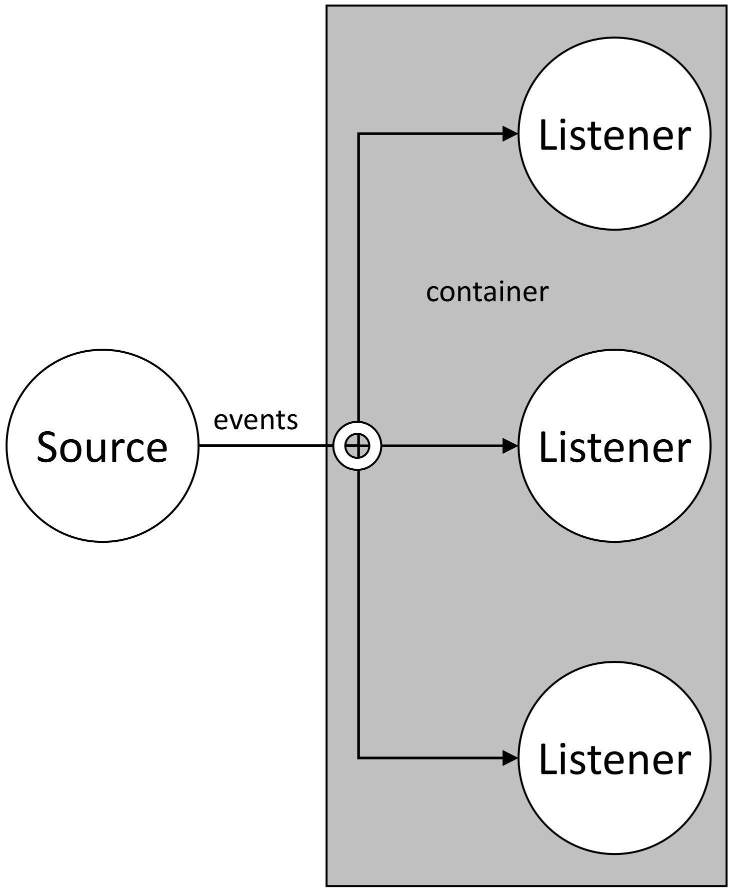
## Event Handling

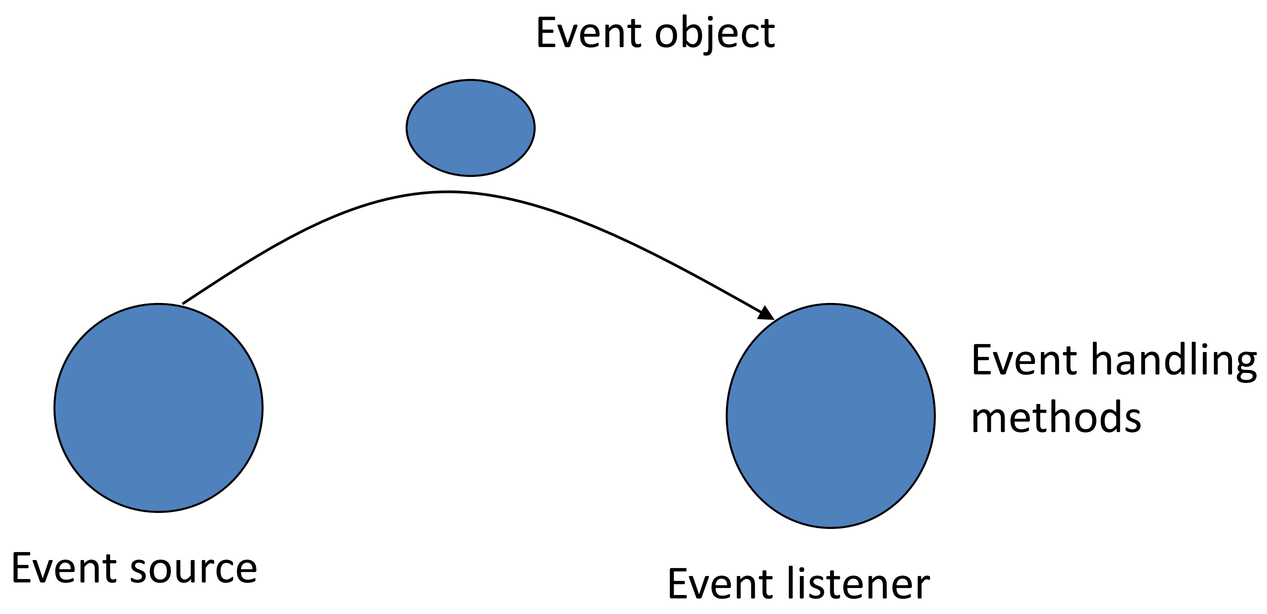
- An event is an object that describes some state change in a source
- Each time a user interacts with a component an event is generated, e.g.:
  - A button is pressed
  - A menu item is selected
  - A window is resized
  - A key is pressed
- An event informs the program about the action that must be performed

# The Delegation Event Model

- Provides a standard mechanism for a **source** to generate an **event** and send it to a set of **listeners**
- A source generates events.
- 3 responsibilities of a source:
  - To provide methods that allow listeners to register and unregister for notifications about a specific type of event
  - To generate the event
  - To send the event to all registered listeners.



