10.5. Event Handling

IT1406

Level I - Semester 1

10.5.1. Event handling mechanism

The Delegation Event Model

- The modern approach to handling events is based on the ***delegation event model,*** which defines standard and consistent mechanisms to generate and process events.
- Its concept is quite simple: a ***source*** generates an event and sends it to one or more ***listeners***.
- In this scheme, the listener simply waits until it receives an event. Once an event is received, the listener processes the event and then returns.
- The advantage of this design is that the application logic that processes events is cleanly separated from the user interface logic that generates those events.
- A user interface element is able to “delegate” the processing of an event to a separate piece of code.
- In the delegation event model, listeners must register with a source in order to receive an event notification. This provides an important benefit: notifications are sent only to listeners that want to receive them.
- This is a more efficient way to handle events than the design used by the original 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.

10.5.1. Event handling mechanism

Event

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

10.5.1. Event handling mechanism

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 thegeneral form:

  public 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 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 isknown as ***multicasting*** the event.
- In all cases, notifications are sent only to listeners that register to receive them.

10.5.1. Event handling mechanism

Event Sources

- Some sources may allow only one listener to register.
- The general form of such a method is this:

  public void add*Type*Listener(*Type*Listener *el*); throws java.util.TooManyListenersException
- 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 remove*Type*Listener(*Type*Listener *el*)
- 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.

10.5.1. Event handling mechanism

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, such as those 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.

10.5.2. Event classes

- The classes that represent events are at the core of Java's event handling mechanism.
- Thus, a discussion of event handling must begin with the event classes.
- It is important to understand, however, that Java defines several types of events and that not all event classes can be discussed
- 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*) ; ***src*** is the object that generates this event.
- **EventObject** defines two methods: **getSource( )** and **toString( )**.
- The **getSource( )** method returns the source of the event.
- Its general form is shown here:

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

  int getID( )

10.5.2. Event classes

- **EventObject** is a superclass of all events.
- **AWTEvent** is a superclass of all AWT events that are handled by the delegation event model.
- The package **java.awt.event** defines many types of events that are generated by various user interface elements. Table below shows several commonlv used event classes.

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.
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 the mouse enters or exits a 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.

10.5.2. Event classes

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 *modifiers*)
- ***src*** is a reference to the object that generated this event.
- The type of the event is specified by ***type,*** and its command string is *cmd.*
- The argument *modifiers* indicates which modifier keys (alt, ctrl, meta, and/or shift) were pressed when the event was generated.

10.5.2. Event classes

The ActionEvent Class

- The *when* parameter specifies when the event occurred.
- You can obtain the command name for the invoking **ActionEvent** object by using the **getActionCommand( )** method, shown here:

```
String getActionCommand( )
```

- For example, when a button is pressed, an action event is generated that has a command name equal to the label on that button.
- The **getModifiers( )** method returns a value that indicates which modifier keys (alt, ctrl, meta, and/or shift) were pressed when the event was generated.
- Its form is shown here:

```
int getModifiers( )
```

- The method **getWhen( )** returns the time at which the event took place.
- This is called the event's *timestamp.*
- The **getWhen( )** method is shown here:

```
long getWhen( )
```

10.5.2. Event classes

The AdjustmentEvent Class

- An **AdjustmentEvent** is generated by a scroll bar.
- There are five types of adjustment events.
- The **AdjustmentEvent** class defines integer constants that can be used to identify them.
- The constants and their meanings are shown in the ale below:

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.

- In addition, there is an integer constant, **ADJUSTMENT_VALUE_CHANGED**, that indicates that a change has occurred.

10.5.2. Event classes

The AdjustmentEvent Class

- Here is one **AdjustmentEvent** constructor:

  AdjustmentEvent(Adjustable *src*, int *id*, int *type*, int *val*)

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

  Adjustable getAdjustable( )

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

  int getAdjustmentType( )

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

  int getValue( )

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

10.5.2. Event classes

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	The component was hidden.
COMPONENT_MOVED	The component was moved.
COMPONENT_RESIZED	The component was resized.
COMPONENT_SHOWN	The component became visible.

