# Event Handling

- Event handling is fundamental to Java programming because it is used to create event driven programs eg

- Applets
- GUI based windows application
- Web Application

- Event handling mechanism have been changed significantly between the original version of Java (1.0) and all subsequent versions of Java, beginning with version 1.1.

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

# Event, Event Source, Event Listener

## What is an Event?

Change in the state of an object is known as event i.e. event describes the change in state of source. Events are generated as result of user interaction with the graphical user interface components. For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page are the activities that causes an event to happen.

# Types of Event

The events can be broadly classified into two categories:

**Foreground Events** - Those events which require the direct interaction of user.They are generated as consequences of a person interacting with the graphical components in Graphical User Interface. For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page etc.

**Background Events** - Those events that require the interaction of end user are known as background events. Operating system interrupts, hardware or software failure, timer expires, an operation completion are the example of background events.

# What is Event Handling?

Event Handling is the mechanism that controls the event and decides what should happen if an event occurs. This mechanism have the code which is known as event handler that is executed when an event occurs. Java Uses the Delegation Event Model to handle the events. This model defines the standard mechanism to generate and handle the events. Let's have a brief introduction to this model.

The Delegation Event Model has the following key participants namely:

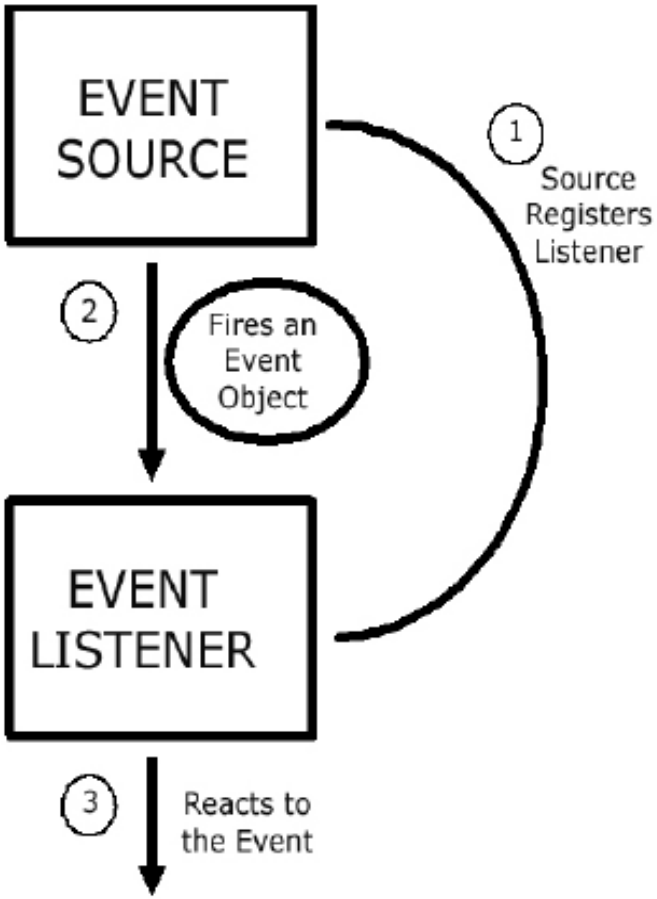
**Source** - The source is an object on which event occurs. Source is responsible for providing information of the occurred event to it's handler. Java provides  classes for source object.

**Listener** - It is also known as event handler. Listener is responsible for generating response to an event. From java implementation point of view the listener is also an object. Listener waits until it receives an event. Once the event is received , the listener process the event an then returns.

# Advantages of event Handling

The benefit of this approach is that the user interface logic is completely separated from the logic that generates the event. The user interface element is able to delegate the processing of an event to the separate piece of code. In this model ,Listener needs to be registered with the source object so that the listener can receive the event notification. This is an efficient way of handling the event because the event notifications are sent only to those listener that want to receive them.