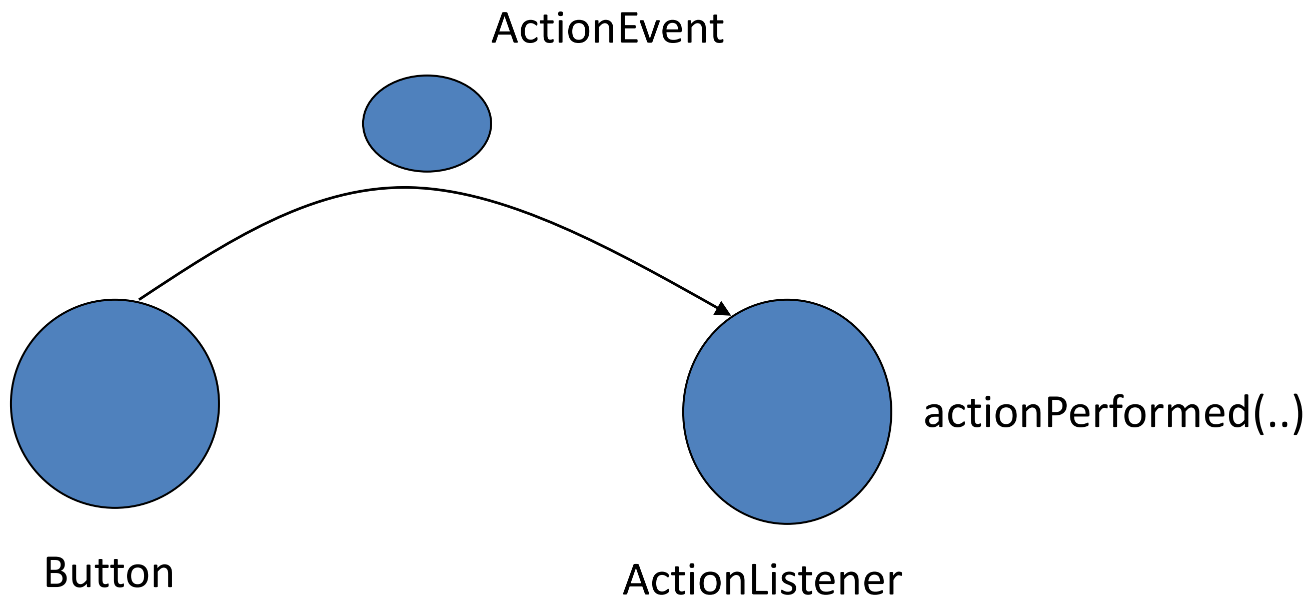
# Event Handling Model of AWT



# Event Classes

- The EventObject class has the two methods
  - The getSource() method returns the object that generated the event
  - toSource() method returns a string equivalent of the event.
- The AWTEvent(Object source, int id)
  - Source is the object that generates the event and id identifies the type of the event.
  - The class has the getID() method that returns the type of the event.
- Event Listener (java.util.EventListener) interface does not define any constraints or methods but exists only to identify those interfaces that process events
- The Component class has the methods that allow a listener to register and unregister for events:
  - void add*Type*Listener(*Type*Listener tl)
  - void remove*Type*Listener(*Type*Listener tl)
  - Eg:- addKeyListener()

# Action Events on  Buttons



## Semantic Event Listener

- The semantic events relate to operations on the components in the GUI. **semantic event** classes.
  - An **ActionEvent** is generated when there was an action performed on a component such as clicking on a menu item or a button.
    - Produced by Objects of Type: Buttons
  - An **ItemEvent** occurs when a component is selected or deselected.
    - Produced by Objects of Type: Menus
  - An **AdjustmentEvent** is produced when an adjustable object, such as a scrollbar, is adjusted.
    - Produced by Objects of Type: Scrollbar
- Semantic Event Listeners
  - Listener Interface: ActionListener, Method: void actionPerformed(ActionEvent e)
  - Listener Interface: ItemListener, Method: void itemStateChanged (ItemEvent e)
  - Listener Interface: AdjustmentListener, Method: void adjustmentValueChanged (AdjustmentEvent e)

# Using the ActionListener

- Stages for Event Handling by ActionListener
  - First, import event class

    import java.awt.event.*;

  - Define an overriding class of event type (implements ActionListener)

```
class ButtonListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
                // Write what to be done. . .
                label.setText("Hello World!");
        }
}
```