- **ComponentEvent** has this constructor:

  ComponentEvent(Component *src*, int *type*)

- **ComponentEvent** is the superclass either directly or indirectly of **ContainerEvent**, **FocusEvent**, **KeyEvent**, **MouseEvent**, and **WindowEvent**, among others.
- The **getComponent( )** method returns the component that generated the event.
- It isshown here:

  Component getComponent( )

10.5.2. Event classes

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, 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, shown here:

  Container getContainer( )

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

  Component getChild( )

10.5.2. Event classes

The FocusEvent Class

- A **FocusEvent** is generated when a component gains or loses input focus.
- These events are identified by the integer constants **FOCUS_GAINED** and **FOCUS_LOST**.
- **FocusEvent** is a subclass of **ComponentEvent** and has these constructors:
  - FocusEvent(Component *src*, int *type*)
  - FocusEvent(Component *src*, int *type*, boolean *temporaryFlag*)
  - FocusEvent(Component *src*, int *type*, boolean *temporaryFlag*, Component *other*)
- Here, the argument *temporaryFlag* is set to **true** if the focus event is temporary. Otherwise, it is set to **false**. (A temporary focus event occurs as a result of another user interface operation. For example, assume that the focus is in a text field. If the user moves the mouse to adjust a scroll bar, the focus is temporarily lost.)
- The other component involved in the focus change, called the *opposite component,* is passed in *other.*
- Therefore, if a **FOCUS_GAINED** event occurred, *other* will refer to the component that lost focus.
- Conversely, if a **FOCUS_LOST** event occurred, *other* will refer to the component that gains focus.
- You can determine the other component by calling **getOppositeComponent( )**, shown here:

  Component getOppositeComponent( )
- The opposite component is returned.
- The **isTemporary( )** method indicates if this focus change is temporary. Its form is shown here:

  boolean isTemporary( ) ; returns **true** if the change is temporary. Otherwise, it returns **false**.

10.5.2. Event classes

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.
- Originally, the **InputEvent** class defined the following eight values to represent the modifiers:

ALT_MASK	BUTTON2_MASK	META_MASK
ALT_GRAPH_MASK	BUTTON3_MASK	SHIFT_MASK
BUTTON1_MASK	CTRL_MASK	

- 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	BUTTON2_DOWN_MASK	META_DOWN_MASK
ALT_GRAPH_DOWN_MASK	BUTTON3_DOWN_MASK	SHIFT_DOWN_MASK
BUTTON1_DOWN_MASK	CTRL_DOWN_MASK	

10.5.2. Event classes

The InputEvent Class

- When writing new code, it is recommended that you use the new, extended modifiers rather than the original modifiers.
- To test if a modifier was pressed at the time an event is generated, use the **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( )
- You can obtain a value that contains all of the original modifier flags by calling the **getModifiers( )** method. It is shown here:

  int getModifiers( )
- You can obtain the extended modifiers by calling **getModifiersEx( )**, which is shown here:

  int getModifiersEx( )

10.5.2. Event classes

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	The user deselected an item.
SELECTED	The user selected an item.

- 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*); 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 changed. Its signature is shown here:

  Object getItem( )
- The **getItemSelectable( )** method can be used to obtain a reference to the **ItemSelectable** object that generated an event. Its general form is shown here:

  ItemSelectable getItemSelectable( )
- Lists and choices are examples of user interface elements that implement the **ItemSelectable** interface.
- The **getStateChange( )** method returns the state change (that is, **SELECTED** or**DESELECTED**) for the event. It is shown here:

  int getStateChange( )

10.5.2. Event classes

The KeyEvent Class

- A **KeyEvent** is generated when keyboard input occurs.
- There are three types of key events, which are identified by these integer constants: **KEY_PRESSED**, **KEY_RELEASED**, and **KEY_TYPED**.
- The first two events are generated when any key is pressed or released.
- The last event occurs only when a character is generated.
- Remember, not all keypresses result in characters.
- For example, pressing shift does not generate a character.
- There are many other integer constants that are defined by **KeyEvent**.
- For example,**VK_0** through **VK_9** and **VK_A** through **VK_Z** define the ASCII equivalents of the numbers and letters. Here are some others:

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

10.5.2. Event classes

The KeyEvent Class

- The **VK** constants specify *virtual key codes* and are independent of any modifiers, such ascontrol, shift, or alt.
- **KeyEvent** is a subclass of **InputEvent**.
- Here is one of its constructors:
  - KeyEvent(Component *src*, int *type*, long *when*, int *modifiers*, int *code*, char *ch*)
- The system time at which the key was pressed is passed in *when.*
- The *modifiers* argument indicates which modifiers were pressed when this key event occurred.
- The virtual key code, such as **VK_UP**, **VK_A**, and so forth, is passed in *code.*
- The character equivalent (if one exists) is passed in *ch.* If no valid character exists, then *ch* contains **CHAR_UNDEFINED**.
- For **KEY_TYPED** events, *code* will contain **VK_UNDEFINED**.
- The **KeyEvent** class defines several methods, but probably the most commonly used ones are **getKeyChar( )**, which returns the character that was entered, and **getKeyCode( )**, which returns the key code.
- Their general forms are shown here:

  char getKeyChar( )

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

10.5.2. Event classes

The MouseEvent Class

- There are eight types of mouse events.
- The **MouseEvent** class defines the following integer constants that can be used to identify them:

MOUSE_CLICKED	The user clicked the mouse.
MOUSE_DRAGGED	The user dragged the mouse.
MOUSE_ENTERED	The mouse entered a component.
MOUSE_EXITED	The mouse exited from a component.
MOUSE_MOVED	The mouse moved.
MOUSE_PRESSED	The mouse was pressed.
MOUSE_RELEASED	The mouse was released.
MOUSE_WHEEL	The mouse wheel was moved.

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

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

- The *modifiers* argument indicates which modifiers were pressed when a mouse event occurred.
- The coordinates of the mouse are passed in *x* and *y.* The click count is passed in *clicks.*
- The *triggersPopup* flag indicates if this event causes a pop-up menu to appear on this platform.

10.5.2. Event classes

The MouseEvent Class

- Two commonly used methods in this class are getX( ) and getY( ).
- These return the X and Y coordinates of the mouse within the component when the event occurred.
- Their forms are shown here: int getX( ); int getY( )
- Alternatively, you can use the **getPoint( )** method to obtain the coordinates of the mouse. It is shown here:

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

  void translatePoint(int *x*, int *y*) ; the arguments *x* and *y* are added to the coordinates of the event.
- The **getClickCount( )** method obtains the number of mouse clicks for this event. Its signature is shown here:

  int getClickCount( )
- The **isPopupTrigger( )** method tests if this event causes a pop-up menu to appear on this platform. Its form is shown here:

  boolean isPopupTrigger( )
- Also available is the **getButton( )** method, shown here:

  int getButton( )

10.5.2. Event classes

The MouseEvent Class

- It returns a value that represents the button that caused the event. For most cases, the return value will be one of these constants defined by **MouseEvent**:

NOBUTTON	BUTTON1	BUTTON2	BUTTON3

- The **NOBUTTON** value indicates that no button was pressed or released.
- Also available are three methods that obtain the coordinates of the mouse relative to the screen rather than the component.
- They are shown here:

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

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

10.5.2. Event classes

The MouseWheelEvent Class

- The **MouseWheelEvent** class encapsulates a mouse wheel event.
- It is a subclass of **MouseEvent**.
- Not all mice have wheels. If a mouse has a wheel, it is typically located between the left and right buttons. Mouse wheels are used for scrolling.
- **MouseWheelEvent** de

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

- Here is one of the constructors defined by **MouseWheelEvent**:
  - MouseWheelEvent(Component *src*, int *type*, long *when*, int *modifiers*, int *x*, int *y*, int *clicks*, boolean *triggersPopup*, int *scrollHow*, int *amount*, int *count*)
- The *triggersPopup* flag indicates if this event causes a pop-up menu to appear on this platform.
- The *scrollHow* value must be either **WHEEL_UNIT_SCROLL** or **WHEEL_BLOCK_ SCROLL**.
- The number of units to scroll is passed in *amount.*
- The *count* parameter indicates the number of rotational units that the wheel moved.

10.5.2. Event classes

The MouseWheelEvent Class

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

```
int getWheelRotation( )
```

- It returns the number of rotational units.
- If the value is positive, the wheel moved counterclockwise.
- If the value is negative, the wheel moved clockwise.
- JDK 7 added a method called **getPreciseWheelRotation( )**, which supports high-resolution wheels.
- It works like **getWheelRotation( )**, but returns a **double**.
- To obtain the type of scroll, call **getScrollType( )**, shown next:

```
int getScrollType( )
```

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

```
int getScrollAmount( )
```

10.5.2. Event classes

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.
- This operation differs from other event objects discussed in this section.
- Think of a text event notification as a signal to a listener that it should retrieve information from a specific text component.