# Delegation Event Model



## Delegation Event Model

Writing event driven program is a two step process-

- **Implement the appropriate interface in the listener** so that it can receive the type of event desired.
- **Implement code to register and unregister (if necessary) the listener** as a recipient for the event notifications.

# Example of Handling Mouse Events

To handle mouse events, we must implement the MouseListener and the MouseMotionListener interfaces.

- **Objective**: To write an applet that 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.
- 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.

# Tools to write this program

- Tool to create Listener class
- Tool to register this Listener to Event Source

EventListener interface is provided in java.util package and is responsible for handling events.

**public interface EventListener;**

***AWT Event Listener Interfaces**: java.awt.event package*

*ActionListener, KeyListner, MouseListener, TextListener, MouseMotionListner are few commonly used Listners.*

***public interface ActionListener extends EventListner {***

***Void actionPerformed(ActionEvent e);***

***}***

***Component Class: java.awt.Component package***

***void addMosueListener(MouseListener ml)***

# Events and Event Clases

The root class is called java.util.EventObject. The only common feature shared by all events is a source object. So we find the following two methods in the EventObject class :

public Object getSource();

Returns the source of the event.

String toString( );

returns the string equivalent of the event.

# Continued..

- 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.
- int getID( );
- The package java.awt.event defines many types of events that are generated by various user interface elements.

# Commonly used Event Classes in java.awt.event

| 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. |

# Continued..

| | |
|---|---|
| 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, |

# Example

- 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 with the mouse pointer as it is dragged.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="MouseEvents" width=300
height=100>
</applet>
*/
```

# Continued..

```
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);
}
```

# Source Code

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

public void mouseEntered(MouseEvent me) {
// save coordinates
mouseX = 0;
mouseY = 10;
msg = "Mouse entered.";
repaint();
}
```

# Continued..

```
public void mouseExited(MouseEvent me) {
// save coordinates
mouseX = 0;
mouseY = 10;
msg = "Mouse exited.";
repaint();
}

public void mousePressed(MouseEvent me) {
// save coordinates
mouseX = me.getX();
mouseY = me.getY();
msg = "Down";
repaint();
}
```

# Continued..

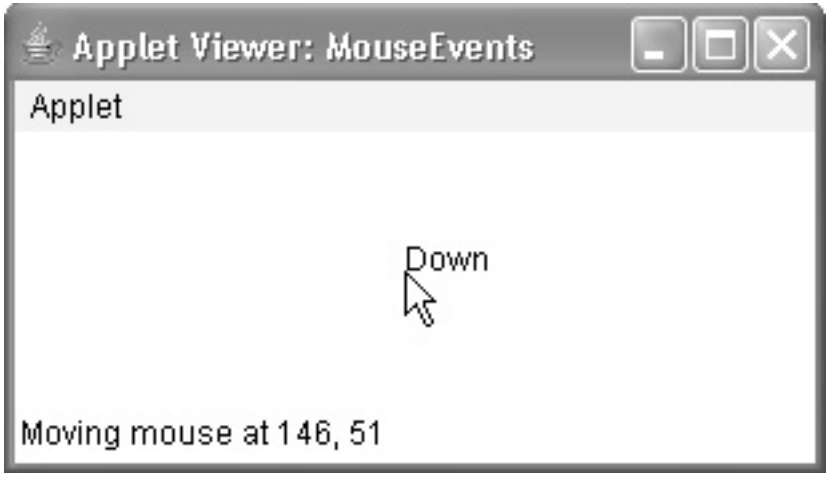
```
public void mouseReleased(MouseEvent me) {
mouseX = me.getX();
mouseY = me.getY();
msg = "Up";
repaint();
}
public void mouseDragged(MouseEvent me) {
mouseX = me.getX();
mouseY = me.getY();
msg = "*";
showStatus("Dragging mouse at " + mouseX + ", " + mouseY);
repaint();
}
```