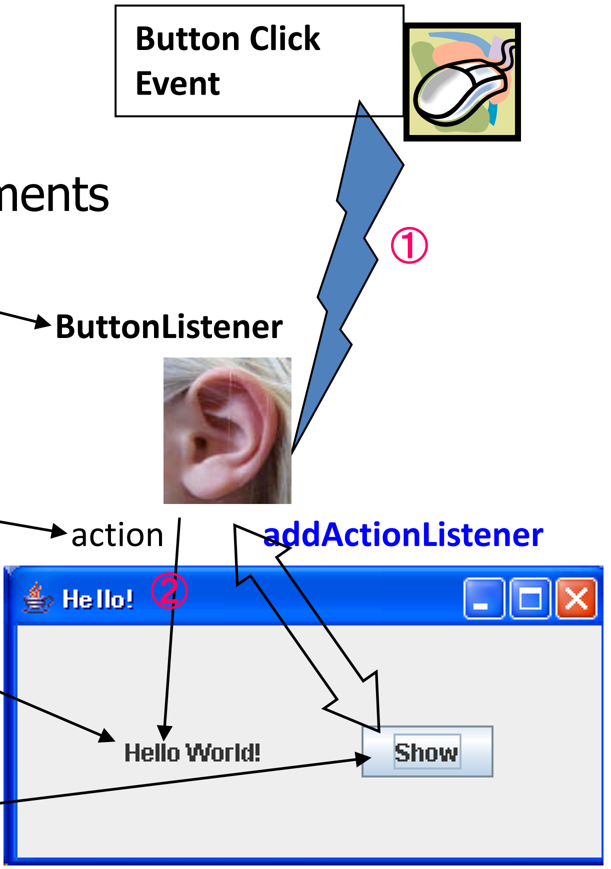
  - Create an event listener object

    ButtonListener bt = new ButtonListener();

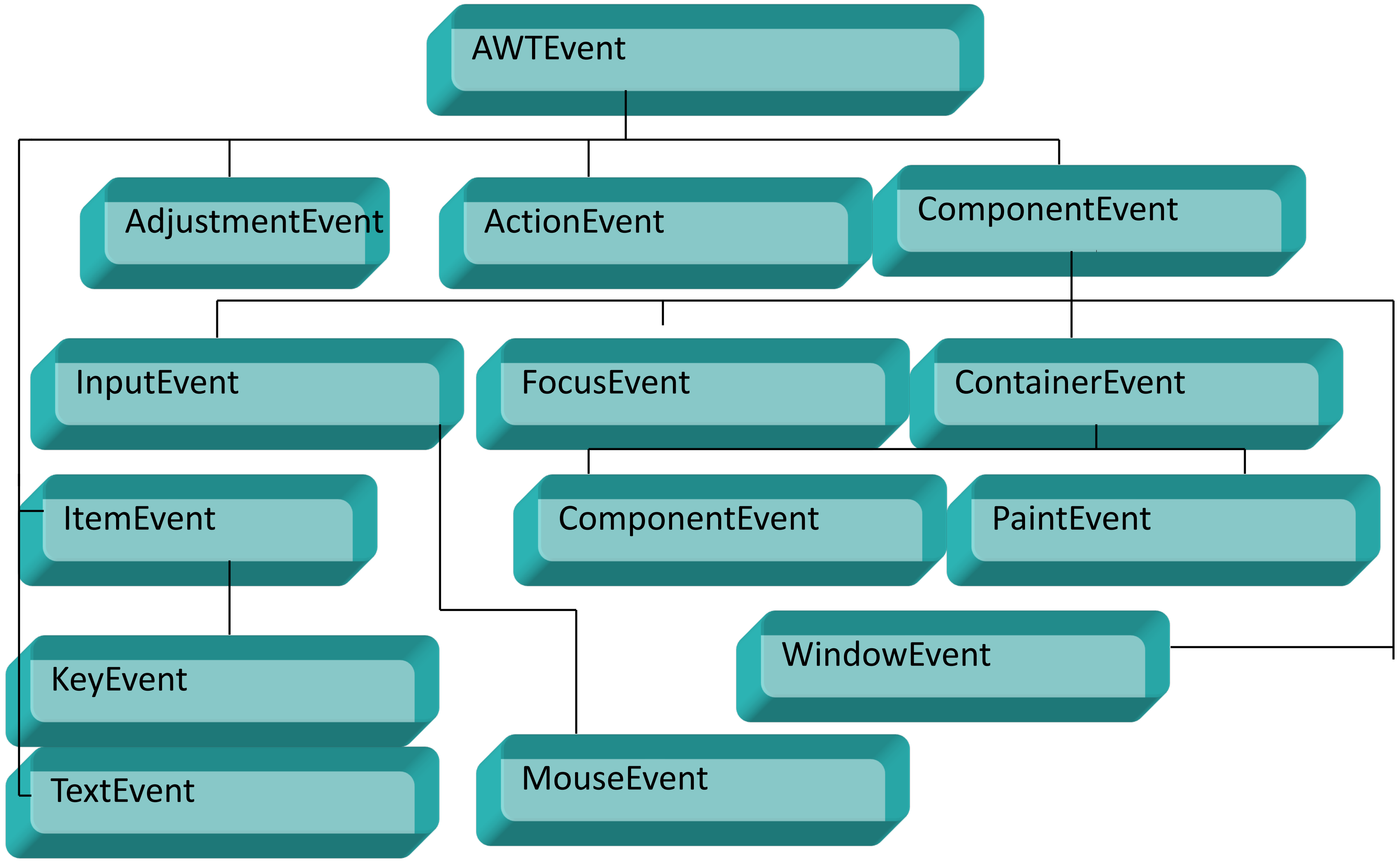
  - Register the event listener object

    b1 = new Button("Show");

    b1.**addActionListener**(bt);