10.5.2. Event classes

The WindowEvent Class

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

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

10.5.2. Event classes

The WindowEvent Class

- **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*)
- Here, *other* specifies the opposite window when a focus or activation event occurs.
- The *fromState* specifies the prior state of the window, and *toState* specifies the new state that thewindow will have when a window state change occurs.
- A commonly used method in this class is **getWindow( )**, which returns the **Window** object that generated the event.
- Its general form is shown here:

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

  Window getOppositeWindow( )

  int getOldState( )

  int getNewState( )

10.5.3. Sources of event

- Table below lists some of the user interface components that can generate the
- In addition to these graphical user interface elements, any class derived from **Component**, such as **Applet**, can generate events.
- For example, you can receive key and mouse events from an applet.

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

10.5.4. Event listener interfaces

- As explained, 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.
- Table given lists several commonly used listener interfaces and provides a brief description of the methods that they define.

Interface	Description
ActionListener	Defines one method to receive action events.
AdjustmentListener	Defines one method to receive adjustment events.
ComponentListener	Defines four methods to recognize when a component is hidden, moved, resized, or shown.
ContainerListener	Defines two methods to recognize when a component is added to or removed from a container.
FocusListener	Defines two methods to recognize when a component gains or loses keyboard focus.
ItemListener	Defines one method to recognize when the state of an item changes.
KeyListener	Defines three methods to recognize when a key is pressed, released, or typed.
MouseListener	Defines five methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released.
MouseMotionListener	Defines two methods to recognize when the mouse is dragged or moved.

10.5.4. Event listener interfaces

The ActionListener Interface

- This interface defines the actionPerformed( ) method that is invoked when an action event occurs.
- Its general form is shown here:
  - 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 shown here:
  - 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 shown here:
  - void componentResized(ComponentEvent ce)
  - void componentMoved(ComponentEvent ce)
  - void componentShown(ComponentEvent ce)
  - void componentHidden(ComponentEvent ce)

10.5.4. Event listener interfaces

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 shown here:
  - 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 shown here:
  - void focusGained(FocusEvent fe)
  - void focusLost(FocusEvent fe)

The ItemListener Interface

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

10.5.4. Event listener interfaces

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 shown here:
  - void keyPressed(KeyEvent ke)
  - void keyReleased(KeyEvent ke)
  - void keyTyped(KeyEvent ke)

The MouseListener Interface

- This interface defines five methods.
- If the mouse is pressed and released at the same point, mouseClicked( ) is invoked. When the mouse enters a component, the mouseEntered( ) method is called.
- When it leaves, mouseExited( ) is called. The mousePressed( ) and mouseReleased( ) methods are invoked when the mouse is pressed and released, respectively.

10.5.4. Event listener interfaces

- 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. The mouseDragged( ) method is called multiple times as the mouse is dragged. The mouseMoved( ) method is called multiple times as the mouse is moved.
- Their general forms are shown here:
  - void mouseDragged(MouseEvent me)
  - void mouseMoved(MouseEvent me)

The MouseWheelListener Interface

- This interface defines the mouseWheelMoved( ) method that is invoked when the mouse wheel is moved.
- Its general form is shown here:
  - void mouseWheelMoved(MouseWheelEvent mwe)

10.5.4. Event listener interfaces

The TextListener Interface

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

The WindowFocusListener Interface

- This interface defines two methods: windowGainedFocus( ) and windowLostFocus( ).
- Theseare called when a window gains or loses input focus.
- Their general forms are shown here:
  - void windowGainedFocus(WindowEvent we)
  - void windowLostFocus(WindowEvent we)

10.5.4. Event listener interfaces

The WindowListener Interface

- This interface defines seven methods. The windowActivated( ) and windowDeactivated( ) methods are invoked when a window is activated or deactivated, respectively.
- If a window is iconified, the windowIconified( ) method is called.
- When a window is deiconified, the windowDeiconified( ) method is called.
- When a window is opened or closed, the windowOpened( ) or windowClosed( ) methods are called, respectively.
- The windowClosing( ) method is called when a window is being closed.
- 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)

10.5.5. Using the delegation event model

- It is time to see it in practice.
- 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 can 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.
- To see how the delegation model works in practice, we will look at examples that handle two commonly used event generators: the mouse and keyboard.

