Java Workshop

DDUC ACM STUDENT CHAPTER
DEEN DAYAL UPADHYAYAYA COLLEGE
UNIVERSITY OF DELHI

Event Handling

AN INTEGRAL CONCEPT TO CREATION OF APPLETS AND OTHER TYPES OF GUI-BASED PROGRAMS

Event Handling

- One of most important points in Java.
- Integral to creation of applets and other GUI-based programs.
- Supported by a number of packages, including **java.util**, **java.awt**, and **java.awt.event**.

Events

- Most events are generated when user interacts with a GUI-based program.
- Events are passed to your program in a variety of ways, with the specific method dependent upon the actual event.
- There are several type of events :
  - Generated by the mouse
  - Generated by the keyboard
  - Generated by various GUI controls, such as button, check box, or radio button.

Event Handling Mechanism

- The way in which events are handled changed significantly between original version of Java(1.0) and modern version 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.
- Many of the methods that support the old 1.0 event model, have been deprecated.

Old Approach for Handling Events

- An event is propagated to up the containment hierarchy until it is handled by a component.
- This requires components to receive events that they did not process and this in result wastes valuable time.
- Modern approach for handling events eliminates this overhead.

Modern Approach for Handling Events

- Modern approach for handling events is based on "***Delegation Event Model***".
- This model defines standard and consistent mechanisms to generate and process events.
- According to *Delegation Event Model* :
  - a *source* generates an event and sends it to one or more *listeners*.
  - the *listener* simply waits until it receives an event.
  - once an event is received, the *listener* processes the event and then returns.
  - *listener* must register with a *source* in order to receive an event notification.

Advantages of Delegation Event Model

- Application logic that processes event (listener) is cleanly separated from the user interface logic that generated those events (source).
- That means user interface element is able to allot the processing of an event to a separate piece of code.
- Registration of listeners with source in order to receive an event notification provides extra benefits:
  - notifications are sent only to listeners that want to receive them which saves valuable time.

Events

- In the delegation model, an *event* is an object the 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.
- Pressing a button, entering a character via the keyboard, and clicking the mouse are some of the activities that cause events to be generated.
- 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.
- You are free to define events that are appropriate for you.

Event Sources

- A *source* is an object that generates an event.
- A source generates event when its internal state changes in some way.
- Sources may generate more than one type of event.
- Source must register listeners in order for the listeners to receive notifications about a specific type of event.

Event Sources

- Each type of event has its own registration method.
  - General form is void add*Type*Listener(*Type*Listener *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 is called **addKeyListener()**.
    - The method that registers a mouse motion listener is called **addMouseMotionListener()**.

Event Sources

- 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, notification are sent only to listeners that register to receive them.
- Some sources may allow only one listener to register.
  - General form of such a method is
    - void add*Type*Listener(*Type*Listener el) throws java.util.TooManyListenersException
    - Here, *type* and *el* have same meaning as previous definition of this method.
  - This is known as ***unicasting*** the event.

Event Listeners

- A ***listener*** is an object that is notified when an event occurs.
- It has two major requirements.
  - register with one or more sources to receive notifications about specific types of events.
  - 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 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.

Using the Delegation Event Model

- Only two simple steps are needed to use delegation event model.
  - Implement the appropriate interface in the listener so that it will receive the type of event desired.
  - Implement code to register and unregister (if necessary) the listener as a recipient for the event notification.
- A source may generate several type of events.
- Each event must be registered separately.
- An object may register to receive several type of events, but it must implement all of the interfaces that are required to receive these events.

Delegation Event Model Example

- ActionPerformedDemo
- FocusEventDemo

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.
- A new class can be defined then to act as an event listener by extending one of the adapter classes and implementing only events, which are to be used.

Adapter Classes

- For example, the **MouseMotionAdapter** class has two methods, **mouseDragged()** and **mouseMoved()**, which are the methods defined by the **MouseMotionListener** interface.
- If one is interested in mouse drag event only, then he can simply extend **MouseMotionAdapter** and override **mouseDragged()**.
- The empty implementation of **mouseMoved()** would handle the mouse motion events itself.