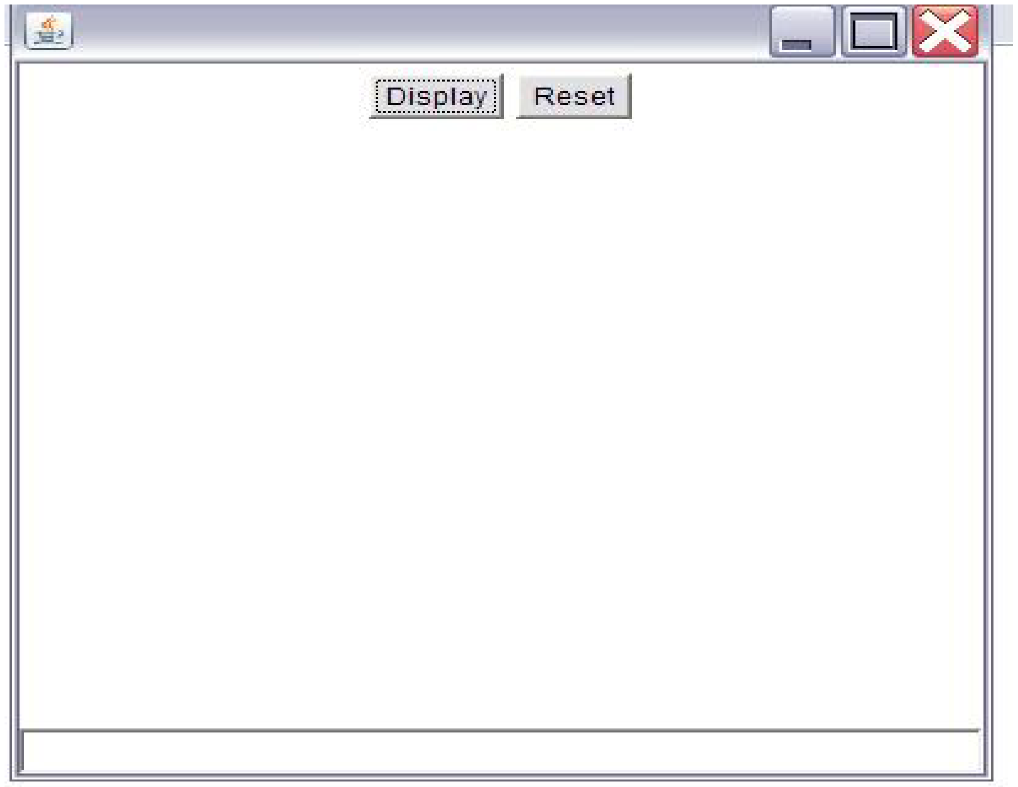


# Types of AWT Events

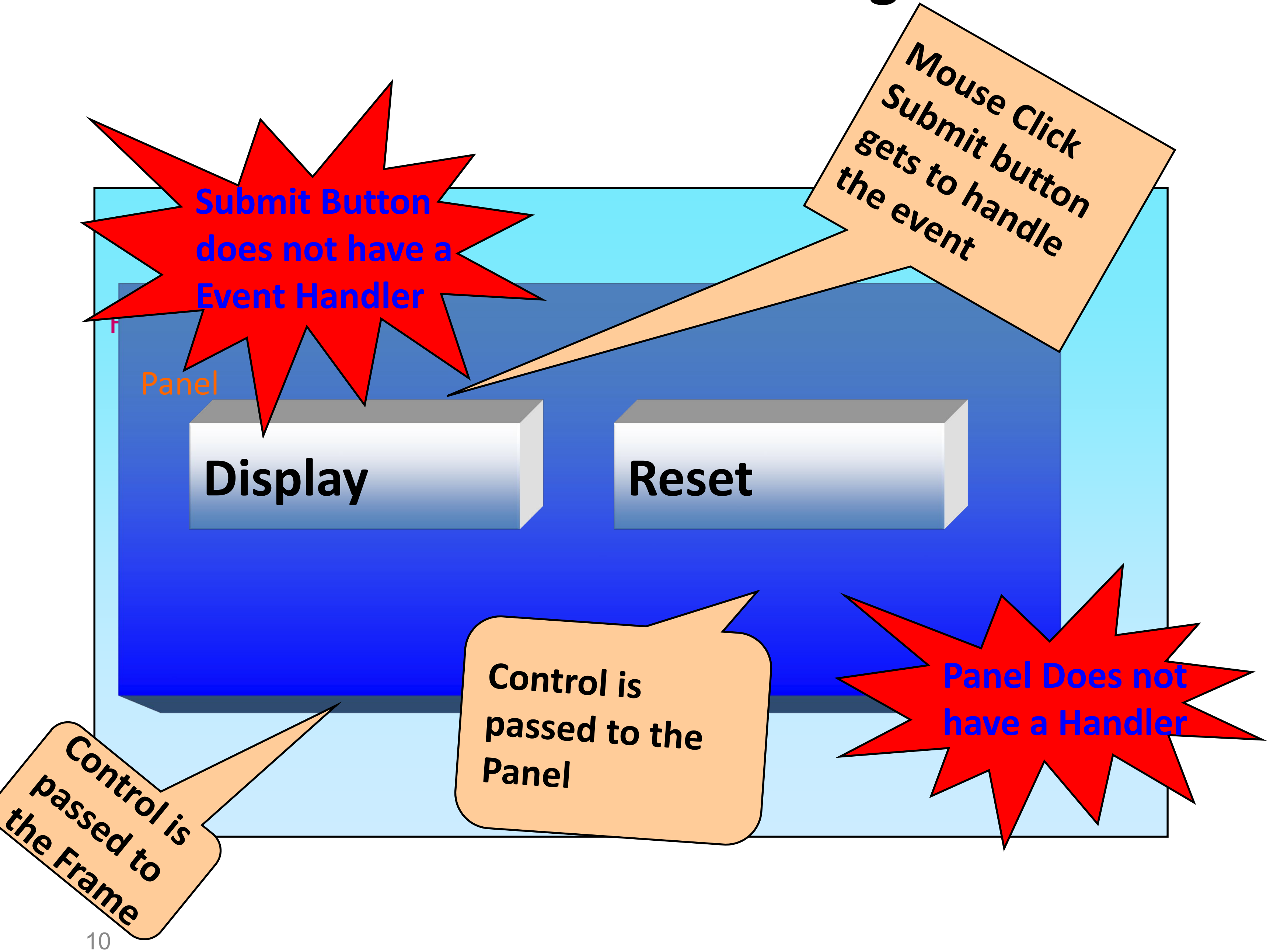


## Event Dispatching

Consider the Following



# Event Handling


Mouse Click Submit button gets to handle the event
Submit Button does not have a Event Handler
Panel
Display
Reset
Control is passed to the Panel
Panel Does not have a Handler
Control is passed to the Frame

## Event Dispatch and Propogation

- When the user clicks into “submit” button
- java language run-time system gathers
- Event Class
- Component

## Event Listener

- An object that would like to be notified of and respond to an event is an *event listener.*
- An object that generates a particular kind of event, called an *event source*, maintains a list of listeners that are interested in being notified when that kind of event occurs.
- When the event source generates an event the event source notifies all the listener objects that the event has occurred.

## Event Listeners

- A Listener must be added to a component to react to the events occurring on the component
- An event is a component’s way of letting the listener know about that something has happened
- A component must have a way to register and deregister listeners
- The components must track its Listeners and pass on the events to those listeners
- Multicasting & Unicasting

