interface is covered in a separate lecture

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason

The Map interface

- A map is a data structure for associating keys and values
- Interface `Map<K,V>`
- The two most important methods are:
 - `V put(K key, V value)` // adds a key-value pair to the map
 - `V get(Object key)` // given a key, looks up the associated value
- Some other important methods are:
 - `Set<K> keySet()`
 - ✓ Returns a set view of the keys contained in this map.
 - `Collection<V> values()`
 - ✓ Returns a collection view of the values contained in this map

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 48

Using Map

```


Map map = new HashMap(); // instantiate a concrete map
// ...
map.put(key, val); // insert a key-value pair
// ...
// get the value associated with key
Object val = map.get(key);
map.remove(key); // remove a key-value pair
// ...
if (map.containsValue(val)) { ... }
if (map.containsKey(key)) { ... }
Set keys = map.keySet(); // get the set of keys
// iterate through the set of keys
Iterator iter = keys.iterator();
while (iter.hasNext()) {
    Key key = (Key) iter.next(); // ...
}
  
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 49

The SortedMap interface

- A sorted map is a map that keeps the *keys* in sorted order
- Interface `SortedMap<K,V>`
- Two of the `SortedMap` methods are:
 - `K firstKey()`
 - `K lastKey()`
- More interestingly, only `Comparable` elements can be used as keys in a `SortedMap`, and the method `Set<K> keySet()` will return a set of keys whose iterator will return them sorted order


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 50



List

- An ordered collection
- Can have duplicates
- includes operations for the following
 - Positional access**
 - manipulates elements based on their numerical position in the list
 - Search**
 - searches for a specified object in the list and returns its numerical position
 - Iteration**
 - extends Iterator semantics to take advantage of the list's sequential nature
 - Range-view**
 - performs arbitrary *range operations* on the list.


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason
U III. 48



The List interface

- A list is an *ordered* sequence of elements
- `interface List<E>` extends `Collection`, `Iterable`
- Some important `List` methods are:
 - `void add(int index, E element)`
 - `E remove(int index)`
 - `boolean remove(Object o)`
 - `E set(int index, E element)`
 - `E get(int index)`
 - `int indexOf(Object o)`
 - `int lastIndexOf(Object o)`
 - `ListIterator<E> listIterator()`
 - ✓ A `ListIterator` is like an `Iterator`, but has, in addition, `hasPrevious` and `previous` methods

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason
U III. 49




Using List

```

List<Integer> myIntList = new LinkedList<Integer>(); // 1'
myIntList.add(new Integer(0)); // 2'
Integer x = myIntList.iterator().next(); // 3'


// Removes the 4-letter words from c
static void expurgate(Collection<String> c) {
    for (Iterator<String> i = c.iterator(); i.hasNext(); )
        if (i.next().length() == 4)
            i.remove();
}
  
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason
U III. 50



```
import java.util.*;
public class Shuffle {
public static void main(String[] args)
{
    List<String> list = Arrays.asList(args);
    Collections.shuffle(list);
    System.out.println(list);
}
}
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 48




Randomly shuffling a List

```
import java.util.*;