Adapter Classes

- Commonly used adapter classes in **java.awt.event** and interface that each implements are:

Adapter Class	Listener Interface
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener

AdapterClass Example

- AdapterDemo

Event Classes

- The classes that represent events are at the core of Java's event handling mechanism.
- Though Java defines several types of events but most widely used events are those defined by AWT and Swing.

EventObject Class

- At the root of Java event class hierarchy is **EventObject**, which is in **java.util**.
- It is the superclass for all events.
  - Its one constructor is EventObject(Object *src*).
  - Here, *src* is the object that generates this event.
- **EventObject** class contains two methods:
  - **getSource()** : returns the source of the event.
    - General form of this method is Object getSource().
  - **toString()** : returns the string equivalent of the event.
    - General form of this method is String toString().

AWTEvent Class

- The class **AWTEvent** is defined within **java.awt** package and 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.
  - General form of the method is int getID().

Commonly Used Event Classes in java.awt.event

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

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 superclass for all component input event classes.

Commonly Used Event Classes in java.awt.event

Event Class	Description
ItemEvent	Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item 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 mouse enters or exits component.
MouseWheelEvent	Generated when the mouse wheel is moved.
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.

The ActionEvent Class

- ActionEvent is generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
- Four integer constant are defined in this class and they can be used to identify any modifiers associated with an action event:
  - **ALT_MASK**, **CTRL_MASK**, **META_MASK**, and **SHIFT_MASK**.
  - An additional integer constant to identify action events named **ACTION_PERFORMED** is also defined in this class.

The ActionEvent Class

- **ActionEvent** has three constructors:
  - ActionEvent(Object *src*, int *type*, String *cmd*)
  - ActionEvent(Object *src*, int *type*, String *cmd*, int *modifier*)
  - ActionEvent(Object *src*, int *type*, String *cmd*, long when, int *modifier*)
- In these four constructors:
  - *src* is a reference to the object that generated this event.
  - *type* specifies the type of event.
  - *cmd* is its command string.
  - *modifier* indicates which modifier keys (Alt, Ctrl, Meta, and/or Shift) were pressed when the event was generated.
  - *when* specifies when the event occurred.

The ActionEvent Class

- To obtain the command name for the invoking object by using method **getActionCommand()**.
    - General form is String getActionCommand().
        - For example, when a button is pressed, an action event is generated with command name equal to the label on that button.
- **getModifiers()** method returns a value that indicates which modifier keys were pressed when then event was generated.
    - General form is int getModifiers().
- The method **getWhen()** returns the time at which the event took place. This is called the event's *timestamp*.
    - General form is long getWhen().

The ActionListener Interface

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

ActionEvent Example

- ActionPerformedDemo

The AdjustmentEvent Class

- AdjustmentEvent is generated by a scroll bar.
- There are five types of adjustment events.
- **AdjustmentEvent** class defines five integer constants to identify them.
  - **BLOCK_DECREMENT** : The user clicked inside the scroll bar to decrease its value.
  - **BLOCK_INCREMENT** : The user clicked inside the scroll bar to increase its value.
  - **TRACK** : The slider was dragged.
  - **UNIT_DECREMENT** : The button at the end of the scroll bar was clicked to decrease its value.
  - **UNIT_INCREMENT** : The button at the end of the scroll bar was clicked to increase its value.
- An additional integer constant **ADJUSTMENT_VALUE_CHANGED** is also defined to indicate that a change has occurred.

The AdjustmentEvent Class

- Here is the **AdjustmentEvent** constructor:
  - AdjustmentEvent(Adjustable *src*, int *id*, int *type*, int *data*)
    - *Here, src* is a reference to the object that generated this event.
    - *id* specifies the event.
    - *type* specifies the type of adjustment.
    - *data* is its associated data.
- **getAdjustable()** method returns the object that generated the event.
  - General form is Adjustable getAdjustable().
- **getAdjustmentType()**, method returns the type of adjustment. It returns one of the constants defined by **AdjustmentEvent** class.
  - General form is int getAdjustmentType().
- **getValue()**, method returns the amount of adjustment.
  - General form is int getValue().