Handling Mouse Events

- To handle mouse events, you must implement the **MouseListener** and the **MouseMotionListener** interfaces. (You may also want to implement **MouseWheelListener**)
- The following applet demonstrates the process.
- It displays the current coordinates of the mouse in the applet's status window.
- Each time a button is pressed, the word "Down" is displayed at the location of the mouse pointer.

10.5.5. Using the delegation event model

- Each time the button is released, the word "Up" is shown.
- If a button is clicked, the message "Mouse clicked" is displayed in the upper-left corner of the applet display area.
- As the mouse enters or exits the applet window, a message is displayed in the upper-left corner of the applet display area.
- When dragging the mouse, a * is shown, which tracks withthe mouse pointer as it is dragged. Notice that the two variables, **mouseX** and **mouseY**, store the location of the mouse when a mouse pressed, released, or dragged event occurs.
- These coordinates are then used by **paint( )** to display output at the point of these occurrences.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="MouseEvents" width=300 height=100>
</applet>
*/
public class MouseEvents extends Applet implements MouseListener,MouseMotionListener
{
    String msg = "";
    int mouseX = 0, mouseY = 0; // coordinates of mouse
    public void init() {
        addMouseListener(this);
        addMouseMotionListener(this);
    }
```

10.5.5. Using the delegation event model

```
public void mouseClicked(MouseEvent me) {
  mouseX = 0;
  mouseY = 10;
  msg = "Mouse clicked.";
 repaint();
}

public void mouseEntered(MouseEvent me) {
  mouseX = 0;
  mouseY = 10;
  msg = "Mouse entered.";
  repaint();
}
public void mouseExited(MouseEvent me) {
  mouseX = 0;
  mouseY = 10;
  msg = "Mouse exited.";
  repaint();
}
public void mousePressed(MouseEvent me) {
   mouseX = me.getX();
   mouseY = me.getY();
   msg = "Down";
   repaint();
}
```

10.5.5. Using the delegation event model

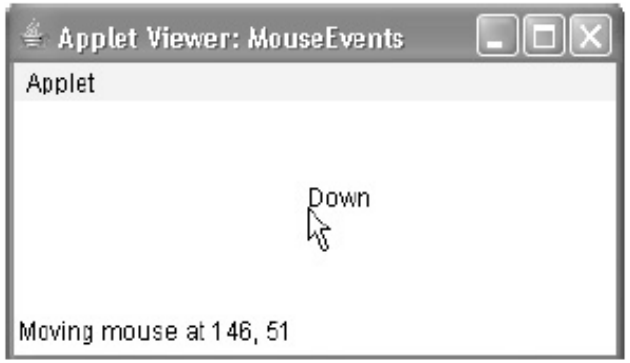
```
    public void mouseReleased(MouseEvent me) {
       mouseX = me.getX();
       mouseY = me.getY();
       msg = "Up";
       repaint();
    }

   public void mouseDragged(MouseEvent me) {
      mouseX = me.getX();
      mouseY = me.getY();
      msg = "*";
      showStatus("Dragging mouse at " + mouseX + ", " + mouseY);
      repaint();
   }

   public void mouseMoved(MouseEvent me) {
     showStatus("Moving mouse at " + me.getX() + ", " + me.getY());
   }