public class Shuffle {
    public static void main(String[] args) {
        List<String> list = Arrays.asList(args);
        Collections.shuffle(list);
        System.out.println(list);
    }
}
```

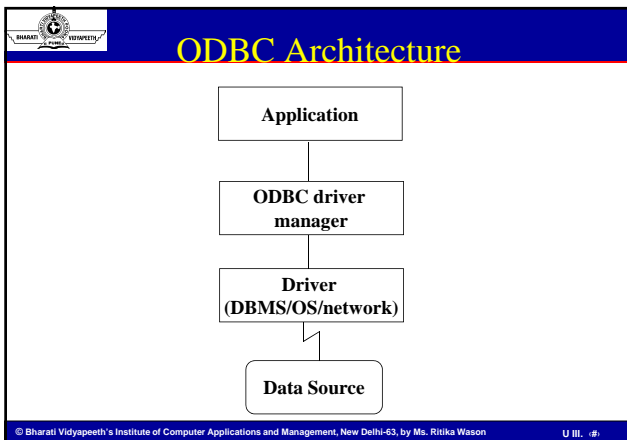
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 48



What is ODBC?

- ODBC is (Open Database Connectivity):
- A standard or open application programming interface (API) for accessing a database.
- SQL Access Group, chiefly Microsoft, in 1992
- By using ODBC statements in a program, you can access files in a number of different databases, including Access, dBase, DB2, Excel, and Text.
- It allows programs to use SQL requests that will access databases without having to know the proprietary interfaces to the databases.
- ODBC handles the SQL request and converts it into a request the individual database system understands.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 48



What is JDBC?


- JDBC is: Java Database Connectivity
 - is a Java API for connecting programs written in Java to the data in relational databases.
 - consists of a set of classes and interfaces written in the Java programming language.
 - provides a standard API for tool/database developers and makes it possible to write database applications using a pure Java API.
 - The standard defined by Sun Microsystems, allowing individual providers to implement and extend the standard with their own JDBC drivers.
- JDBC:
 - establishes a connection with a database
 - sends SQL statements
 - processes the results.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 49

JDBC vs ODBC

- ODBC is used between applications
- JDBC is used by Java programmers to connect to databases
- With a small "bridge" program, you can use the JDBC interface to access ODBC-accessible databases.
- JDBC allows SQL-based database access for EJB persistence and for direct manipulation from CORBA, DJB or other server objects

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 50

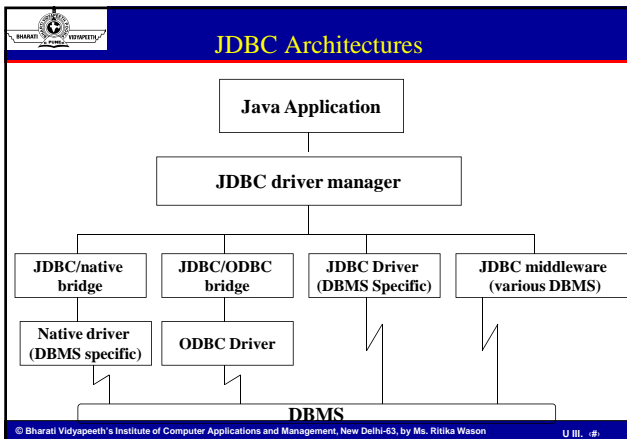



JDBC API

- The JDBC API supports both two-tier and three-tier models for database access.
- Two-tier model -- a Java applet or application interacts directly with the database.
- Three-tier model -- introduces a middle-level server for execution of business logic:
 - the middle tier to maintain control over data access.
 - the user can employ an easy-to-use higher-level API which is translated by the middle tier into the appropriate low-level calls.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason

U III. .d





The JDBC Steps

1. Importing Packages
2. Registering the JDBC Drivers
3. Opening a Connection to a Database
4. Creating a Statement Object
5. Executing a Query and Returning a Result Set Object
6. Processing the Result Set
7. Closing the Result Set and Statement Objects
8. Closing the Connection

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason

U III. .d

Types of JDBC drivers

- JDBC Drivers are divided into four types or levels. Each type defines a JDBC driver with increasingly higher levels of platform independence, performance and deployment administration.
- The four types are:
 - Type I: JDBC-ODBC bridge
 - Type II: Native-API/partly Java technology driver
 - Type III: Net-protocol fully Java enabled driver
 - Type IV: Native –protocol fully Java enabled driver

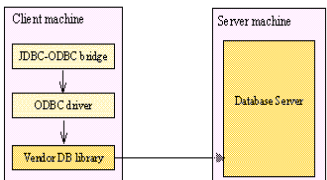
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III, .d

Type 1: JDBC-ODBC Bridge

- The type 1 driver, JDBC-ODBC Bridge, translates all JDBC calls into ODBC (Open Database Connectivity) calls and sends them to the ODBC driver. As such, the ODBC driver (native code), as well as, in many cases, the client database code must be present on the client machine using this driver. This kind of driver is generally most appropriate when automatic installation and downloading of a Java technology application is not important.
- Pros**
 - The JDBC-ODBC Bridge allows access to almost any database, since the database's ODBC drivers are already available. Type 1 drivers may be useful for those companies that have an ODBC driver already installed on client machines.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III, .d

Cons



- The performance is degraded since the JDBC call goes through the bridge to the ODBC driver, then to the native database connectivity interface. The result comes back through the reverse process. Considering the performance issue, type 1 drivers may not be suitable for large-scale applications.