The AdjustmentListener Interface

- This interface defines a method named **adjustmentValueChanged()**, that is invoked when an adjustment event occurs.
- General form of the method is:
  - void adjustmentValueChanged(AdjustmentEvent *ae*).

AdjustmentEvent Example

- AdjustmentEventDemo

The ComponentEvent Class

- ComponentEvent is generated when the size, position, or visibility of a component is changed.
- There are four types of component events.
- **ComponentEvent** class defines four integer constants to identify them.
  - **COMPONENT_HIDDEN** : The component was hidden.
  - **COMPONENT_MOVED** : The component was moved.
  - **COMPONENT_RESIZED** : The component was resized.
  - **COMPONENT_SHOWN** : The component became visible.

The ComponentEvent Class

- Here is the **ComponentEvent** constructor:
  - ComponentEvent(Component *src*, int *type*)
    - *Here, src* is a reference to the object that generated this event.
    - *type* specifies the type of event.
- **getComponent()** method returns the component that generated the event.
  - General form is Component getComponent().
- **ComponentEvent** is the superclass either directly or indirectly of **ContainerEvent**, **FocusEvent**, **KeyEvent**, **MouseEvent**, and **WindowEvent**, among others.

The ComponentListener Interface

- This interface defines four methods that are invoked when a component is resized, moved, shown, or hidden.
- General form of the methods are:
  - void ComponentResized(ComponentEvent *ce*).
  - void ComponentMoved(ComponentEvent *ce*).
  - void ComponentShown(ComponentEvent *ce*).
  - void ComponentHidden(ComponentEvent *ce*).

ComponentEvent Example

- CompponentEventDemo

The ContainerEvent Class

- ContainerEvent is generated when a component is added to or removed from a container. There are two types of component events.
- **ComponentEvent** class defines two integer constants to identify them.
  - **COMPONENT_ADDED** : A component has been added.
  - **COMPONENT_REMOVED** : A component has been removed.
- Here is the **ContainerEvent** constructor:
  - ContainerEvent(Component *src*, int *type*, Component *comp*)
    - *Here, src* is a reference to the container that generated this event.
    - *type* specifies the type of event.
    - *comp* specifies the component that has been added to or removed from the container.

The ContainerEvent Class

- **getContainer()** method returns a reference to the container that generated the event.
  - General form is Container getContainer().
- **getChild()** method returns a reference to the component that has been added to or removed from the container.
  - General form is Component getChild().

The ContainerListener Interface

- This interface defines two methods that are invoked when a component is added, or removed from container.
- General form of the methods are:
  - void componentAdded(ContainerEvent *ce*).
  - void componentRemoved(ContainerEvent *ce*).

The FocusEvent Class

- FocusEvent is generated when a component gains or loses input focus. There are two types of focus events.
- **FocusEvent** class defines two integer constants to identify them.
  - **FOCUS_GAINED** : A component has been added.
  - **FOCUS_LOST** : A component has been removed.

The FocusEvent Class

- **FocusEvent** class has three constructor:
  - FocusEvent(Component *src*, int *type*)
  - FocusEvent(Component *src*, int *type*, boolean *temporaryFlag*)
  - FocusEvent(Component *src*, int *type*, boolean *temporaryFlag*, Component *other*)
    - *Here, src* is a reference to the component that generated this event.
    - *type* specifies the type of event.
    - *temporaryFlag* is set to **true** if the focus is temporary. Otherwise, it is set to **false**.
      - A temporary focus event occurs as a result of another user interface operation. For example, if the focus is in a text field and user moves mouse to adjust scroll bar, the focus is temporarily lost for text field.
    - *other* specifies the other component involved in the focus change, called the *opposite component*.
      - Therefore, if a **FOCUS_GAINED** event occurred, *other* will refer to the component that lost focus and same goes in its opposite case.

The FocusEvent Class

- **getOppositeComponent()** method returns a reference to the other component.
  - General form is Component getOppositeComponent().
- **isTemporary()** method indicates if this focus change is temporary.
  - General form is boolean isTemporary().