# Continued..

```
public void mouseMoved(MouseEvent me) {
mouseX = me.getX();
mouseY = me.getY();
showStatus("Moving mouse at " + mouseX + ", " +
mouseY);
}
// Display msg in applet window at current X,Y location.

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

# Sample Output



# Important Points to remember

- 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
- **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.

# Continued..

- Inside init( ), the applet registers itself as a listener for mouse events. This is done by using addMouseListener( ) & addMouseMotionListener( ).

- The applet then implements all of the methods of the MouseListener and MouseMotionListener interfaces.

- These are the event handlers for the various mouse events. Each method handles its event and then returns.

## Simplifying previous program using inner class

- An inner class is a class which is defined in another class.
- In this program a class MyHandler is designed which implements both MouseListener and MouseMotionListner interfaces.
- So now applet works just as source and not as Listener.
- MyHandler class need to be registered to applet through both addMouseListener() and addMouseMotionListener().

# Simplified program

```
import java.applet.Applet;
import java.awt.*;
 import java.awt.event.*;

<applet code="MouseEvents1" width=300 height=100>
</applet>

public class MouseEvents1 extends Applet {
String msg = "";
int mouseX = 0, mouseY = 0; // coordinates of mouse
public void init() {
addMouseListener (new MyHandler());
addMouseMotionListener (new MyHandler());
}
```

# Continued..

```
// Display msg in applet window at current X,Y location.
public void paint(Graphics g) {
g.drawString(msg, mouseX, mouseY);
}
class MyHandler implements MouseListener,
MouseMotionListener {
public void mouseClicked(MouseEvent me) {
// save coordinates
mouseX = 0;
mouseY = 10;
msg = "Mouse clicked.";
repaint();
}
```

# Continued..

```
public void mouseEntered(MouseEvent me) {
// save coordinates
mouseX = 0;
mouseY = 10;
msg = "Mouse entered.";
repaint();
}

public void mouseExited(MouseEvent me) {
// save coordinates
mouseX = 0;
mouseY = 10;
msg = "Mouse exited.";
repaint();
}
```

# Continued..

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

# Continued..

```
public void mouseDragged(MouseEvent me) {
mouseX = me.getX();
mouseY = me.getY();
msg = "*";
showStatus("Dragging mouse at " + mouseX + ", " + mouseY);
repaint();
}
public void mouseMoved(MouseEvent me) {
mouseX = me.getX();
mouseY = me.getY();
showStatus("Moving mouse at " + mouseX + ", " + mouseY);
}
}  //MyHandler class
}  // MouseEvents1 class
```

# Handling Keyboard Events

**Objective**: To echo keystrokes to the applet window and shows the pressed/released status of each key in the status window

Which listener interface needs to be implemented by Applet?

KeyListener Interface

What are the methods defined by KeyListener Interface?

keyPressed(KeyEvent e)

keyReleased(KeyEvent e)

keyTyped(KeyEvent e)

# Source code

```
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);
}
```

**Continued..**

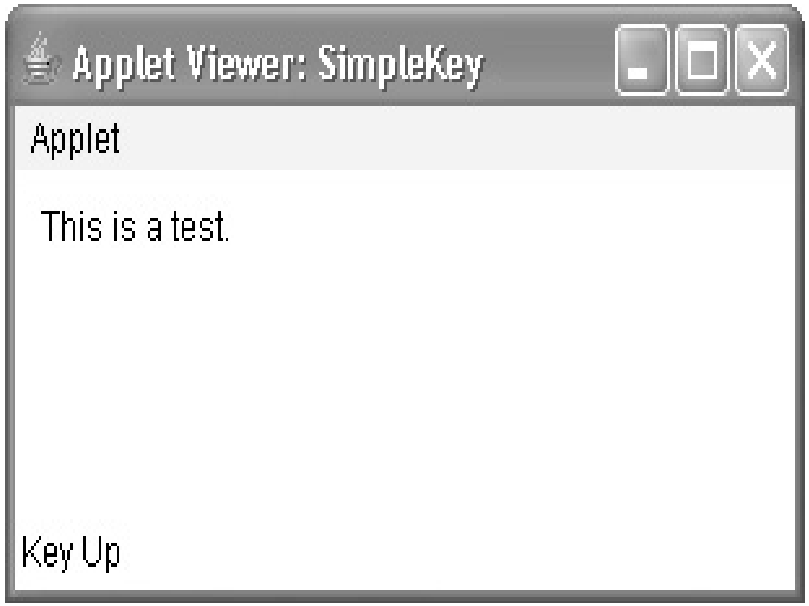
```
public void keyPressed(KeyEvent ke) {
showStatus("Key Down");
}
public void keyReleased(KeyEvent ke) {
showStatus("Key Up");
}
public void keyTyped(KeyEvent ke) {
msg += ke.getKeyChar();
repaint();
}
```

# Continued..

