Chapter 1

Event handling

1.1 Introduction

An event is any happening or occurring. Event-handling code is the heart of every useful application. All event-driven programs are structured around their event-processing model. Java events are a part of the Java Abstract Windowing Toolkit (AWT) package.
As explained in next chapter, applets are event-driven programs. Thus, event handling is at the core of successful applet programming.

An applet is basically an event-driven program in which events are generated by mouse, keyboard and window or other graphical user interface components. The events are passed to the events methods and there are specific methods for recognizing and handling the events.

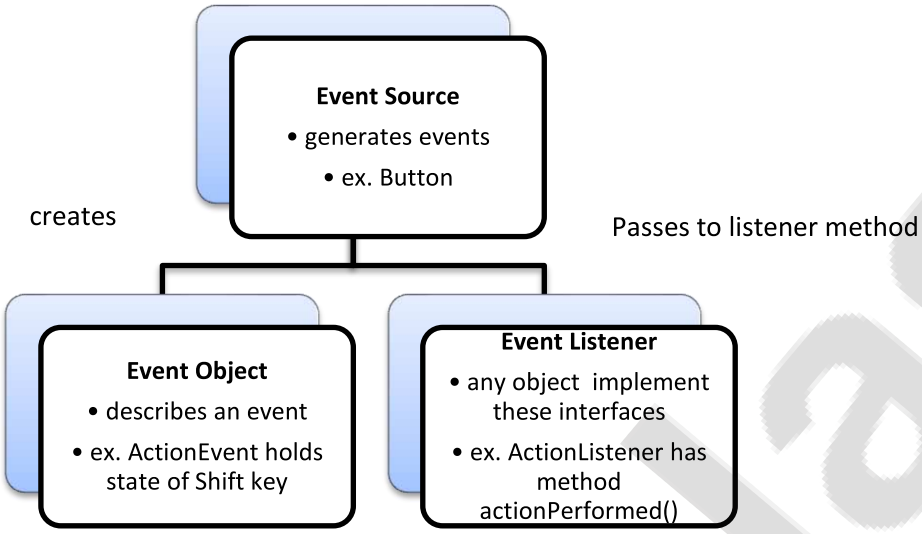
Events are supported by the **java.awt.event** package.

There are two main models for handling events in Java: the old model, called the Inheritance Event Model, is obsolete; the new model, called the Delegation Event Model, will be discussed in detail.

1.2 The Delegation event model

The delegation event model derives its name from the fact that event handling is delegated from an event source to one or more event listeners.

Event Model is based on the concept of an "Event Source" and "Event Listeners". Any object that is interested in receiving messages (or events) is called an Event Listener, and any object that generates these messages (or events) is called an Event Source.



1.3 Events, Event Sources, Event Listeners

An **event** is propagated from a "Source" object to a "Listener" object by invoking a method on the listener and passing in the instance of the event subclass which defines the event type generated. A Listener is an object that implements a specific EventListener interface that extends from the generic `java.util. EventListener`.

An **EventListener** interface defines one or more methods that are to be invoked by the event source in response to each specific event type handled by the interface.

An **Event Source** is an object that originates or "fires" events. A source is an object that generates an event. This occurs when object changes in some way. Source 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.

General form:

```
public void addTypeListener (TypeListener el)
```

When an event occurs, all registered listeners are notified and receive a copy of the event object.

1.4 Event classes

Event Model defines a large number of event classes. At the root of the Java event class hierarchy is `java.util.EventObject`. Every event is a subclass of `java.util.EventObject`. It is a very general class with only one method of interest.

```
Object getSource()
```

This method returns the object that originated the event. Every event has a source object, from which the event originated. This method returns a reference to that source.

`java.awt.AWTEvent`: AWT events, which is the main concern here, are subclasses of java. `awt.AWTEvent`. This is the super class of all the delegation model event classes. The most interesting method in this class is:

```
int getID()
```

This method returns the ID of the event. An event's ID is an int that specifies the exact nature of the event. This value is used to distinguish the various types that are represented by any event class.