   public void paint(Graphics g) {
     g.drawString(msg, mouseX, mouseY);
   }
}
```

10.5.5. Using the delegation event model



- Sample output from this program is shown in this diagram:
- Let's look closely at this example.
- The **MouseEvents** class extends **Applet** and implements both the **MouseListener** and **MouseMotionListener** interfaces.
- These two interfaces contain methods that receive and process the various types of mouse events.
- Notice that the applet is both the source and the listener for these events.
- This works because **Component**, which supplies the **addMouseListener( )** and **addMouseMotionListener( )** methods, is a superclass of **Applet**.
- Being both the source and the listener for events is a common situation for applets.
- Inside **init( )**, the applet registers itself as a listener for mouse events.
- This is done by using **addMouseListener( )** and **addMouseMotionListener( )**, which, as mentioned, are members of **Component**. They are shown here:
  - void addMouseListener(MouseListener *ml*)
  - Void addMouseMotionListener(MouseMotionListener *mml*)
- *ml* is a reference to the object receiving mouse events, and *mml* is a reference to the object receiving mouse motion events.
- In this program, the same object is used for both.
- The applet then implements all of the methods defined by the **MouseListener** and **MouseMotionListener** interfaces.
- These are the event handlers for the various mouseevents. Each method handles its event and then returns.

10.5.5. Using the delegation event model

Handling Keyborad Events

- To handle keyboard events, you use the same general architecture as that shown in the mouse event example in the preceding section.
- The difference, of course, is that you will be implementing the **KeyListener** interface.
- Before looking at an example, it is useful to review how key events are generated.
- When a key is pressed, a **KEY_PRESSED** event is generated.
- This results in a call to the **keyPressed( )** event handler. When the key is released, a **KEY_RELEASED** event is generated and the **keyReleased( )** handler is executed.
- If a character is generated by the keystroke, then a **KEY_TYPED** event is sent and the **keyTyped( )** handler is invoked.
- Thus, each time the user presses a key, at least two and often three events are generated.
- If all you care about are actual characters, then you can ignore the information passed by the key press and release events.
- However, if your program needs to handle special keys, such as the arrow or function keys, then it must watch for them through the **keyPressed( )** handler.
- The following program demonstrates keyboard input. It echoes keystrokes to the applet window and shows the pressed/released status of each key in the status window.

10.5.5. Using the delegation event model

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="SimpleKey" width=300 height=100>
</applet>
*/
public class SimpleKey extends Applet
   implements KeyListener {
     String msg = "";
     int X = 10, Y = 20; // output coordinates
     public void init() {
        addKeyListener(this);
     }
     public void keyPressed(KeyEvent ke) {
        showStatus("Key Down");
     }
     public void keyReleased(KeyEvent ke) {
        showStatus("Key Up");
     }
```

10.5.5. Using the delegation event model

```
    public void keyTyped(KeyEvent ke) {
        msg += ke.getKeyChar();
        repaint();
    }
    public void paint(Graphics g) {
        g.drawString(msg, X, Y);
    }
}
```

- Sample output is shown here:

- If you want to handle the special keys, such as the arrow or function keys, you need to respond to them within the **keyPressed( )** handler.
- They are not available through **keyTyped( )**.

10.5.6. Understanding 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.
- For example, the **MouseMotionAdapter** class has two methods, **mouseDragged( )** and **mouseMoved( )**, which are the methods defined by the **MouseMotionListener** interface.
- If you were interested in only mouse drag events, then you could simply extend **MouseMotionAdapter** and override **mouseDragged( )**.
- The empty implementation of **mouseMoved( )** would handle the mouse motion events for you.
- Table below lists several commonly used adapter classes in **java.awt.event** and notes the interface that each implements.

10.5.6. Understanding adapter classes

Adapter Class	Listener Interface
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener and (as of JDK 6) MouseMotionListener and MouseWheelListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener, WindowFocusListener, and WindowStateListener

- The following example demonstrates an adapter.
- It displays a message in the status bar of an applet viewer or browser when the mouse is clicked or dragged.
- However, all other mouse events are silently ignored.
- The program has three classes. AdapterDemo extends Applet.

10.5.6. Understanding adapter classes

- Its init( ) method creates an instance of MyMouseAdapter and registers that object to receive notifications of mouse events.
- It also creates an instance of MyMouseMotionAdapter and registers that object to receive notifications of mouse motion events.
- Both of the constructors take a reference to the applet as an argument.
- **MyMouseAdapter** extends **MouseAdapter** and overrides the **mouseClicked( )** method.
- The other mouse events are silently ignored by code inherited from the **MouseAdapter** class.
- **MyMouseMotionAdapter** extends **MouseMotionAdapter** and overrides the **mouseDragged( )** method.
- The other mouse motion event is silently ignored by code inherited from the **MouseMotionAdapter** class.
- Note that both of the event listener classes save a reference to the applet.
- This information is provided as an argument to their constructors and is used later to invoke the **showStatus( )** method.