```
// Display keystrokes.
public void paint(Graphics g)
{
g.drawString(msg, X, Y);
}
}
```

Question: What if you wish to know if a special key like function key or arrow key is pressed?

**It is to be handled in keyPressed method.**



# To handle special keys

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( ).

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="KeyEvents" width=300 height=100>
</applet>
*/
```

# Continued..

```
public class KeyEvents extends Applet
implements KeyListener {
String msg = "";
int X = 10, Y = 20; // output coordinates
public void init() {
addKeyListener(this);
}
```

# Continued..

```
public void keyPressed(KeyEvent ke) {
showStatus("Key Down");
int key = ke.getKeyCode();
switch(key) {
case KeyEvent.VK_F1:
msg += "<F1>";
break;
case KeyEvent.VK_F2:
msg += "<F2>";
break;
```

## Continued..

```
case KeyEvent.VK_F3:
msg += "<F3>";
break;
case KeyEvent.VK_PAGE_DOWN:
msg += "<PgDn>";
break;
case KeyEvent.VK_PAGE_UP:
msg += "<PgUp>";
break;
```

## Continued..

```
case KeyEvent.VK_LEFT:
msg += "<Left Arrow>";
break;
case KeyEvent.VK_RIGHT:
msg += "<Right Arrow>";
break;
}
repaint();
}
```

# Continued..

```
public void keyReleased(KeyEvent ke) {
showStatus("Key Up");
}
public void keyTyped(KeyEvent ke) {
msg += ke.getKeyChar();
repaint();
}
// Display keystrokes.
public void paint(Graphics g) {
g.drawString(msg, X, Y);
}
}
```

# Sample Output

# Program to Add a Button to a Frame

```
import java.awt.*;
import java.awt.event.*;
public class ButtonText {
        public static void main(String[] args) {
                Frame frame=new Frame("Button Frame");
                Button button = new Button("Submit");
                frame.add(button);
                frame.setLayout(new FlowLayout());
                frame.setSize(200,100);
                frame.setVisible(true);
```

# Continued..

```
frame.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent e){System.exit(0);} });
  }
}
```

# Another Example

```
import java.awt.*;
 import java.awt.event.*;
 class AEvent extends Frame implements ActionListener {
TextField tf;
Button b;
AEvent() {
 //create components
  tf=new TextField();
 tf.setBounds(60,50,170,20);
 b=new Button("click me");
b.setBounds(100,120,80,30);
```

# Continued..

```
//register listener
b.addActionListener(this);    //passing current instance
 //add components and set size, layout and visibility
addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent e){
System.exit(0);
                                }
});
add(b);
add(tf);
```

# Continued..

```
setSize(300,300);
setLayout(null);
setVisible(true);
 }
public void actionPerformed(ActionEvent e) {
tf.setText("Welcome" + this);
// this is used to check which kind of object this refers to
 }
public static void main(String args[]){
new AEvent();
}
}
```

# Adapter Classes

- Java provides a special feature, **called an *adapter class***, that can simplify the creation of event handlers.