The FocusListener Interface

- This interface defines two methods that are invoked when a component obtains keyboard focus and when a component loses keyboard focus.
- General form of the methods are:
  - void focusGained(FocusEvent *fe*).
  - void focusLost(FocusEvent *fe*).

FocusEvent Example

- FocusEventDemo

The InputEvent Class

- **InputEvent** class is an abstract class.
- It is subclass of **ComponentEvent** class and superclass for component input events like **KeyEvent** and **MouseEvent** class.
- **InputEvent** defines several integer constants that represent any modifiers. Originally, the **InputEvent** class defined the following eight values to represent the modifers:
  - ALT_MASK, ALT_GRAPH_MASK, BUTTON1_MASK, BUTTON2_MASK, BUTTON3_MASK, CTRL_MASK, META_MASK, SHIFT_MASK

The InputEvent Class

- However, because of possible conflicts between the modifiers used by keyboard events and mouse events, and other issues, the following extended modifier values were added:
  - ALT_DOWN_MASK, ALT_GRAPH_DOWN_MASK, BUTTON1_DOWN_MASK, BUTTON2_DOWN_MASK, BUTTON3__DOWNMASK, CTRL_DOWN_MASK, META_DOWN_MASK, SHIFT_DOWN_MASK
- When writing new code, its recommended that new, extended modifiers are used in place of original modifiers.

The InputEvent Class

- To test if a modifier was pressed at the time an event is generated, following methods can be used.
    - boolean isAltDown()
    - boolean isAltGraphDown()
    - boolean isControlDown()
    - boolean isMetaDown()
    - boolean isShiftDown()
- **getModifiers()** method can be used to obtain a value that contains all the original modifier flags.
    - Its general form is int getModifiers().
- **getModifiersEx()** method can be used to obtain the extended modifiers.
    - Its general form is int getModifiersEx().

The ItemEvent Class

- 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.
- **ItemEvent** class defines two integer constants to identify them.
  - **DESELECTED** : User deselects an item.
  - **SELECTED** : User selects an item.
- In addition **ItemEvent** class defines one integer constant **ITEM_STATE_CHANGED**, that signifies a change of state.

The ItemEvent Class

- **ItemEvent** class has one constructor:
  - ItemEvent(ItemSelectable *src*, int *type*, Object *entry*, int *state*)
    - *Here, src* is a reference to the component that generated this event. This might be a list or choice element.
    - *type* specifies the type of event.
    - *entry* specifies the specific item that generated the item event.
    - *state* is the current state of item.
- **getItem()** method can be used to obtain a reference to the item that generated an event.
  - General form is Object getItem().

The ItemEvent Class

- **getItemSelectable()** method can be used to obtain a reference to the **ItemSelectable** object that generated an event.
  - General form is ItemSelectable getItemSelectable().
  - Lists and choices are examples of user interface elements that implement the **ItemSelectable** interface.
- **getStateChanged()** method returns the state change (i.e. **SELECTED** or **DESELCTED)**
  - General form is int getStateChange().

The ItemListener Interface

- This interface defines a method which is invoked when the state of an item changed.
- General form of the method is:
  - void itemStateChanged(ItemEvent *fe*).

The KeyEvent Class

- KeyEvent is generated when keyboard input occurs. There are three types of key events.
- **KeyEvent** class defines three integer constants to identify them.
  - **KEY_PRESSED** : When a key is pressed.
  - **KEY_RELEASED** : When a key is released.
  - **KEY_TYPED** : When a character is generated.
    - Not all keypresses result in characters. For example, pressing Ctrl does not generate a character
- Many other integer constants are defined by **KeyEvent** class. For example, **VK_0** through **VK_9** and **VK_A** through **VK_Z** define ASCII equivalents of the numbers and letters.