10.5.6. Understanding adapter classes

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="AdapterDemo" width=300 height=100>
</applet>
*/
   public class AdapterDemo extends Applet {
      public void init() {
         addMouseListener(new MyMouseAdapter(this));
         addMouseMotionListener(new MyMouseMotionAdapter(this));
      }
  }
   class MyMouseAdapter extends MouseAdapter {
       AdapterDemo adapterDemo;
       public MyMouseAdapter(AdapterDemo adapterDemo) {
          this.adapterDemo = adapterDemo;
       }

       public void mouseClicked(MouseEvent me) {
          adapterDemo.showStatus("Mouse clicked");
       }
   }

   class MyMouseMotionAdapter extends MouseMotionAdapter {
     AdapterDemo adapterDemo;
     public MyMouseMotionAdapter(AdapterDemo adapterDemo) {
         this.adapterDemo = adapterDemo;
    }
     public void mouseDragged(MouseEvent me) {
          adapterDemo.showStatus("Mouse dragged");
    }
}
```

10.5.6. Understanding adapter classes

- As you can see by looking at the program, not having to implement all of the methods defined by the **MouseMotionListener** and **MouseListener** interfaces saves you a considerable amount of effort and prevents your code from becoming cluttered with empty methods.
- As an exercise, you might want to try rewriting one of the keyboard input examples shown earlier so that it uses a **KeyAdapter**.

10.5.7. Inner classes in action

- Recall that an *inner class* is a class defined within another class, or even within an expression.
- This part of the lecture, illustrates how inner classes can be used to simplify the code when using event adapter classes.
- To understand the benefit provided by inner classes, consider the applet shown in the following listing.
- It *does not* use an inner class.
- Its goal is to display the string "Mouse Pressed“ in the status bar of the applet viewer or browser when the mouse is pressed.
- There are two top-level classes in this program.
- **MousePressedDemo** extends **Applet**, and **MyMouseAdapter** extends **MouseAdapter**.
- The **init( )** method of **MousePressedDemo** instantiates **MyMouseAdapter** and provides this object as an argument to the **addMouseListener( )** method.
- Notice that a reference to the applet is supplied as an argument to the **MyMouseAdapter** constructor.
- This reference is stored in an instance variable for later use by the **mousePressed( )** method.
- When the mouse is pressed, it invokes the **showStatus( )** method of the applet through the stored applet reference.
- In other words, **showStatus( )** is invoked relative to the applet reference stored by **MyMouseAdapter**.

10.5.7. Inner classes in action

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

10.5.7. Inner classes in action

- The following listing shows how the preceding program can be improved by using an inner class. Here, **InnerClassDemo** is a top-level class that extends Applet.
- MyMouseAdapter is an inner class that extends MouseAdapter.
- Because MyMouseAdapter is defined within the scope of InnerClassDemo, it has access to all of the variables and methods within the scope of that class.
- Therefore, the mousePressed( ) method can call the showStatus( ) method directly.
- It no longer needs to do this via a stored reference to the applet.
- Thus, it is no longer necessary to pass MyMouseAdapter( ) a reference to the invoking object.

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

10.5.7. Inner classes in action

Anonymous Inner Classes

- An anonymous inner class is one that is not assigned a name.
- Consider the applet shown in the following listing.
- As before, its goal is to display the string "Mouse Pressed" in the status bar of the applet viewer or browser when the mouse is pressed.

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

10.5.7. Inner classes in action

Anonymous Inner Classes

- There is one top-level class in this program: **AnonymousInnerClassDemo**.
- The **init( )** method calls the **addMouseListener( )** method.
- Its argument is an expression that defines and instantiates an anonymous inner class.
- Let's analyze this expression carefully.
- The syntax **new MouseAdapter(){...}** indicates to the compiler that the code between the braces defines an anonymous inner class.
- Furthermore, that class extends **MouseAdapter**.
- This new class is not named, but it is automatically instantiated when this expression is executed.
- Because this anonymous inner class is defined within the scope of **AnonymousInnerClassDemo**, it has access to all of the variables and methods within the scope of that class.
- Therefore, it can call the **showStatus( )** method directly.
- As just illustrated, both named and anonymous inner classes solve some annoying problems in a simple yet effective way.
- They also allow you to create more efficient code.