- **An adapter class provides an empty implementation of all methods in an event listener interface.**

- Eg. the **MouseMotionAdapter** class has two methods, **mouseDragged( )** and **mouseMoved( )**, which are the methods defined by the **MouseMotionListener** interface.

- 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.

# Continued..

- If only mouse drag events is to be used, simply extend **MouseMotionAdapter** and override **mouseDragged( )**.

- The empty implementation of **mouseMoved( )** would handle the mouse motion.

- The adapter classes are available in java.awt.event package.

# List of adapter classes

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

# Example

- **Objective**: To display a message in the status bar of an applet viewer or browser when the mouse is clicked or dragged.
- The program has three classes, **AdapterDemo** extends**Applet**.
- 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.

# Continued..

- **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.

# Continued..

- **Note:** **MouseAdapter** also provides an empty implementation for **MouseMotionListener**. However, for the sake of illustration, this example handles each separately.

- 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.

# Continued..

```
// Demonstrate an adapter.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="AdapterDemo" width=300
height=100>
</applet>
*/
```

# Continued..

```
public class AdapterDemo extends Applet {
public void init() {
// register listener
addMouseListener (new MyMouseAdapter(this));
addMouseMotionListener (new
MyMouseMotionAdapter(this));
}
}
```

Note: Here Applet class does not implements listener class. Implication??

Event Source and Event Listener are different.

# Continued..

```
class MyMouseAdapter extends MouseAdapter {
AdapterDemo adapterDemo;
// constructor
public MyMouseAdapter(AdapterDemo
adapterDemo) {
this.adapterDemo = adapterDemo;
}
// Handle mouse clicked.
public void mouseClicked(MouseEvent me) {
adapterDemo.showStatus ("Mouse clicked");
}
}
```

# Continued..

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

# Inner Classes

Not having to implement all of the methods defined by the MouseMotionListener and MouseListener interfaces saves a considerable amount of effort.

- An inner class is a class defined within another class, or even within an expression.
- Let us see how inner classes can be used to simplify the code when using event adapter classes.

# Use of 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.

# Source Code

```
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 Class

A class that has no name is known as anonymous inner class in java. It should be used if you have to override method of class or interface. Java Anonymous inner class can be created by two ways:

**1. Using Class (may be abstract or concrete).**

**2. Using Interface**

## 1.Java anonymous inner class example using class

```
abstract class Person{
  abstract void eat();
}
```

# Continued..

```
class TestAnonymousInner{
    public static void main(String args[]){
     Person p=new Person(){
     void eat(){System.out.println("nice fruits");}
     };
     p.eat();
    }
   }
```

# Continued..

**Internal Working:**

1. A class is created but its name is decided by the compiler which extends the Person class and provides the implementation of the eat() method.

2. An object of Anonymous class is created that is referred by p reference variable of Person type.

# 2.anonymous inner class: using interface

```
interface Eatable{
     void eat();
    }
    class TestAnnonymousInnerClass{
     public static void main(String args[]){
     Eatable e=new Eatable(){
      public void eat(){System.out.println("nice
fruits");}
     };
     e.eat();
     }
    }
```

# Internal Working

- A class is created but its name is decided by the compiler **which implements the Eatable interface** and provides the implementation of the eat() method.
- An object of Anonymous class is created that is referred by p reference variable of Eatable type

# Anonymous Inner Classes

- Let us see how an anonymous inner class can facilitate the writing of event handlers.

```
// Anonymous inner class demo.
import java.applet.*;
import java.awt.event.*;
Import java.awt.*;

/*  <applet code="AnonymousInnerClassDemo"
width=200 height=100>  </applet>
*/
```

# Continued..