The KeyEvent Class

- Some other integer constants defined in **KeyEvent** class are :
  - VK_ALT, VK_CANCEL, VK_CONTROL, VK_DOWN, VK_ENTER, VK_ESCAPE, VK_LEFT, VK_PAGE_DOWN, VK_PAGE_UP, VK_RIGHT, VK_SHIFT, VK_UP
- The **VK** constants specify ***virtual key codes*** and are independent of any modifiers, such as control, shift, or alt.

The KeyEvent Class

- **KeyEvent** class has one of its constructors as:
  - KeyEvent(Component *src*, int *type*, long *when*, int *modifiers*, int *code*, char *ch*)
    - *Here, src* is a reference to the component that generated this event. This might be a list or choice element.
    - *type* specifies the type of event.
    - *when* specifies the system time when the key was pressed.
    - *modifiers* specifies which modifiers were pressed when this key event occurred.
    - *code* specifies the virtual key code, such as **VK_UP**, **VK_A** and so forth.
    - *ch* specifies the character equivalent (if one exists). If no valid character exists, then *ch* contains **CHAR_UNDEFINED**. For **KEY_TYPED** events, *code* will contain **VK_UNDEFINED**.

The KeyEvent Class

- **KeyEvent** class defines several methods, but probably the most commonly used ones are :
  - **getKeyChar()**, which returns the character that was entered.
    - General form is char getKeyChar().
  - **getKeyCode()**, which returns the key code..
    - General form is int getKeyCode().
- If no valid character is available, then **getKeyChar()** returns **CHAR_UNDEFINED**. When a **KEY_TYPED** event occurs, **getKeyCode()** returns **VK_INDEFINED**.

The KeyListener Interface

- This interface defines three methods that are invoked when a key is pressed, released or a character has been entered.
- General form of the methods are:
  - void keyPressed(KeyEvent *ke*).
  - void keyReleased(KeyEvent *ke*).
  - void keyTyped(KeyEvent *ke*).
- When you press and release ‘Z’ key, sequence of events generated is: key pressed, typed, and released.
- When you press ‘SHIFT’ key, sequence of events generated is : key pressed, and released.

KeyEvent Example

- KeyEventDemo

The MouseEvent Class

- MouseEvent is generated when activity related to mouse takes place. There are eight types of mouse events.
- **MouseEvent** class defines eight integer constants to identify them.
  - **MOUSE_CLICKED** : The user clicked the mouse.
  - **MOUSE_DRAGGED** : The user dragged the mouse.
  - **MOUSE_ENTERED** : The mouse entered a component.
  - **MOUSE_EXITED** : The mouse exited from a component.
  - **MOUSE_MOVED** : The mouse moved.
  - **MOUSE_PRESSED** : The mouse was pressed.
  - **MOUSE_RELEASED** : The mouse was released.
  - **MOUSE_WHEEL** : The mouse wheel was moved.

The MouseEvent Class

- **MouseEvent** class has one of its constructors as:
  - MouseEvent(Component *src*, int *type*, long *when*, int *modifiers*, int *x*, int *y*, int *clicks*, boolean *triggersPopup*)
    - *Here, src* is a reference to the component that generated this event.
    - *type* specifies the type of event.
    - *when* specifies the system time when the mouse event was occurred.
    - *modifiers* specifies which modifiers were pressed when a mouse event occurred.
    - *x* and *y* are coordinates of mouse.
    - *clicks* specifies the click count.
    - *triggersPopup* flag indicates if this event causes a pop-up menu to appear on this platform.

The MouseEvent Class

- **getX()** and **getY()** are two commonly used methods which return the X and Y coordinates of the mouse within the component when the event occurred.
  - General form is int getX() and int getY().
- Alternatively, **getPoint()** can be used to obtain the coordinates of the mouse.
  - General form is Point getPoint().
- **translatePoint()** method changes the location of this event.
  - General form is void translatePoint(int *x*, int *y*).
    - Arguments *x* and *y* are added to coordinates of the event.
- **getClickCount()** method obtains the number of mouse clicks for this events.
  - General form is int getClickCount().
- **isPopupTrigger()** method checks if this event causes a pop-up menu to appear on this platform.
  - General form is boolean isPopupTrigger().