## Some Common EventListeners

| Event Listener | Listener methods | Registered On |
|---|---|---|
| **ActionListener** | actionPerformed() | AbstractButton, Button, ButtonModel, ComboBoxEditor, JComboBox, JFileChooser, JTextField, List, MenuItem, TextField, Timer |
| **ItemListener** | itemStateChanged() | CheckBox,Choice etc |
| **MouseListener** | mouseClicked(),mousePressed(), mouseReleased(),mouseEntered(), mouseExited() | Component |
| **TextListener** | textValueChanged() | TextComponent |
| **MouseMotionListener** | mouseDragged(),mouseMoved() | Component |

## Some Common EventListeners

| Event Listener | Listener methods | Registered On |
|---|---|---|
| **WindowListener** | **windowActivated(), windowClosed(), windowClosing(), windowDeactivated(), windowDeiconified(), windowIconified(), windowOpened()** | **Window** |
| **FocusListener** | **focusGained(),focusLost()** | **Component** |
| **KeyListener** | **keyTyped(),keyReleased(),keyPressed()** | **Component** |

## Example

```
import java.awt.*;
import java.awt.event.*;
public class MyApplication extends Frame
   implements ActionListener
{  Button b1,b2;
   TextField t1;
  Panel p1;
  MyApplication()
   {           b1=new Button(“Display”);
               b2=new Button(“Clear”);
               p1=new Panel();
               t1=new TextField(20);
```