```
public class AnonymousInnerClassDemo extends
Applet {
public void init() {
addMouseListener (new MouseAdapter() {
public void mousePressed(MouseEvent me) {
showStatus("Mouse Pressed");
}
}); // header closed
}
}
```

# Continued..

- The classAnonymousInnerClassDemo extends Applet class.

- The init( ) method calls the addMouseListener( ) method.
- Its argument is an expression that defines

and instantiates an anonymous inner class.

- Analyze this expression carefully. The syntax new MouseAdapter(){...} indicates to the compiler that the code between the braces defines an anonymous inner class..

# Continued.

- 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 Anonymous Inner Class Demo, it has access to all of the variables and methods within the scope of that class. Therefore, it can call the showStatus( ) method directly.
- Inner and Anonymous inner classes simplify event Handling

## Continued..

- 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.
- 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.

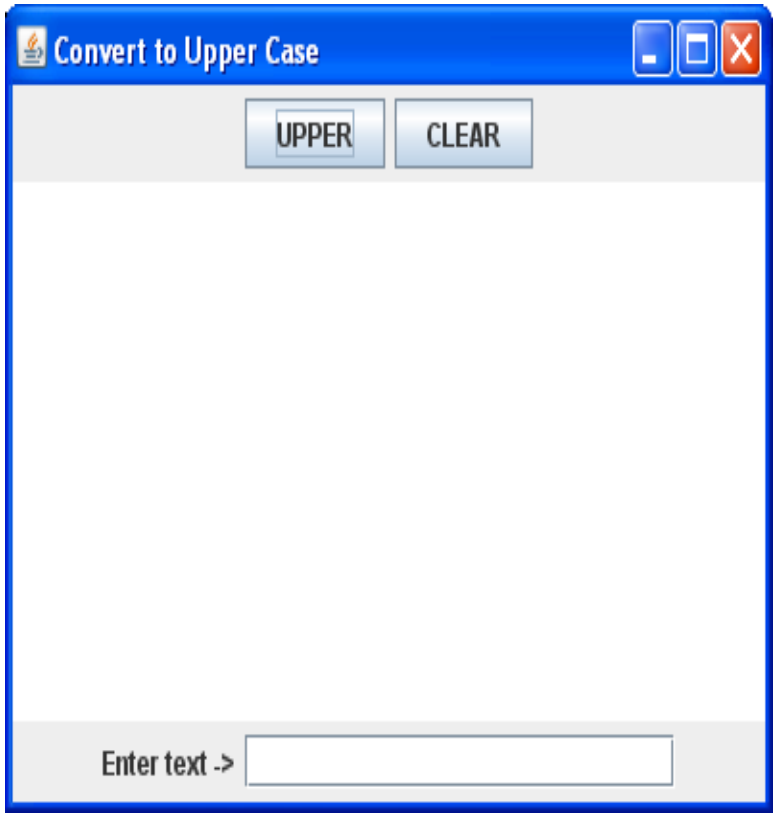
## Continued..

- In the delegation event model, listeners must register with a source in order to receive an event notification.
- **benefit**: notifications are sent only to listeners that want to receive them.
- In original Java 1.0 approach, 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.

# Three players of Delegation Event Model

- Event source which generates the event object
- Event listener which receives the event object and handles it
- Event object that describes the event.

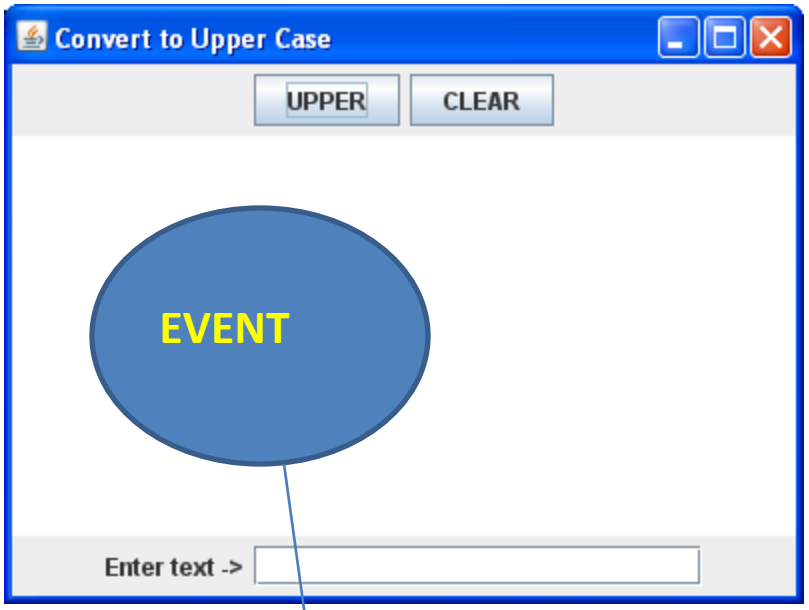
# Continued..




- An already created GUI.
- How many components?
- What are some possible events?
- Two Button components, a TextArea component and a Textfield component.
- MouseClick, KeyPress etc

# Example

- Click on UPPER *Button*
- Generates an *ActionEvent*
- Event object is sent to an *ActionListener* that is registered with the UPPER *Button*
- *ActionListener* handles in *actionPerformed* method.