The MouseEvent Class

- **getButton()** method returns a value that represents the button that caused the event.
  - General form is int getButton().
  - Value return will be one of these constants defined by **MouseEvent** class:
    - NOBUTTON, BUTTON1, BUTTON2, and BUTTON3.
    - NOBUTTON value indicates that no button was pressed or released.
- Also available are three methods that obtain the coordinates of the mouse relative to the screen rather than the component. These methods are:
  - Point getLocationOnScreen() returns the point object containing both X and Y coordinates.
  - int getXOnScreen() returns X coordinate.
  - int getYOnScreen() returns Y coordinate.

The MouseListener Interface

- This interface defines five methods.
- General form of the methods and when are they invoked is:
  - void mousePressed (MouseEvent *me*)
    - When the mouse is pressed.
  - void mouseReleased(MouseEvent *me*)
    - When the mouse released.
  - void mouseClicked(MouseEvent *me*)
    - When the mouse is pressed and released at the same point.
  - void mouseEntered(MouseEvent *me*)
    - When mouse enters a component.
  - void mouseExited(MouseEvent *me*)
    - When mouse leaves a component.

The MouseMotionListener Interface

- This interface defines two methods that are invoked multiple times as the mouse is dragged and moved.
- General form of the methods are:
  - void mouseDragged(MouseEvent *me*).
  - void mouseMoved(MouseEvent *me*).

MouseEvent Example

- MouseListenerDemo

The MouseWheelEvent Class

- **MouseWheelEvent** class encapsulates a mouse wheel event.
- It is subclass of **MousEvent**.
- **MouseEvent** class defines two integer constants to identify two type of scrolls.
  - **WHEEL_BLOCK_SCROLL** : A page-up or page-down scroll event occurred.
  - **WHEEL_UNIT_SCROLL** : A line-up or line-down scroll event occurred.

The MouseWheelEvent Class

- **MouseWheelEvent** class has one of its constructors as:
  - MouseWheelEvent(Component *src*, int *type*, long *when*, int *modifiers*, int *x*, int *y*, int *clicks*, boolean *triggersPopup*, int *scrollHow*, int *amount*, int *count*)
    - *Here, src* is a reference to the component that generated this event.
    - *type* specifies the type of event.
    - *when* specifies the system time when the mouse event was occurred.
    - *modifiers* specifies which modifiers were pressed when a mouse event occurred.
    - *x* and *y* are coordinates of mouse.
    - *clicks* specifies the click count.
    - *triggersPopup* flag indicates if this event causes a pop-up menu to appear on this platform.
    - *scrollHow* value must be either **WHEEL_UNIT_SCROLL** or **WHEEL_BLOCK_SCROLL.**
    - *amount* specifies number of units to scroll.
    - *count* indicates the number of rotational units that the wheel moved.

The MouseWheelEvent Class

- **getWheelRotation()** method returns the number of rotational units.
  - General form is int getWheelRotation().
  - If returned value is positive, the wheel moved counterclockwise else it moved clockwise.
- Newly added method **getPreciseWheelRotation()** can be used to support high-resolution wheels.
  - General form is double getPreciseWheelRotation().
- **getScrollType ()** tells the type of scroll.
  - General form is int getScrollType().
  - Returns **WHEEL_UNIT_SCROLL** or **WHEEL_BLOCK_SCROLL**.
    - If scroll type is **WHEEL_UNIT_SCROLL**, number of units to scroll can be obtained using:
      - int getScrollAmount().

The MouseWheelListener Interface

- This interface defines a method that is invoked when the mouse wheel is moved.
- General form of the method is:
  - void mouseWheelMoved(MouseWheelEvent *mwe*).

MouseWheelEvent Example

- MouseWheelEventDemo

The TextEvent Class

- This class describes text events. These events are generated by text fields and text areas when characters are entered by user.
- **TextEvent** class defines one integer constant named **TEXT_VALUE_CHANGED**.
- **TextEvent** class has one of its constructors as:
  - TextEvent(Component *src*, int *type*)
    - *Here, src* is a reference to the component that generated this event. This might be a list or choice element.
    - *type* specifies the type of event.
- **TextEvent** object does not include the characters currently in the text component that generated the event.