Table enumerates the most important of these event classes and provides a brief description of when they are generated.

Event Class	**Description**
ActionEvent	Generated when a button is pressed, a list is double-clicked, or a menu item is selected.
AdjustmentEvent	Generated when a scroll bar is manipulated.
ComponentEvent	Generated when a component is hidden, moved, resized, or becomes visible.
ContainerEvent	Generated when a component is added to or

	removed from a container.
FocusEvent	Generated when a component gains or loses keyboard focus.
InputEvent	Abstract super class for all component input event classes.
ItemEvent	Generated when a check box or a list item is clicked; also occurs when a choice selection is made or a checkable menu is selected or deselected.
KeyEvent	Generated when input is received from the keyboard.
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.
TextEvent	Generated when the value of a text area or text field is changed.
WindowEvent	Generated when a window os activated, closed, deactivated, deiconified, iconified, opened, or quit.

The most commonly used constructors and methods in each class are described in the following sections.

1.4.1 The ActionEvent Class

An object of this class represents a high-level action event generated by an AWT component. Instead of representing a direct user event, such as a mouse or keyboard event, `ActionEvent` represents some sort of action performed by the user on an AWT component.

The `getID()` method returns the type of action that has occurred. For AWT-generated action events, this type is always `ActionEvent.ACTION_PERFORMED`; custom components can generate action events of other types.

The `getActionCommand()` method returns a String that serves as a kind of name for the action that the event represents. The `Button` and `MenuItem` components have a `setActionCommand()` method that allows the programmer to specify an action command string to be included with any action events generated by those components. It is this value that is returned by the `getActionCommand()` method. When more than one `Button` or other component notifies the same `ActionListener`, you can use `getActionCommand()` to help determine the appropriate response to the event.

`getModifiers()` returns a value that indicates the keyboard modifiers that were in effect when the action event was triggered. Use the various `_MASK` constants, along with the `&` operator, to decode this value.

ActionEvent has these constructors:
ActionEvent(Object *src*, int *type*, String *cmd*)
ActionEvent(Object *src*, int *type*, String *cmd*, int *modifiers*)

Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type*, and its command string is *cmd*.
The command name for the invoking **ActionEvent** object by using the **getActionCommand( )** method, shown here:

```
String getActionCommand( )
```

1.4.2 The AdjustmentEvent class

An event of this type indicates that an adjustment has been made to an `Adjustable` object--usually, this means that the user has interacted with a `Scrollbar` component.

The `getValue()` method returns the new value of the `Adjustable` object. This is usually the most important piece of information stored in the event. `getAdjustable()` returns the `Adjustable` object that was the source of the event. It is a convenient alternative to the inherited `getSource()` method.

The `getID()` method returns the type of an `AdjustmentEvent`. The standard AWT components only generate adjustment events of the type `AdjustmentEvent.ADJUSTMENT_VALUE_CHANGED`.

The `getAdjustmentType()` method returns one of five constants to indicate which type has occurred.

There are several types of adjustments that can be made to an Adjustable object, the constants and their meanings are shown here:

- `UNIT_INCREMENT` indicates that the `Adjustable` value has been incremented by one unit, as in a scroll-line-down operation.
- `UNIT_DECREMENT` indicates the opposite: scroll-line-up.
- `BLOCK_INCREMENT` and `BLOCK_DECREMENT` indicate that the `Adjustable` object has been incremented or decremented by multiple units, as in a scroll-page-down or scroll-page-up operation.
- The `TRACK` constant indicates that the `Adjustable` value has been set to an absolute value unrelated to its previous value, as when the user drags a scrollbar to a new position.

Here is one **AdjustmentEvent** constructor:

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

Here, *src* is a reference to the object that generated this event. The *id* equals `ADJUSTMENT_VALUE_CHANGED`. The type of the event is specified by *type*, and its associated data is *data*.