- The ODBC driver and native connectivity interface must already be installed on the client machine. Thus any advantage of using Java applets in an intranet environment is lost, since the deployment problems of traditional applications remain.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III, .d

Type 2: Native-API/partly Java driver

- The native-API partly Java technology-enabled driver -- converts JDBC calls into database-specific calls for databases such as SQL Server, Informix, Oracle, or Sybase. The type 2 driver communicates directly with the database server; therefore it requires that some binary code be present on the client machine.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 48

Type 2: Native-API partly Java Driver (contd.)

- Pros**
- Type 2 drivers typically offer significantly better performance than the JDBC-ODBC Bridge.
- Cons**
- The vendor database library needs to be loaded on each client machine. Consequently, type 2 drivers cannot be used for the Internet. Type 2 drivers show lower performance than type 3 and type 4 drivers.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 49

Type 3: A Net-Protocol /all-Java Driver

The net-protocol fully Java-technology enabled driver follows a three-tiered approach whereby the JDBC database requests are passed through the network to the middle-tier server. The middle-tier server then translates the request (directly or indirectly) to the database-specific native-connectivity interface to further the request to the database server. If the middle-tier server is written in Java, it can use a type 1 or type 2 JDBC driver to do this.

A net-protocol fully Java technology-enabled driver translates JDBC API calls into a DBMS-independent net protocol which is then translated to a DBMS protocol by a server. This net server middleware is able to connect all of its Java technology-based clients to many different databases. The specific protocol used depends on the vendor. In general, this is the most flexible JDBC API alternative.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 50

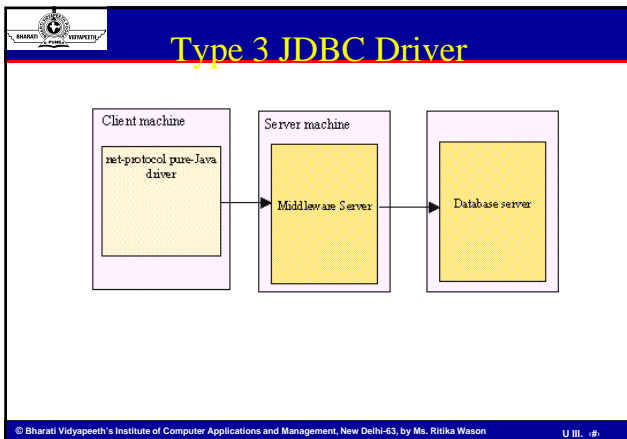
Pros

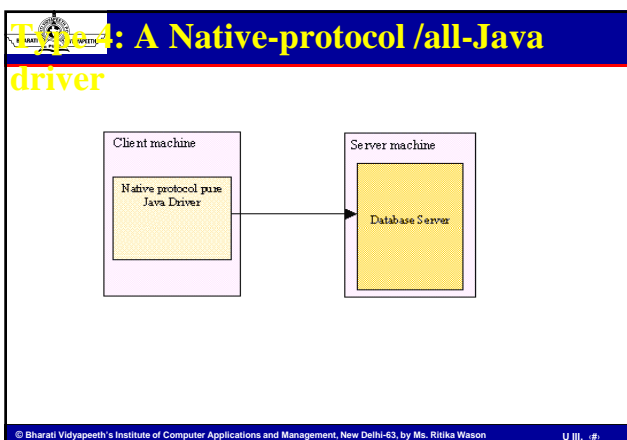
- The net-protocol/all-Java driver is server-based, so there is no need for any vendor database library to be present on client machines. Further, there are many opportunities to optimize portability, performance, and scalability. Moreover, the net protocol can be designed to make the client JDBC driver very small and fast to load. Additionally, provides support for features such as caching (connections, query results, and so on), load balancing, and advanced system administration such as logging and auditing.


Cons

Type 3 drivers require database-specific coding to be done in the middle tier. Additionally, traversing the recordset may take longer, since the data comes through the backend server.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 48








Type 4: JDBC Driver (Contd.)

- The native-protocol/all-Java driver (JDBC driver type 4) converts JDBC calls into the vendor-specific database management system (DBMS) protocol so that client applications can communicate directly with the database server. Level 4 drivers are completely implemented in Java to achieve platform independence and eliminate deployment administration issues. A *native-protocol fully Java technology-enabled driver* converts JDBC technology calls into the network protocol used by DBMSs directly. This allows a direct call from the client machine to the DBMS server and is a practical solution for Intranet access. Since many of these protocols are proprietary the database vendors themselves will be the primary source for this style of driver. Several database vendors have these in progress.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III, 41



Pros

- Since type 4 JDBC drivers don't have to translate database requests to ODBC or a native connectivity interface or to pass the request on to another server, performance is typically quite good. Moreover, the native-protocol/all-Java driver boasts better performance than types 1 and 2. Also, there's no need to install special software on the client or server. Further, these drivers can be downloaded dynamically.
- Cons*
- With type 4 drivers, the user needs a different driver for each database.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III, 41