The TextListener Interface

- This interface defines a method that is invoked when a change takes place in a text area or text field.
- General form of the methods are:
  - void textChanged(TextEvent *te*).

TextEvent Example

- TextEventDemo

The WindowEvent Class

- There are ten types of window events.
- **WindowEvent** class defines ten integer constants to identify them.
  - **WINDOW_ACTIVATED** : The window was activated.
  - **WINDOW_DEACTIVATED** : The window was deactivated.
  - **WINDOW_OPENED** : The window was opened.
  - **WINDOW_CLOSED** : The window has been closed.
  - **WINDOW_CLOSING** : The user requested that the windows be closed.
  - **WINDOW_STATE_CHANGED** : The state of the window changed.
  - **WINDOW_ICONIFIED** : The window was iconified.
  - **WINDOW_DEINCONIFIED** : The window was deiconified.
  - **WINDOW_GAINED_FOCUS** : The window gained input focus.
  - **WINDOW_LOST_FOCUS** : The window lost input focus.

The WindowEvent Class

- Few of the constructors of **WindowEvent** class are :
  - WindowEvent(Window *src*, int *type*)
    - *Here, src* is a reference to the object that generated this event.
    - *type* specifies the type of event.
  - WindowEvent(Window *src*, int *type*, Window *other*)
    - *other* specifies the opposite window when a focus or activation event occurs.
  - WindowEvent(Window *src*, int *type*, int *fromState*, int *toState*)
    - *fromState* specifies the prior state of window.
    - *toState* specifies the new state that the window will have when a window state change occurs.
  - WindowEvent(Window *src*, int *type*, Window *other*, int *fromState*, int *toState*)

The WindowEvent Class

- A commonly used method **getWindow()** returns the **Window** object that generated the event.
  - General form is Window getWindow().
- **WindowEvent** class defines methods to obtain the opposite window (when a focus or activation event has occurred), the previous window state and the current window state.
  - Window getOppositeWindow().
  - int getOldState().
  - int getNewState().

The WindowFocusListener Interface

- This interface defines two methods that are invoked when a window gains or loses input focus.
- General form of the methods are:
  - void windowGainedFocus (WindowEvent *we*).
  - void windowLostFocus(WindowEvent *we*).

WindowFocusListener Example

- WindowFocusListenerDemo

The WindowListener Interface

- This interface defines seven methods.
- General form of the methods and there description are:
  - void windowActivated(WindowEvent *we*)
    - When a window is activated.
  - void windowDeactivated(WindowEvent *we*)
    - When a window is deactivated.
  - void windowiconified(WindowEvent *we*)
    - When a window is iconified.
  - void windowDeiconified(WindowEvent *we*)
    - When a window is deiconified.
  - void windowOpened(WindowEvent *we*)
    - When a window is opened.
  - void windowClosed(WindowEvent *we*)
    - When a window is closed.
  - void windowClosing(WindowEvent *we*)
    - When a window is being closed.

WindowEvent Example

- WindowEventDemo

Inner Classes

- Inner class is a class that is defined within another class.
- It may be even defined within an expression.
- Inner class has access to all of the variables and methods within the scope of its parent class.
- By using inner classes, we no longer require passing objects of parent class as arguments to inner class.

Inner Classes

```
// 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.");
      }
}
```

```
// 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");
            }
      }
}
```

Anonymous Inner Classes

- An anonymous inner class is an inner class which is not assigned a name.
- For example:

```
// 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");
                }
        });
    }
}
```

- The syntax new MouseAdapter(){...} indicates to compiler that the code between the braces defines an anonymous inner class. That class extends **MouseAdapter**.

Inner and Anonymous Inner Classes

- Anonymous inner class is not named but it is automatically instantiated when the expression is executed.
- Both inner and anonymous inner classes are defined inside the parent class so they has access to all variables and methods of the parent class.
- Both type of classes solve some annoying problems in a simple yet effective way.
- They also allow one to create more effective code.