1.4.3 The ComponentEvent class

An event of this type serves as notification that the source `Component` has been moved, resized, shown, or hidden. Note that this event is a notification only: the AWT handles these `Component` operations internally, and the recipient of the event need take no action itself.

`getComponent()` returns the component that was moved, resized, shown, or hidden. It is simply a convenient alternative to `getSource()`. `getID()` returns one of four `COMPONENT_` constants to indicate what operation was performed on the `Component`.

The constants and their meanings are shown here:

- `COMPONENT_HIDDEN` indicates that the component was hidden.
- `COMPONENT_MOVED` indicates that the component was moved.

- `COMPONENT_RESIZED` indicates that the component was resized.
- `COMPONENT_SHOWN` indicates that the component became visible.

`ComponentEvent` has this constructor:

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

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

1.4.4 The ContainerEvent class

An event of this type serves as notification that the source `Container` has had a child added to it or removed from it. Note that this event is a notification only; the AWT adds or removes the child internally, and the recipient of this event need take no action itself.

`getChild()` returns the child `Component` that was added or removed, and `getContainer()` returns the `Container` to which it was added or from which it was removed. `getContainer()` is simply a convenient alternative to `getSource()`. `getID()` returns the constant `COMPONENT_ADDED` or `COMPONENT_REMOVED` to indicate whether the specified child was added or removed.

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*.

1.4.5 The FocusEvent class

An event of this type indicates that a `Component` has gained or lost focus on a temporary or permanent basis. Use the inherited `getComponent()` method to determine which component has gained or lost focus. Use `getID()` to determine the type of focus event; it returns `FOCUS_GAINED` or `FOCUS_LOST`.

When focus is lost, you can call `isTemporary()` to determine whether it is a temporary loss of focus. Temporary focus loss occurs when the window that contains the component loses focus, for example, or when focus is temporarily diverted to a popup menu or a scrollbar. Similarly, you can also use `isTemporary()` to determine whether focus is being granted to a component on a temporary basis.

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

```
FocusEvent(Component src, int type)

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

Here, *src* is a reference to the component that generated this event. The type of the event is specified by *type*. The argument *temporaryFlag* is set to **true** if the focus event is temporary. Otherwise, it is set to **false**.

1.4.6 The InputEvent class

This abstract class serves as the superclass for the raw user input event types `MouseEvent` and `KeyEvent`. Use the inherited `getComponent()` method to determine in which component the event occurred. Use `getWhen()` to obtain a timestamp for the event. Use `getModifiers()` to determine which keyboard modifier keys or mouse buttons were down when the event occurred. You can decode the `getModifiers()` return value using the various `_MASK` constants defined by this class. The class also defines four convenience methods for determining the state of keyboard modifiers.

As input events are delivered to the appropriate listener objects before they are delivered to the AWT components themselves. If a listener calls the `consume()` method of the event, the event is not passed on to the component. For example, if a listener registered on a `Button` consumes a mouse click, it prevents the button itself from responding to that event. You can use `isConsumed()` to test whether some other listener object has already consumed the event.

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

```
ALT_GRAPH_MASK
ALT_MASK
BUTTON1_MASK
BUTTON2_MASK
BUTTON3_MASK
CTRL_MASK
META_MASK
SHIFT_MASK
```

To test if a modifier was pressed at the time an event is generated, use the `isAltDown( ), isAltGraphDown( ), isControlDown( ), isMetaDown( ), and isShiftDown( )` methods.

1.4.7 The ItemEvent class

An event of this type indicates that an item within an `ItemSelectable` component has had its selection state changed. `getItemSelectable()` is a convenient alternative to `getSource()` that returns the `ItemSelectable` object that originated the event. `getItem()` returns an object that represents the item that was selected or deselected.

`getID()` returns the type of the `ItemEvent`. The standard AWT components always generate item events of type `ITEM_STATE_CHANGED`. The `getStateChange()` method returns the new selection state of the item: it returns one of the constants `SELECTED` or `DESELECTED`. (This value can be misleading for `Checkbox` components that are part of a `CheckboxGroup`. If the user attempts to deselect a selected component, a `DESELECTED` event is delivered, but the `CheckboxGroup` immediately reselects the component to enforce its requirement that at least one `Checkbox` be selected at all times.)

ItemEvent has this constructor:

```
public ItemEvent (ItemSelectable src, int id, Object item,
int stateChange);
```

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 *id*. The specific item that generated the item event is passed in *item*. The current state of that itemis in *stateChange*.

1.4.8 The KeyEvent class

An event of this type indicates that the user has pressed or released a key on the keyboard. Call `getID()` to determine the particular type of key event that has occurred. The constant `KEY_PRESSED` indicates that a key has been pressed, while the constant `KEY_RELEASED` indicates that a key has been released. Not all keystrokes actually correspond to or generate Unicode characters. Modifier keys and function keys, for example, do not correspond to characters. Furthermore, for internationalized input, multiple keystrokes are sometimes required to generate a single character of input. Therefore, `getID()` returns a third constant, `KEY_TYPED`, to indicate a `KeyEvent` that actually contains a character value.

For `KEY_PRESSED` and `KEY_RELEASED` key events, use `getKeyCode()` to obtain the virtual keycode of the key that was pressed or released. `KeyEvent` defines a number of `VK_` constants that represent these virtual keys. For example, `VK_0` through `VK_9` and `VK_A` through `VK_Z` define the ASCII equivalents of the numbers and letters.

Note that not all keys on all keyboards have corresponding constants in the `KeyEvent` class, and not all keyboards can generate all of the virtual keycodes defined by this class. If the key that was pressed or

released corresponds directly to a Unicode character, you can obtain that character by calling `getKeyChar()`. If there is not a corresponding Unicode character, this method returns the constant `CHAR_UNDEFINED`.
The `isActionKey()` method returns `true` if the key that was pressed or released does not have a corresponding character.

For `KEY_TYPED` key events, use `getKeyChar()` to return the Unicode character that was typed. If you call `getKeyCode()` for this type of key event, it returns `VK_UNDEFINED`.

See `InputEvent` for information on inherited methods you can use to obtain the keyboard modifiers that were down during the event and other important methods. Use `getComponent()`, inherited from `ComponentEvent`, to determine over what component the event occurred. The static method `getKeyText()` returns a (possibly localized) textual name for a given keycode. The static method `getKeyModifiersText()` returns a (possibly localized) textual description for a set of modifiers.

`KeyEvent` has methods that allow you to change the keycode, key character, or modifiers of an event. These methods, along with the `consume()` method, allow a `KeyListener` to perform filtering of key events before they are passed to the underlying AWT component.

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)
```

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

1.4.9 The MouseEvent class

An event of this type indicates that the user has moved the mouse or pressed one of the mouse buttons. Call `getID()` to determine the specific type of mouse event that has occurred. This method returns one of the following seven constants, which corresponds to a method in either the `MouseListener` or `MouseMotionListener` interface:

- `MOUSE_PRESSED`
  The user has pressed a mouse button.
- `MOUSE_RELEASED`

The user has released a mouse button.

- `MOUSE_CLICKED`
  The user has pressed and released a mouse button without any intervening mouse drag.
- `MOUSE_DRAGGED`
  The user has moved the mouse while holding a button down.
- `MOUSE_MOVED`
  The user has moved the mouse without holding any buttons down.
- `MOUSE_ENTERED`
  The mouse pointer has entered the component.
- `MOUSE_EXITED`
  The mouse pointer has left the component.

Use `getX()` and `getY()` or `getPoint()` to obtain the coordinates of the mouse event.
Use `translatePoint()` to modify these coordinates by a specified amount.

Use `getModifiers()` and other methods and constants inherited from `InputEvent` to determine the mouse button or keyboard modifiers that were down when the event occurred.
See `InputEvent` for details. Note that mouse button modifiers are not reported for `MOUSE_RELEASED` events, since, technically, the mouse button in question is no longer pressed.