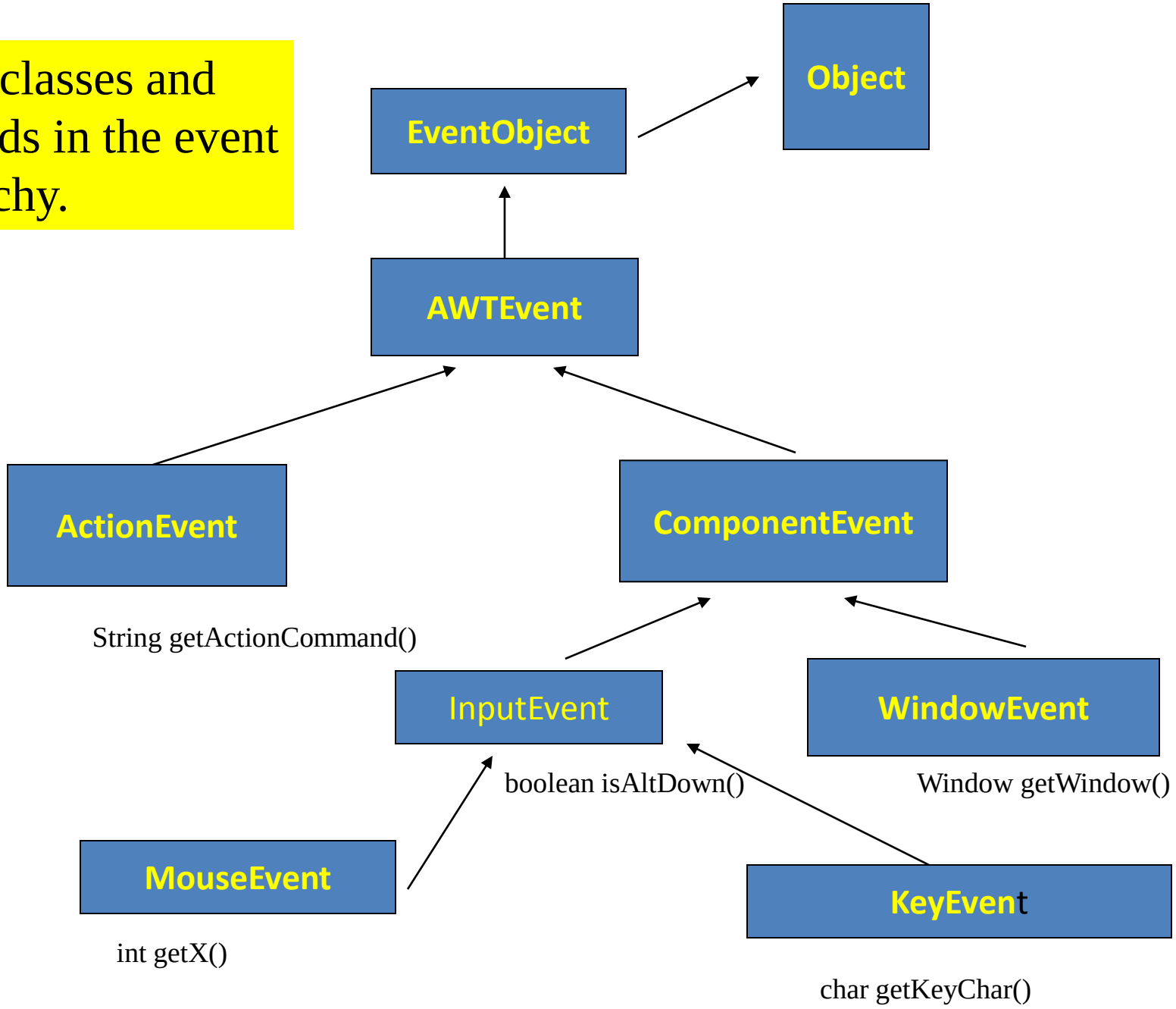
# Events

- An event is an object that describes a state change in a source.
- 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.
- Eg. an event may be generated when a timer expires, a counter exceeds a value etc.
- You can define your own application specific events.

## Events and Event Objects

- Since events may be of different types but still exhibit some shared traits (inheritance) Java represents the event classes in a hierarchy.

Some classes and methods in the event hierarchy.



**Event handling usually involves three types of objects**

- Objects that are used to hold and report information about the event.
- Objects that are typically the source of events.
- Objects, called listeners, are used to handle events.

# Example

A mouse object

Event Source

An event object that describes, say, the x and y coordinate of where the mouse was clicked

Event data and methods

Event Listener

The listener object has methods that are called for particular events

# Example

A mouse object

An event object that describes, say, the x and y coordinate of where the mouse was clicked

Implements MouseListener

A mouse object must be told who its listener is.

The listener object has methods that are called for particular events

The event object is sent to a listener method

## 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.
- Some general Event Sources are: Button, CheckBox,List,MenusItem,Window,TextItems Etc…

# Continued..

| 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 | 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 |

# Continued..

- In addition to these graphical user interface elements, any class derived from Component, such as Applet, can generate events.
- For example, you can generate key and mouse events from an applet.
- Similarly generated events can be received by applets.
- **In applets, an applet can be both source and listener.**
- You may also build your own components that generate events.

# 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 package.
- .