## Example contd.

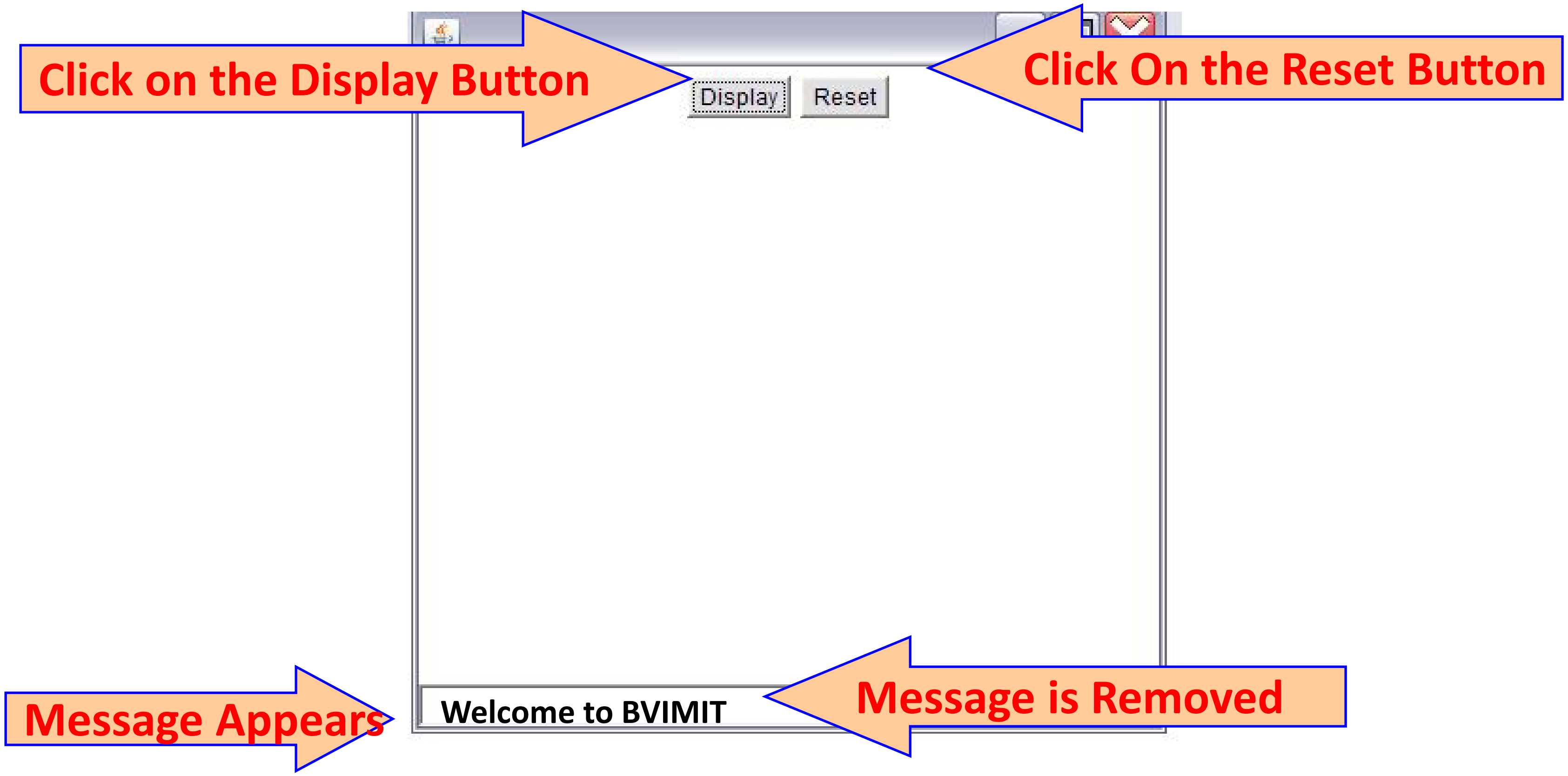
```
            b1.addActionListener(this);
            b2.addActionListener(this);
            p1.add(b1);
            p1.add(b2);
            add(p1,BorderLayout.NORTH);
            add(t1,BorderLayout.SOUTH);
            setSize(400,400);
            setVisible(true);
    }
```

Buttons registered with an ActionListener

## Implementing the Action Listener

```
public void actionPerformed(ActionEvent e)
{       if(e.getSource()==b1)
        {       t1.setText("Welcome to  BVIMIT");
        }
        else
        {       t1.setText(" ");
        }
}
public static void main(String s[])
{       new MyApplication();
}
}
```

## Output of Example



## Example

```
import java.awt.*;
import java.awt.event.*;
public class MyApplication extends Frame
   implements MouseListener
{  Button b1;
   TextField t1;
   Panel p1;
   MyApplication()

   {
        b1=new Button("Display");
        t1=new TextField(20);
        p1=new Panel();
```

## Implementing MouseListener

```
        b1.addMouseListener(this);
        p1.add(b1);
        add(p1,BorderLayout.NORTH);
        add(t1,BorderLayout.SOUTH);
        setSize(400,400);
        setVisible(true);}
    public void mouseEntered(MouseEvent e)
      {
                b1.setBackground(Color.RED);
      }
      public void mouseClicked(MouseEvent e)
      {         t1.setText("Welcome to IBM");
      }
```

Button is Registered to a MouseListener

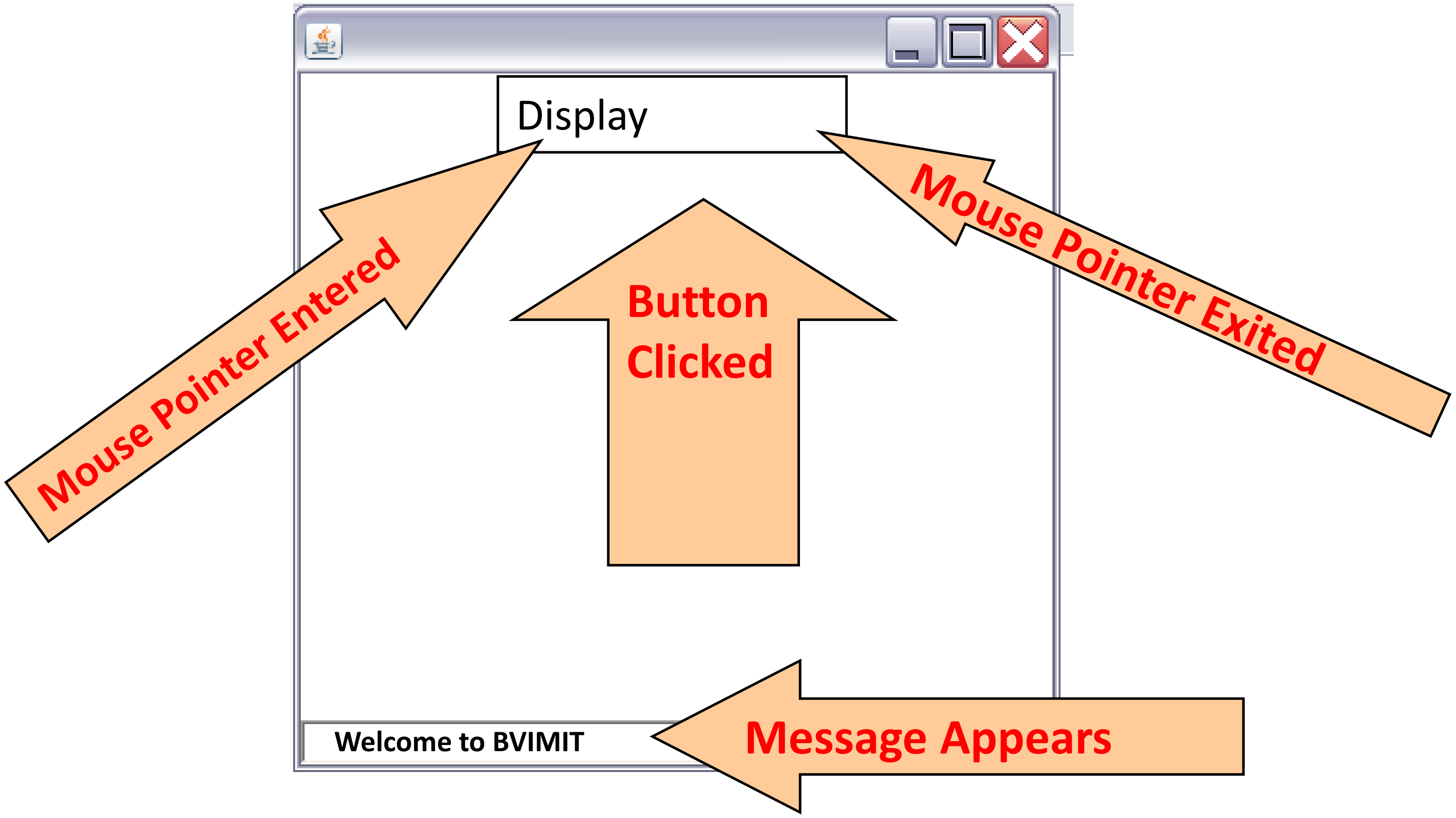
## Implementing MouseListener

```
public void mouseExited(MouseEvent e)
    {
                  b1.setBackground(Color.BLUE);
    }
    public void mousePressed(MouseEvent e)
    {}
    public void mouseReleased(MouseEvent e)
    {}

    public static void main(String s[])
    {
          new MyApplication();
    }}
```

Though the application does not need to respond to these activities, blank implementations must be provided

## Output of the Example



Back

# Event Classes Hierarchy

$$\begin{tikzcd}
 & & \text{Object} & & \\
 & & \text{EventObject} \arrow[u] & & \\
 & & \text{AWTEvent} \arrow[u] & & \\
\text{ActionEvent} \arrow[urr] & \text{ComponentEvent} \arrow[ur] & & \text{ItemEvent} \arrow[ul] & \text{TextEvent} \arrow[ull] \\
\text{FocusEvent} \arrow[ur] & \text{InputEvent} \arrow[u] & \text{WindowEvent} \arrow[ul] & & \\
\text{KeyEvent} \arrow[ur] & & \text{MouseEvent} \arrow[ul] & &
\end{tikzcd}$$

| Event, listener interface and add-and<br>remove-methods | Components supporting this event |
|---|---|
| ActionEvent<br>ActionListener ;<br>addActionListener( )<br>removeActionListener( ) | Button, List, TextField, MenuItem, CheckboxMenuItem, Menu and PopupMenu |
| AdjustmentEvent<br>AdjustmentListener ;<br>addAdjustmentListener( )<br>removeAdjustmentListener( ) | Scrollbar, Anything you create that implements Adjustable |
| ComponentEvent<br>ComponentListener<br>addComponentListener( )<br>removeComponentListener( ) | Component and its derivatives, including Button, Canvas, Checkbox, Choice, Container, Panel, Applet,<br>ScrollPane,Window,Dialog,FileDialog,Fra |

| Event, listener interface and add-and remove-methods | Components supporting this event |
|---|---|
| ContainerEvent<br>ContainerListener<br>addContainerListener( )<br>removeContainerListener( ) | Container and its derivatives, including  Panel, Applet, ScrollPane, Window, Dialog, FileDialog and Frame |
| FocusEvent<br>FocusListener<br>addFocusListener( )<br>removeFocusListener( ) | Component and its derivatives, including  Button, Canvas, Checkbox, Choice,<br>Container, Panel, Applet, ScrollPane,<br>Window, Dialog, FileDialog, Frame Label,<br>List, Scrollbar, TextArea and TextField |
| KeyEvent<br>KeyListener<br>addKeyListener( )<br>removeKeyListener( ) | Component and its derivatives, including Button, Canvas, Checkbox, Choice,<br>Container, Panel, Applet, ScrollPane,<br>Window,Dialog,FileDialog,Frame,Label,<br>List, Scrollbar, TextArea and TextField |

| Event, listener interface and add-and remove-methods | Components supporting this event |
|---|---|
| MouseEvent (for both clicks and motion)<br>MouseListener;<br>addMouseListener( )<br>removeMouseListener( ) | **Component and its derivatives**, including Button, Canvas, Checkbox, Choice,<br>Container, Panel, Applet, ScrollPane,<br>Window,Dialog,FileDialog,Frame,Label,<br>List, Scrollbar, TextArea and TextField<br>MouseEvent (for both clicks and motion) |
| MouseMotionEvent<br>MouseMotionListener<br>addMouseMotionListener( )<br>removeMouseMotionListener( ) | **Component and its derivatives**, including Button, Canvas, Checkbox, Choice,<br>Container, Panel, Applet, ScrollPane,<br>Window,Dialog,FileDialog,Frame,Label,<br>List, Scrollbar, TextArea and TextField |
| WindowEvent<br>WindowListener<br>addWindowListener( )<br>removeWindowListener( ) | **Window and its derivatives**, including Dialog, FileDialog, Frame, JFrame, |

- Event type: ItemEvent
  - listener interface: ItemListener
  - add-and-remove-methods : addItemListener( ), removeItemListener( )
  - Components supporting this event : Checkbox, CheckboxMenuItem, Choice, List and anything that implements ItemSelectable.
- Event type: TextEvent
  - listener interface: TextListener
  - add-and-remove-methods : addTextListener( ), removeTextListener( )
  - Components supporting this event :Anything derived from TextComponent, including TextArea and TextField

## Overview of Adapter Classes

- Adapter classes are used to reduce the code for Event Listeners
- avoids implementing all of the unneeded methods

All adapter classes are in:
java.awt.event package

- An adapter class provides an empty implementation of all methods in an event listener interface.
- 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( )** & **mouseMoved( ).**
- The signatures of these empty methods are exactly as defined in the MouseMotionListener interface. If you were interested in only *mouse drag* events, then
- you could simply extend MouseMotionAdapter and implement mouseDragged( ).
- The empty implementation of mouseMoved( ) would handle the mouse motion events for you.

## Adapter classes

- ComponentAdapter
- ContainerAdapter
- WindowAdapter
- MouseAdapter
- MouseMotionAdapter
- WindowAdapter
- FocusAdapter

## MouseAdapter class

```
package java.awt.event;
 import java.awt.*;
 import java.awt.event.*;
 public class MouseAdapter implements MouseListener {
                public void mouseClicked(MouseEvent evt) {}
                public void mousePressed(MouseEvent evt) {}
                public void mouseReleased(MouseEvent evt) {}
                public void mouseEntered(MouseEvent evt) {}
                public void mouseExited(MouseEvent evt) {} }
```

## Example using an Adapter class

```
import java.awt.*;
import java.awt.event.*;
public class MyApplication extends Frame
{
    Button b1;
    TextField t1;
    Panel p1;
    MyApplication()

    {
        b1=new Button("Display");
        t1=new TextField(20);
        p1=new Panel();
```

## Example(contd)

```
            b1.addMouseListener(new HandleEvent())
            p1.add(b1);
            add(p1,BorderLayout.NORTH);
            add(t1,BorderLayout.SOUTH);
            setSize(400,400);
            setVisible(true);}
 public class HandleEvent extends MouseAdapter
{
            public void mouseEntered(MouseEvent e)
            {
                        b1.setBackground(Color.RED);
            }
```

Inner Class extending the Mouse Adapter

## Example(Contd)

```
public void mouseClicked(MouseEvent e)
    {                       t1.setText("Welcome to IBM");
    }
public void mouseExited(MouseEvent e)``
    {
                            b1.setBackground(Color.BLUE);
    }
}
public static void main(String s[])
    {
            new MyApplication();
    }
}
```

output

# Nested Classes

- A class can be defined inside another class
- Benefits:
  - to structure and scope members
  - to connect logically related objects
- A nested class is considered a part of its enclosing class
- They share a trust relationship, i.e. everything is mutually accessible
- Nested types could be:
  - static – allows simple structuring of types
  - nonstatic – defines a special relationship between a nested object and an object of the enclosing class

# Inner Class

- Class in the Class
  - Provide the method to define the object type to use in the class
  - Solve the class name conflict to restrict the reference scope of class
  - Information hiding

```
class OuterClass {
        // ...
        class InnerClass {
                // ...
        }
}
```

## Inner Class

- Name Reference
  - OuterClass inside : Use InnerClass  Simple name
  - OuterClass outside : OuterClass.InnerClass

```
public static void main(String[] args) {
        OuterClass  outObj = new  OuterClass();
       OuterClass.InnerClass   inObj = outObj.new  InnerClass();
}
```

- Access Modifier
  - public, private, protected

Inner class cannot have static variable

## Inner classes

```
// This applet does NOT use an inner class.
import java.applet.*;
import java.awt.event.*;
/*
<applet code="MousePressedDemo" width=200
    height=100>
</applet>
*/
public class MousePressedDemo extends Applet {
public void init() {
addMouseListener(new MyMouseAdapter(this));
}
}
class MyMouseAdapter extends MouseAdapter {
MousePressedDemo mousePressedDemo;
public MyMouseAdapter(MousePressedDemo
    mousePressedDemo) {
this.mousePressedDemo = mousePressedDemo;
}
public void mousePressed(MouseEvent me) {
mousePressedDemo.showStatus("Mouse
    Pressed.");
}
}
```

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

# Anonymous Inner Classes

- An *anonymous inner class is one that is not assigned a name.*
- an anonymous inner class can facilitate the writing of event handlers.
- 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
- There is one top-level class in this program:**AnonymousInnerClassDemo. The init( )** method calls the **addMouseListener( ) method.**
- The syntax **new MouseAdapter( ) { ... } indicates to the compiler that the code** between the braces defines an anonymous inner class.

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