Use `getComponent()`, inherited from `ComponentEvent`, to determine over which component the event occurred. For mouse events of type `MOUSE_CLICKED`, `MOUSE_PRESSED`, or `MOUSE_RELEASED`, call `getClickCount()` to determine how many consecutive clicks have occurred. If you are using popup menus, use `isPopupTrigger()` to test whether the current event represents the standard platform-dependent popup menu trigger event.

MouseEvent is a subclass of **InputEvent**. Here is one of its constructors.
`public MouseEvent(Component src, int type, long when, int modifiers, int x, int y, int clicks, boolean triggersPopup)`

Here, *src* is a reference to the component that generated the event. The type of the event is specified by *type*. The system time at which the mouse event occurred is passed in *when*. The *modifiers* argument indicates which modifiers were pressed when a mouse event occurred. The coordinates of the mouse are passed in *x* and *y*. The click count is passed in *clicks*. The *triggersPopup* flag indicates if this event causes a pop-up menu to appear on this platform.

1.4.9 The TextEvent class

An event of this type indicates that the user has edited the text value that appears in a `TextField`, `TextArea`, or other `TextComponent`. This event is triggered by any change to the

displayed text. Note that this is not the same as the `ActionEvent` sent by the `TextField` object when the user edits the text and strikes the **Return** key.

Use the inherited `getSource()` to determine the object that was the source of this event. You have to cast that object to its `TextComponent` type. Call `getID()` to determine the type of a `TextEvent`. The standard AWT components always generate text events of type `TEXT_VALUE_CHANGED`.

The one constructor for this class is shown here:

```
public TextEvent(Object src, int type)
```

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

1.4.10 The WindowEvent class

An event of this type indicates that an important action has occurred for a `Window` object. Call `getWindow()` to determine the Window object that is the source of this event. Call `getID()` to determine the specific type of event that has occurred. Each of the following seven constants corresponds to one of the methods of the `WindowListener` interface:

- `WINDOW_OPENED`
  Indicates that the window has been created and opened; it is delivered only the first time that a window is opened.
- `WINDOW_CLOSING`
  Indicates that the user has requested that the window be closed through the system menu, through a close button on the window's border, or by invoking a platform-defined keystroke, such as **Alt-F4** in Windows. The application should respond to this event by calling `hide()` or `dispose()` on the `Window` object.
- `WINDOW_CLOSED`
  Delivered after a window is closed by a call to `hide()` or `dispose()`.
- `WINDOW_ICONIFIED`
  Delivered when the user iconifies the window.
- `WINDOW_DEICONIFIED`
  Delivered when the user deiconifies the window.
- `WINDOW_ACTIVATED`
  Delivered when the window is activated--that is, when it is given the keyboard focus and becomes the active window.
- `WINDOW_DEACTIVATED`
  Delivered when the window ceases to be the active window, typically when the user activates some other window.

WindowEvent is a subclass of **ComponentEvent**. It defines constructor:

```
public WindowEvent(Window src, int type)
```

Here, *src* is a reference to the object that generated this event. The type of the event is *type.* adds the next three constructors.

```
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 event occurs. The *fromState* specifies the prior state of the window and *toState* specifies the new state that the window will have when a window state change occurs.

1.5 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. Following Table lists 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.
MouseWheelListener	Defines one method to recognize when the mouse wheel is moved.
TextListener	Defines one method to recognize when a text value changes.
WindowFocusListener	Defines two methods to recognize when a window gains or loses input focus.
WindowListener	Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Table: Commonly Used Event Listener Interfaces

The following
sections examine the specific methods that are contained in each interface.

1.5.1 The ActionListener Interface

This interface defines the method that an object must implement to listen for action events on AWT components. When an `ActionEvent` occurs, an AWT component notifies its registered `ActionListener` objects by invoking their `actionPerformed()` methods.
Its general form is shown here:

```
void actionPerformed(ActionEvent ae)
```

1.5.2 The AdjustmentListener Interface

This interface defines the method that an object must implement to listen for adjustment events on AWT components. When an `AdjustmentEvent` occurs, an AWT component notifies its registered `AdjustmentListener` objects by invoking their `adjustmentValueChanged()` methods.
Its general form is shown here:

```
void adjustmentValueChanged(AdjustmentEvent ae)
```

1.5.3 The ComponentListener Interface

This interface defines the methods that an object must implement to listen for component events on AWT components. When a `ComponentEvent` occurs, an AWT component notifies its registered `ComponentListener` objects by invoking one of their methods.
Their general forms are shown here:

```
void componentResized(ComponentEvent ce)
void componentMoved(ComponentEvent ce)
void componentShown(ComponentEvent ce)
void componentHidden(ComponentEvent ce)
```

1.5.4 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)
```

1.5.5 The FocusListener Interface

This interface defines the methods that an object must implement to listen for focus events on AWT components. When a `FocusEvent` occurs, an AWT component notifies its registered `FocusListener` objects by invoking one of their methods.
Their general forms are shown here:

```
void focusGained(FocusEvent fe)
void focusLost(FocusEvent fe)
```

1.5.6 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)
```

1.5.7 The KeyListener Interface

This interface defines the methods that an object must implement to listen for key events on AWT components. When a `KeyEvent` occurs, an AWT component notifies its registered `KeyListener` objects by invoking one of their methods.
The general forms of these methods are shown here:

```
void keyPressed(KeyEvent ke)
void keyReleased(KeyEvent ke)
void keyTyped(KeyEvent ke)
```

1.5.8 The MouseListener Interface

This interface defines the methods that an object must implement to listen for mouse events on AWT components. When a `MouseEvent` occurs, an AWT component notifies its registered `MouseListener` objects by invoking one of their 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)
```

1.5.9 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 mme)
void mouseMoved(MouseEvent mme)
```

1.5.10 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)
```

1.5.11 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 shown here:

```
void textChanged(TextEvent te)
```

1.5.12 The WindowFocusListener Interface

This interface defines two methods `windowGainedFocus( )` and `windowLostFocus( )`.

These are called when a window gains or losses input focus.
Their general forms are shown here:

```
void windowGainedFocus(WindowEvent wfe)
void windowLostFocus(WindowEvent wfe)
```

1.5.13 The WindowListener Interface

This interface defines the methods that an object must implement to listen for window events on AWT components. When a `WindowEvent` occurs, an AWT component notifies its registered `WindowListener` objects by invoking one of their methods.
The general forms are shown here:

```
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)
```

1.6 Using the delegation event model

Up to Now you have learned the theory behind the delegation event model and have had an overview of its various components and its methods, now we will see it in practice. Applet programming using the delegation event model is quite easy.
The Java involves using listener classes that are effectively "attached" to components to process specific events. This lends itself well for GUI builders to generate event handling code.

Following steps are required to perform:

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.

```
someComponent.addActionListener(instanceOfMyClass);
```

To see how the delegation model works in practice, we will look at examples that handle the two most commonly used event generators: the mouse and keyboard.

1.6.1 Handling Mouse Events

Essentially, we'd use these events to know where the mouse is, either with respect to a GUI component or in terms of *x* and *y* coordinates.

Here's how we'll set up this mouse event handling:

1. Add **implements MouseListener**.
2. Mouse listening is based upon a particular component. For an applet, that's the whole applet. Thus, we only need the call **addMouseListener(this);**.
3. Finally, we need to write code to handle the event. Class MouseEvent has accessors getX() and getY() that get us the coordinates of where the event occurred. We can store these an use them in other methods. (For example, we could store them and use them in paint() to draw something.) This time, though, there are five different mouse event methods we must implement:

    - **mousePressed()**, which is triggered when the mouse is on the component.
    - **mouseClicked()**, which is triggered when the mouse is pressed and released on the component. (mousePressed() is called first.)
    - **mouseReleased()**, which is triggered when the mouse is released on the component. (mousePressed() must have been called first.)
    - **mouseEntered()**, which is triggered upon entry to the component.
    - **mouseExited()**, which is triggered upon leaving the component.