# Listeners are Interfaces

Remember what an interface provides? If class X implements an interface then class X promises to provide (at least) the methods declared in the interface.

Eg. the MouseMotionListener interface, that defines two methods to receive notifications when the mouse is dragged or moved

# Listener Interfaces

| 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. |

# Continued..

| | |
|---|---|
| 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. |

# Continued..

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.

# How to Attach an Event Listener to an Event Source?

o is an event source

h is an event listener of type XXX

o.addXXX(h)

where XXX is one of the following:

- ActionListener
- MouseListener
- MouseMotionListener
- KeyListener
- WindowListener
- ComponentListener
- FocusListener
- TextListener
- AdjustmentListener
- ItemListener

# Continued..

- public void addTypeListener (TypeListener el )

Type is the name of the event, and el is a reference to the event listener.

- When an event occurs, all registered listeners are notified and receive a copy of the event object. This is known **as multicasting the event**

- Some sources may allow only one listener to register. The method to register in this case is

public void addTypeListener(TypeListener el )
throws java.util.TooManyListenersException

- . When such an event occurs, the registered listener is notified. This is known **as unicasting the event.**

# Removing listeners from Source

A source must also provide a method that allows a listener to unregister an interest in a specific type of event.

public void removeTypeListener(TypeListener el )

Here, Type is the name of the event, and el is a reference to the event listener.

To remove a keyboard listener, call removeKeyListener( ).

Methods to add or remove listeners are provided by the source that generates events.

# continued

- 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.

# Inheritance tree of applets & frames