Note that when we implement the MouseListener interface, we *must* provide all five methods. The full headers look like this:

```
public void mousePressed(MouseEvent e)
public void mouseClicked(MouseEvent e)
public void mouseReleased(MouseEvent e)
public void mouseEntered(MouseEvent e)
public void mouseExited(MouseEvent e)
```

One workaround to not needing all five methods is to provide empty braces for those we don't need.

Here is a code to demonstrate the mouse events:

```
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);
   }
   // Handle mouse clicked.
   public void mouseClicked(MouseEvent me)
   {
      // save coordinates
      mouseX = me.getX();
      mouseY = me.getY();
      msg = "Mouse clicked.";
      repaint();
   }
   // Handle mouse entered.
```

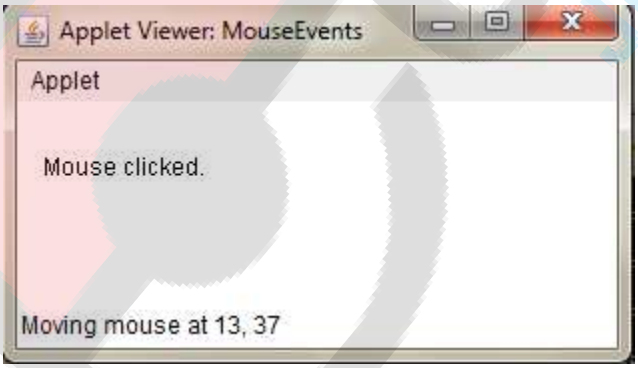
```
public void mouseEntered(MouseEvent me)
{
   // save coordinates
   mouseX = 0;
   mouseY = 10;
   msg = "Mouse entered.";
   repaint();
}
// Handle mouse exited.
public void mouseExited(MouseEvent me)
{
   // save coordinates
   mouseX = 0;
   mouseY = 10;
   msg = "Mouse exited.";
   repaint();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

```
    repaint();
}
// Handle button released.
public void mouseReleased(MouseEvent me)
{
    // save coordinates
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "Up";
    repaint();
}
// Handle mouse dragged.
public void mouseDragged(MouseEvent me)
{
    // save coordinates
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "Mouse Dragged";
    showStatus("Dragging mouse at " + mouseX + ", " + mouseY);
    repaint();
}
// Handle mouse moved.
public void mouseMoved(MouseEvent me)
{
```

```
    // show status
    mouseX = me.getX();
    mouseY = me.getY();
    msg="Mouse Moved";
    showStatus("Moving mouse at " + me.getX() + ", " + me.getY());
    repaint();
  }
  // Display msg in applet window at current X,Y location.
  public void paint(Graphics g)
  {
    g.drawString(msg, mouseX, mouseY);
  }
}
```

Run applet program as: appletviewer MouseEvents.java

Sample output is shown here:



1.7 Adapter classes

Some of the listener interface, such as MouseListener and ComponentListener, include a lot of methods. The rules for interfaces say that when a class implements an interface, it must include a definition for each method declared in the interface. For example, if empty is included only using the `mousePressed` method of the MouseListener interface, one ends up including empty definitions for `mouseClicked ()`, `mouseReleased()`, `mouseEntered()` and `mouseExited`(). If a specially created nested class is used to handle events, there is a way to avoid this. As a convenience, the package java.awt.event includes several adapter classes, such as `MouseAdapter` and `ComponentAdapter`. An adapter class is a class that provides an empty implementation of all methods in an event listener interface. Adapter classes can be used when one is not interested in all of the methods of the interface, except one or two. The `MouseAdapter` class is a trivial class that implements the `MouseListener` interface by defining each of the methods in that interface to be empty. To make the programmer's mouse listener classes, one can extend the `MouseAdapter` class and override just those methods of interest. `ComponentAdapter` and other adapter classes work in the same way.

Since many of the EventListener interfaces are designed to listen to multiple event subtypes (i.e. the MouseListener listens to mouse-down, mouse-up, mouse-enter, etc.), the AWT will provide a a set of abstract "adapter" classes, one which implements each listener interface. This will allow programs to easily subclass the Adapters and override ONLY the methods representing event types they are interested in.

The Adapter classes provided by AWT are as follows:

```
java.awt.event.ComponentAdapter
java.awt.event.ContainerAdapter
java.awt.event.FocusAdapter
java.awt.event.KeyAdapter
java.awt.event.MouseAdapter
java.awt.event.MouseMotionAdapter
java.awt.event.WindowAdapter
```

Note: There are no default Adapters provided for the semantic listeners, since each of those only contain a single method and an adapter would provide no real value.

Following Table lists the commonly used adapter classes in **java.awt.event** and notes the interface that each implements.

Adapter Class	Listener Interface
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener

KeyAdapter	KeyListener
MouseAdapter	MouseListener

The following Code demonstrate adapter class:

```
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;
```

```
}

// Handle mouse clicked

public void mouseClicked (MouseEvent me) {

adapterDemo.showStatus ("Mouse Clicked");

}

}

class MyMouseMotionAdapter extends MouseMotionAdapter {

AdapterDemo adapterDemo;

public MyMouseMotionAdapter (AdapterDemo adapterDemo) {

this. adapterDemo = adapterDemo;

}

//          Handle Mouse Drag

public void mouseDragged (MouseEvent me) {

adapterDemo.showStatus("Mouse Dragged");

}

}
```

Run applet program as: appletviewer AdapterDemo.java

Sample output is shown here:

1.8 Inner classes

The idea behind this was that certain names in the program need to be known only in the vicinity where they are defined. From the beginning, Java adopted this idea of limited scope for variables and methods, but not for packages.

In Java 1.0, all classes were top level ; all classes were defined at the same level. If a class was known anywhere in a source file, it was known throughout the file.

Java 1.1 allows classes to be nested within other classes. A class defined inside another class is known as an inner class. Inner classes are useful for at least two reasons:

- The name of the inner class is known only within its scope. Thus, it does not “pollute” the namespace of the package.

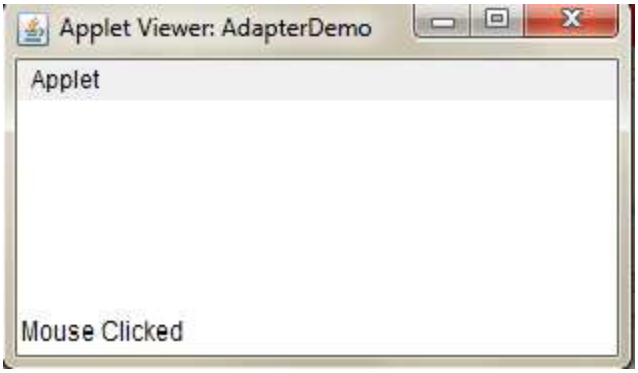
- The code of an inner class can refer directly to names from enclosing scopes, including both class and instance variables and methods, and local variables of enclosing blocks.

The following code demonstrate the normal case:

```
import java.awt.*;

import java.awt.event.*;

import java.applet.*;

/*

<applet code="MousePressedDemo" width=300 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;

}

// Handle mouse clicked

public void mouseClicked (MouseEvent me)
{

mousePressedDemo.showStatus ("Mouse Clicked");

}

}
```

Output:



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**.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="InnerClassDemo" width=300 height=100>
</applet>
*/
public class InnerClassDemo extends Applet
{
public void init ()
{
addMouseListener(new MyMouseAdapter ());
}
class MyMouseAdapter extends MouseAdapter
{
public void mouseClicked (MouseEvent me)
{
showStatus ("Mouse Clicked");
}
}
}
```

Output: