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 don't 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.

In AWT components, we came to know every component (except Panel and Label) generates events when interacted by the user like clicking over a button or pressing enter key in a text field etc. Listeners handle the events. Let us know the style (or design pattern) Java follows to handle the events.

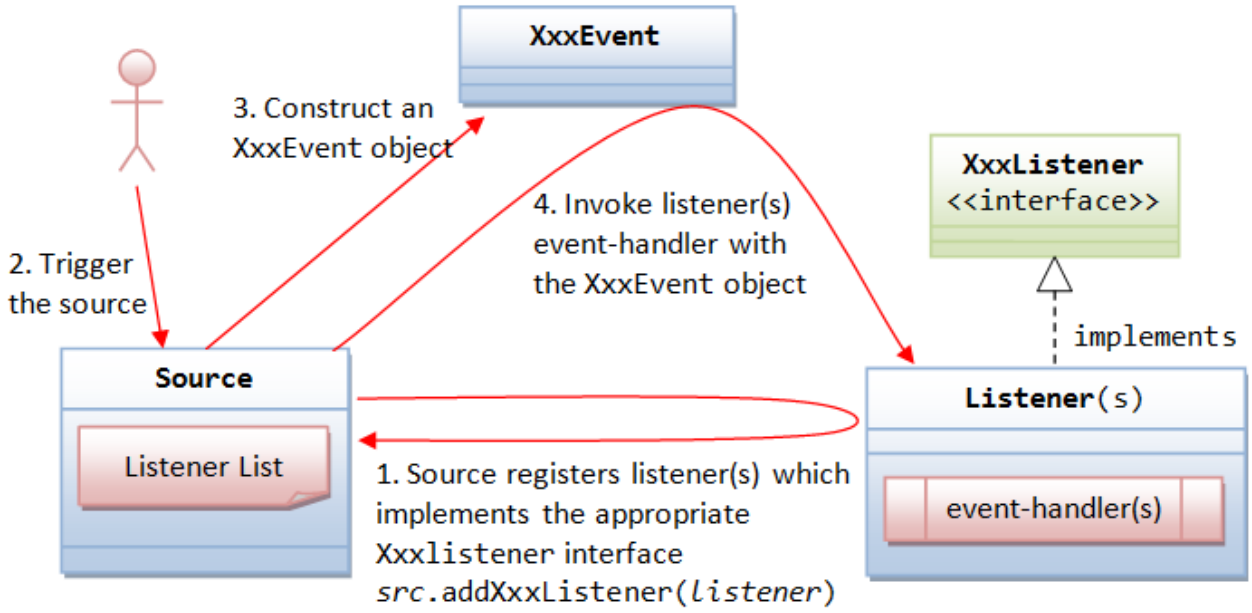
AWT Event-Handling

Java adopts the so-called "Event-Driven" (or "Event-Delegation") programming model for event-handling, similar to most of the visual programming languages (such as Visual Basic and Dot.NET etc).

The AWT's event-handling classes are kept in package java.awt.event.

The *source* object (such as Button and Textfield) interacts with the user. Upon triggered, it creates an *event* object. This *event* object will be messaged to all the *registered listener* object(s), and an appropriate event-handler method of the listener(s) is called-back to provide the response. In other words, *triggering a source fires an event to all its listener(s), and invoke an appropriate handler of the listener(s).*

Three objects are involved in the event-handling: a *source*, *listener*(s) and an *event* object.



The sequence of steps is illustrated above:

1. *The source object registers its listener(s)* for a certain type of *event*.

   Source object fires event **event** upon triggered. For example, clicking an Button fires an ActionEvent, mouse-click fires MouseEvent, key-type fires KeyEvent, etc.

   How the source and listener understand each other? The answer is via an agreed-upon interface. For example, if a source is capable of firing an event called XxxEvent (e.g., MouseEvent) involving various operational modes (e.g., mouse-clicked, mouse-entered, mouse-exited, mouse-pressed, and mouse-released). Firstly, we need to declare an interface called XxxListener (e.g., MouseListener) containing the names of the handler methods. Recall that an interface contains only abstract methods without implementation. For example,

```
// A MouseListener interface, which declares the signature of the handlers
//  for the various operational modes.
public interface MouseListener
{
  public void mousePressed(MouseEvent evt);//Called back upon mouse-button
                                            pressed

  public void mouseReleased(MouseEvent evt); // Called back upon mouse-button
                                              released
public void mouseClicked(MouseEvent evt);  // Called back upon mouse-button
                                            clicked (pressed and released)
public void mouseEntered(MouseEvent evt);  // Called back when mouse pointer
                                            entered the component
public void mouseExited(MouseEvent evt);   // Called back when mouse pointer
                                            exited the component
```

Event Delegation Model

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:

1. Event Sources } **Source**
2. Event Object classes }
3. Event Listeners }
4. Event Adapters } **Listener**

- **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 provide as with classes for source object.

  I Event Sources

  Event sources are components, subclasses of **java.awt.Component**, capable to generate events. The event source can be a button, TextField or a Frame etc.

  II Event Object classes

  Almost every event source(except Panel and Label) generates an event and is named by some Java class. For example, the event generated by button is known as **ActionEvent** and that of Checkbox is known as **ItemEvent**. All the events are listed in **java.awt.event** package. Following list gives a few components and their corresponding listeners.

Component	Event it generates
Button, TextField, List, Menu	ActionEvent
Frame	WindowEvent
Checkbox, Choice, List	ItemEvent
Scrollbar	AdjustmentEvent
Mouse (hardware)	MouseEvent
Keyboard (hardware)	KeyEvent

  The events generated by hardware components (like **MouseEvent** and **KeyEvent**) are known as **low-level events** and the events generated by software components (like Button, List) are known as **semantic events**.

Event Listeners

The events generated by the GUI components are handled by a special group of interfaces known as "**listeners**". Note, Listener is an interface. Every component has its own listener, say, **AdjustmentListener** handles the events of scrollbar. Some listeners handle the events of multiple

components. For example, **ActionListener** handles the events of Button, TextField, List and Menus. Listeners are from **java.awt.event** package.

Event Classe	Description	Listener Interface
ActionEvent	generated when button is pressed, menu-item is selected, list-item is double clicked	ActionListener
MouseEvent	generated when mouse is dragged, moved,clicked,pressed or released also when the enters or exit a component	MouseListener
KeyEvent	generated when input is received from keyboard	KeyListener
ItemEvent	generated when check-box or list item is clicked	ItemListener
TextEvent	generated when value of textarea or textfield is changed	TextListener
MouseWheelEvent	generated when mouse wheel is moved	MouseWheelListener
WindowEvent	generated when window is activated, deactivated, deiconified, iconified, opened or closed	WindowListener
ComponentEvent	generated when component is hidden, moved, resized or set visible	ComponentEventListener
ContainerEvent	generated when component is added or removed from container	ContainerListener
AdjustmentEvent	generated when scroll bar is manipulated	AdjustmentListener
FocusEvent	generated when component gains or loses keyboard focus	FocusListener

4. Event Adapters

When a listener includes many abstract methods to override, the coding becomes heavy to the programmer. For example, to close the frame, you override seven abstract methods of WindowListener, in which, infact you are using only one method. To avoid this heavy coding, the designers come with another group of classes known as "**adapters**". Adapters are abstract classes defined in **java.awt.event** package. Every listener that has more than one abstract method has got a corresponding adapter class.

Steps involved in event handling

- The User clicks the button and the event is generated.

- Now the object of concerned event class is created automatically and information about the source and the event get populated with in same object.
- Event object is forwarded to the method of registered listener class.
- the method is now get executed and returns.

Points to remember about listener

- In order to design a listener class we have to develop some listener interfaces. These Listener interfaces forecast some public abstract **callback methods** which must be implemented by the listener class.
- If you do not implement the any if the predefined interfaces then your class can not act as a listener class for a source object.

What happens internally at a button click?

We know the events are handled by **listeners** and **ActionListener** handles the events of a button. Observe the following skeleton code.

```
public class ButtonDemo extends Frame implements ActionListener
{
  public ButtonDemo()
  {
    Button btn = new Button("OK");
    btn.addActionListener(this);
    add(btn);
  }
  public void actionPerformed(ActionEvent e)
  {
    String str = e.getActionCommand();
  }
}
```

A button object **btn** is created for which events are yet to be linked. For this, the first step is implementing **ActionListener** to the class ButtonDemo. In the second step, we link or register the button **btn** with the **ActionListener**. For this **addActionListener()** method of Button class is used. The parameter "**this**" refers the **ActionListener**.

btn.addActionListener(this);

With the above statement, **btn** is linked with the **ActionListener**.

1. When the button **btn** is clicked, the button generates an event called **ActionEvent**. It is the nature of the button taken care by JVM.

2. This **ActionEvent** reaches the **ActionListener** because we registered the button with ActionListener earlier.

3. Now, the question is, what **ActionListener** does with the **ActionEvent** object it received?

The **ActionListener** simply calls **actionPerformed()** method and passes the **ActionEvent** object to the parameter as follows.

public void actionPerformed(ActionEvent e)

The parameter for the above method comes from ActionListener.

4. Finally the **ActionEvent** object generated by the button **btn** reaches the **e** object of ActionEvent. All this is done by JVM implicitly. For this reason, the **getActionCommand()** method of ActionEvent class knows the label of the button **btn**.

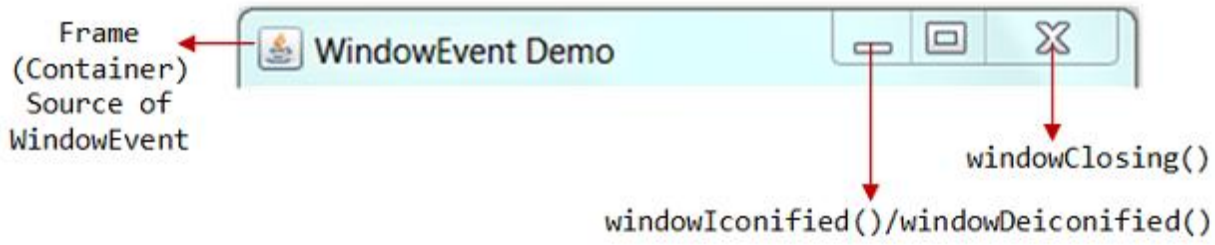
String str = e.getActionCommand();

The **e** represents an object of **ActionEvent** and the value for the **e** is coming from button **btn**. **str** is nothing but the label of the button OK.

Example 3: WindowEvent and WindowListener Interface

A WindowEvent is fired (to all its WindowEvent listeners) when a window (e.g., Frame) has been opened/closed, activated/deactivated, iconified/deiconified via the 3 buttons at the top-right corner or other means. The source of WindowEvent shall be a top-level window-container such as Frame.

A WindowEvent listener must implement WindowListener interface, which declares 7 abstract event-handling methods, as follows. Among them, the windowClosing(), which is called back upon clicking the window-close button, is the most commonly-used.



public void windowClosing**(WindowEvent e)**
// Called-back when the user attempts to close the window by clicking the window close button.
// This is the most-frequently used handler.
public void windowOpened(WindowEvent e)
// Called-back the first time a window is made visible.
public void windowClosed(WindowEvent e)
// Called-back when a window has been closed as the result of calling dispose on the window.
public void windowActivated(WindowEvent e)
// Called-back when the Window is set to be the active Window.

```
public void windowDeactivated(WindowEvent e)
  // Called-back when a Window is no longer the active Window.
public void windowIconified(WindowEvent e)
  // Called-back when a window is changed from a normal to a minimized state.
public void windowDeiconified(WindowEvent e)
  // Called-back when a window is changed from a minimized to a normal state.
```

The following program added support for "close-window button" to Example 1: AWTCounter.

```
import java.awt.*;       // Using AWT containers and components
import java.awt.event.*;  // Using AWT events and listener interfaces

// An AWT GUI program inherits the top-level container java.awt.Frame
public class WindowEventDemo extends Frame
    implements ActionListener, WindowListener {
    // This class acts as listener for ActionEvent and WindowEvent
    // Java supports only single inheritance, where a class can extend
    // one superclass, but can implement multiple interfaces.

  private TextField tfCount;
  private Button btnCount;
  private int count = 0;  // Counter's value

  /** Constructor to setup the UI components and event handling */
  public WindowEventDemo() {
    setLayout(new FlowLayout()); // "super" Frame sets to FlowLayout

    add(new Label("Counter"));   // "super" Frame adds an anonymous Label

    tfCount = new TextField("0", 10); // Allocate TextField
    tfCount.setEditable(false);       // read-only
    add(tfCount);                     // "super" Frame adds tfCount

    btnCount = new Button("Count");  // Declare and allocate a Button
    add(btnCount);                   // "super" Frame adds btnCount

    btnCount.addActionListener(this);
      // btnCount fires ActionEvent to its registered ActionEvent listener
      // btnCount adds "this" object as an ActionEvent listener

    addWindowListener(this);
      // "super" Frame fires WindowEvent to its registered WindowEvent listener
      // "super" Frame adds "this" object as a WindowEvent listener

    setTitle("WindowEvent Demo"); // "super" Frame sets title
    setSize(250, 100);            // "super" Frame sets initial size
    setVisible(true);             // "super" Frame shows
  }

  /** The entry main() method */
```

```
  public static void main(String[] args) {
    new WindowEventDemo();  // Let the construct do the job
  }

  /** ActionEvent handler */
  @Override
  public void actionPerformed(ActionEvent evt) {
    ++count;
    tfCount.setText(count + "");
  }

  /** WindowEvent handlers */
  // Called back upon clicking close-window button
  @Override
  public void windowClosing(WindowEvent e) {
    System.exit(0);  // Terminate the program
  }

  // Not Used, but need to provide an empty body
  @Override
  public void windowOpened(WindowEvent e) { }
  @Override
  public void windowClosed(WindowEvent e) { }
  @Override
  public void windowIconified(WindowEvent e) { }
  @Override
  public void windowDeiconified(WindowEvent e) { }
  @Override
  public void windowActivated(WindowEvent e) { }
  @Override
  public void windowDeactivated(WindowEvent e) { }
}
```

In this example, we shall modify the earlier AWTCounter example to handle the WindowEvent. Recall that pushing the "close-window" button on the AWTCounter has no effect, as it did not handle the WindowEvent of windowClosing(). We included the WindowEvent handling codes in this example.

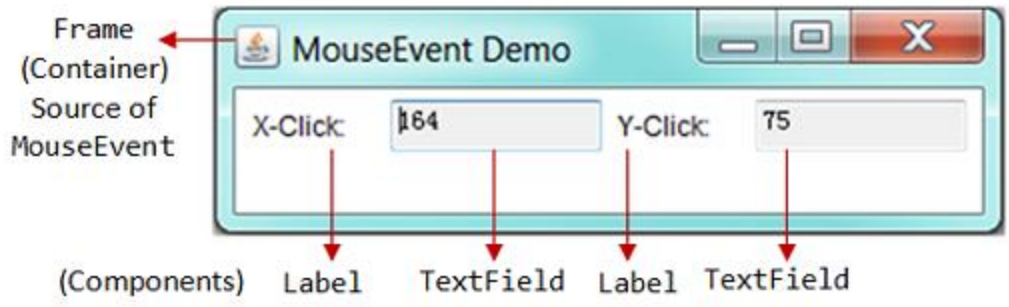
1. We identify super Frame as the source object.
2. The Frame fires the WindowEvent to all its registered WindowEvent listener(s).
3. We select this object as the WindowEvent listener (for simplicity)
4. We register this object as the WindowEvent listener to the source Frame via method addWindowListener(this).
5. The WindowEvent listener (this class) is required to implement the WindowListener interface, which declares 7 abstract methods: windowOpened(), windowClosed(), windowClosing(), windowActivated(), windowDeactivated(), windowIconified() and windowDeiconified().
6. We override the windowClosing() handler to terminate the program using System.exit(0). We ignore the other 6 handlers, but required to provide an empty body.

Example 4: MouseEvent and MouseListener Interface

A MouseEvent is fired to all its registered listeners, when you press, release, or click (press followed by release) a mouse-button (left or right button) at the source object; or position the mouse-pointer at (enter) and away (exit) from the source object.

A MouseEvent listener must implement the MouseListener interface, which declares the following five abstract methods:

public void mouseClicked(MouseEvent e)
// Called-back when the mouse-button has been clicked (pressed followed by released) on the source.
public void mousePressed(MouseEvent e)
public void mouseReleased(MouseEvent e)
// Called-back when a mouse-button has been pressed/released on the source.
// A mouse-click invokes mousePressed(), mouseReleased() and mouseClicked().
public void mouseEntered(MouseEvent e)
public void mouseExited(MouseEvent e)
// Called-back when the mouse-pointer has entered/exited the source.




```
import java.awt.*;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;

public class MouseEventDemo extends Frame implements MouseListener {

  // Private variables
  private TextField tfMouseX; // to display mouse-click-x
  private TextField tfMouseY; // to display mouse-click-y

  // Constructor - Setup the UI
  public MouseEventDemo() {
    setLayout(new FlowLayout()); // "super" frame sets layout

    // Label
    add(new Label("X-Click: ")); // "super" frame adds component

    // TextField
    tfMouseX = new TextField(10); // 10 columns
    tfMouseX.setEditable(false);  // read-only
```

```
    add(tfMouseX);               // "super" frame adds component

    // Label
    add(new Label("Y-Click: ")); // "super" frame adds component

    // TextField
    tfMouseY = new TextField(10);
    tfMouseY.setEditable(false);  // read-only
    add(tfMouseY);               // "super" frame adds component

    addMouseListener(this);
       // "super" frame fires the MouseEvent
       // "super" frame adds "this" object as MouseEvent listener

    setTitle("MouseEvent Demo"); // "super" Frame sets title
    setSize(350, 100);           // "super" Frame sets initial size
    setVisible(true);            // "super" Frame shows
  }

  public static void main(String[] args) {
    new MouseEventDemo();  // Let the constructor do the job
  }

  // MouseEvent handlers
  @Override
  public void mouseClicked(MouseEvent e) {
    tfMouseX.setText(e.getX() + "");
    tfMouseY.setText(e.getY() + "");
  }

  @Override
  public void mousePressed(MouseEvent e) { }
  @Override
  public void mouseReleased(MouseEvent e) { }
  @Override
  public void mouseEntered(MouseEvent e) { }
  @Override
  public void mouseExited(MouseEvent e) { }
}
```

In this example, we setup a GUI with 4 components (two Labels and two non-editable TextFields), inside a top-level container Frame, arranged in FlowLayout.

To demonstrate the MouseEvent:

1. We identity super Frame as the source object.
2. The Frame fires a MouseEvent to all its MouseEvent listener(s) when you click/press/release a mouse-button or enter/exit with the mouse-pointer.
3. We select this object as the MouseEvent listener (for simplicity).

4. We register this object as the MouseEvent listener to super Frame (source) via the method addMouseListener(this).
5. The listener (this class) is required to implement the MouseListener interface, which declares 5 abstract methods: mouseClicked(), mousePressed(), mouseReleased(), mouseEntered(), and mouseExit(). We override the mouseClicked() to display the (x, y) co-ordinates of the mouse click on the two displayed TextFields. We ignore all the other handlers (for simplicity - but you need to provide an empty body for compilation).

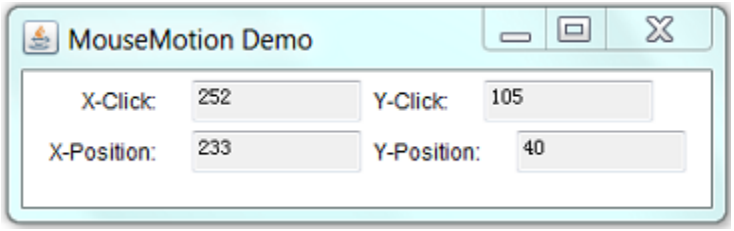
Try: Include a WindowListener to handle the close-window button.

Example 5: MouseEvent and MouseMotionListener Interface

A MouseEvent is also fired when you moved and dragged the mouse pointer at the source object. But you need to use MouseMotionListener to handle the mouse-move and mouse-drag. The MouseMotionListener interface declares the following two abstract methods:

public void mouseDragged(MouseEvent e)
// Called-back when a mouse-button is pressed on the source component and then dragged.

public void mouseMoved(MouseEvent e)
// Called-back when the mouse-pointer has been moved onto the source component but no buttons have been pushed.




```
import java.awt.*;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;

// An AWT GUI program inherits from the top-level container java.awt.Frame
public class MouseMotionDemo extends Frame
      implements MouseListener, MouseMotionListener {
      // This class acts as MouseListener and MouseMotionListener

   // To display the (x, y) coordinates of the mouse-clicked
   private TextField tfMouseClickX;
   private TextField tfMouseClickY;
   // To display the (x, y) coordinates of the current mouse-pointer position
   private TextField tfMousePositionX;
   private TextField tfMousePositionY;

   /** Constructor to setup the GUI */
```

```
public MouseMotionDemo() {
  setLayout(new FlowLayout()); // "this" frame sets to FlowLayout

  add(new Label("X-Click: "));
  tfMouseClickX = new TextField(10);
  tfMouseClickX.setEditable(false);
  add(tfMouseClickX);
  add(new Label("Y-Click: "));
  tfMouseClickY = new TextField(10);
  tfMouseClickY.setEditable(false);
  add(tfMouseClickY);

  add(new Label("X-Position: "));
  tfMousePositionX = new TextField(10);
  tfMousePositionX.setEditable(false);
  add(tfMousePositionX);
  add(new Label("Y-Position: "));
  tfMousePositionY = new TextField(10);
  tfMousePositionY.setEditable(false);
  add(tfMousePositionY);

  addMouseListener(this);
  addMouseMotionListener(this);
    // "super" frame fires MouseEvent to all its registered MouseListener and MouseMotionListener
    // "super" frame adds "this" object as MouseListener and MouseMotionListener

  setTitle("MouseMotion Demo"); // "super" Frame sets title
  setSize(400, 120);           // "super" Frame sets initial size
  setVisible(true);            // "super" Frame shows
}

/** The entry main() method */
public static void main(String[] args) {
  new MouseMotionDemo();  // Let the constructor do the job
}

/** MouseListener handlers */
// Called back when a mouse-button has been clicked
@Override
public void mouseClicked(MouseEvent e) {
  tfMouseClickX.setText(e.getX() + "");
  tfMouseClickY.setText(e.getY() + "");
}

// Not Used, but need to provide an empty body for compilation
@Override
public void mousePressed(MouseEvent e) { }
@Override
public void mouseReleased(MouseEvent e) { }
@Override
public void mouseEntered(MouseEvent e) { }
```

```
  @Override
  public void mouseExited(MouseEvent e) { }

  /** MouseMotionEvent handlers */
  // Called back when the mouse-pointer has been moved
  @Override
  public void mouseMoved(MouseEvent e) {
    tfMousePositionX.setText(e.getX() + "");
    tfMousePositionY.setText(e.getY() + "");
  }

  // Not Used, but need to provide an empty body for compilation
  @Override
  public void mouseDragged(MouseEvent e) { }
}
```

In this example, we shall illustrate both the MouseListener and MouseMotionListener.

1. We identify super Frame as the source, which fires the MouseEvent to its registered MouseListener and MouseMotionListener.
2. We select this object as the MouseListener and MouseMotionListner (for simplicity).
3. We register this object as the listener to super Frame via method addMouseListener(this) and addMouseMotionListener(this).
4. The MouseMotionListener (this class) needs to implement 2 abstract methods: mouseMoved() and mouseDragged() declared in the MouseMotionListener interface.
5. We override the mouseMoved() to display the (x, y) position of the mouse pointer. We ignore the MouseDragged() handler by providing an empty body for compilation.

Try: Include a WindowListener to handle the close-window button.

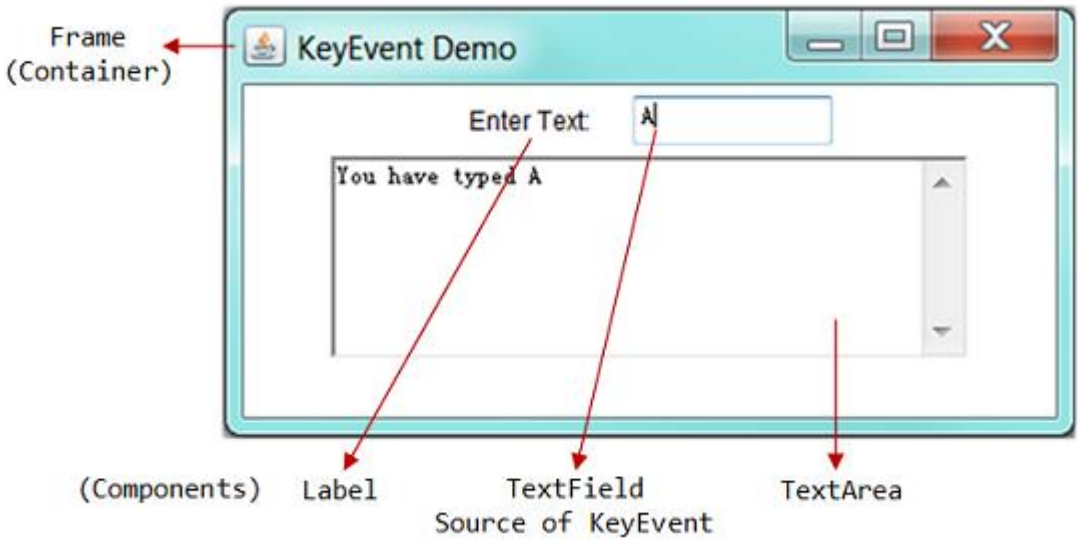
Example 6: KeyEvent and KeyListener Interface

A KeyEvent is fired (to all its registered KeyListeners) when you pressed, released, and typed (pressed followed by released) a key on the source object. A KeyEvent listener must implement KeyListener interface, which declares three abstract methods:

public void keyTyped(KeyEvent e)
// Called-back when a key has been typed (pressed and released).

public void keyPressed(KeyEvent e)
public void keyReleased(KeyEvent e)

// Called-back when a key has been pressed/released.




```
import java.awt.*;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;

// An AWT GUI program inherits from the top-level container java.awt.Frame
public class KeyEventDemo extends Frame implements KeyListener
{
    // This class acts as KeyEvent Listener

  private TextField tfInput;  // single-line TextField to receive tfInput key
  private TextArea taDisplay; // multi-line TextArea to taDisplay result

   /** Constructor to setup the GUI */
   public KeyEventDemo() {
     setLayout(new FlowLayout()); // "super" frame sets to FlowLayout

     add(new Label("Enter Text: "));
     tfInput = new TextField(10);
     add(tfInput);
     taDisplay = new TextArea(5, 40); // 5 rows, 40 columns
     add(taDisplay);

     tfInput.addKeyListener(this);
        // tfInput TextField fires KeyEvent to its registered KeyListeners
        // tfInput adds "this" object as a KeyEvent listener

     setTitle("KeyEvent Demo"); // "super" Frame sets title
     setSize(400, 200);         // "super" Frame sets initial size
     setVisible(true);          // "super" Frame shows
   }

   /** The entry main() method */
   public static void main(String[] args) {
```

```
    new KeyEventDemo();  // Let the constructor do the job
  }

  /** KeyEvent handlers */
  // Called back when a key has been typed (pressed and released)
  @Override
  public void keyTyped(KeyEvent e) {
    taDisplay.append("You have typed " + e.getKeyChar() + "\n");
  }

  // Not Used, but need to provide an empty body for compilation
  @Override
  public void keyPressed(KeyEvent e) { }
  @Override
  public void keyReleased(KeyEvent e) { }
}
```

In this example:

1. We identify the tfInput (TextField) as the source object.
2. The source fires a KeyEvent when you press/release/type a key to all its KeyEvent listener(s).
3. We select this object as the KeyEvent listener.
4. We register this object as the KeyEvent listener to the source TextField via method input.addKeyListener(this).
5. The KeyEvent listener (this class) needs to implement the KeyListener interface, which declares 3 abstract methods: keyTyped(), keyPressed(), keyReleased().
6. We override the keyTyped() to display key typed on the display TextArea. We ignore the keyPressed() and keyReleased().

A Named Inner Class as Event Listener

A nested class is useful if you need a *small* class which relies on the enclosing outer class for its private variables and methods. It is ideal in an event-driven environment for implementing event handlers. This is because the event handling methods (in a listener) often require access to the private variables (e.g., a private TextField) of the outer class.

In this example (modified from Example 1 AWTCounter), instead of using "this" as the ActionEvent listener for the Button, we define a new class called BtnCountListener, and create an instance of BtnCountListener as the ActionEvent listener for the btnCount. The BtnCountListener needs to implement the ActionListener interface, and override the actionPerformed() handler. Since "this" is no long a ActionListener, we remove the "implements ActionListener" from "this" class's definition.

BtnCountListener needs to be defined as an inner class, as it needs to access private variables (count and tfCount) of the outer class.

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

// An AWT GUI program inherits from the top-level container java.awt.Frame

public class AWTCounterNamedInnerClass extends Frame {

   // This class is NOT a ActionListener, hence, it does not
      implement ActionListener

   // The event-handler actionPerformed() needs to access these
      "private" variables
   private TextField tfCount;
   private int count = 0;

   /** Constructor to setup the GUI */
   public AWTCounterNamedInnerClass () {
      setLayout(new FlowLayout());  // "super" Frame sets to FlowLayout
      add(new Label("Counter"));    // anonymous instance of Label
      tfCount = new TextField("0", 10);
      tfCount.setEditable(false);   // read-only
      add(tfCount);                 // "super" Frame adds tfCount

      Button btnCount = new Button("Count");
      add(btnCount);                // "super" Frame adds btnCount

      // Construct an anonymous instance of BtnCountListener (a named inner class).
      // btnCount adds this instance as a ActionListener.
      btnCount.addActionListener(new BtnCountListener());

      setTitle("AWT Counter");
      setSize(250, 100);
      setVisible(true);
   }

   /** The entry main method */
   public static void main(String[] args) {
      new AWTCounterNamedInnerClass(); // Let the constructor do the job
   }

   /**
    * BtnCountListener is a "named inner class" used as ActionListener.
    * This inner class can access private variables of the outer class.
    */
   private class BtnCountListener implements ActionListener {
      @Override
      public void actionPerformed(ActionEvent e) {
         ++count;
         tfCount.setText(count + "");
      }
   }
```

```
}
```

Dissecting the Program

- An inner class named BtnCountListener is used as the ActionListner.
- An anonymous instance of the BtnCountListener inner class is constructed. The btnCount source object adds this instance as a listener, as follows:
- btnCount.addActionListener(new BtnCountListener());
- The inner class can access the private variable tfCount and count of the outer class.
- Since "this" is no longer a listener, we remove the "implements ActionListener" from this class' definition.
- The inner class is compiled into AWTCount$BtnCountListener.class, in the format of *OuterClassName$InnerClassName*.class.

Using an Ordinary (Outer) Class as Listener

Try moving the BtnCountListener class outside, and define it as an ordinary class. You would need to pass a reference of the AWTConnter into the constructor of BtnCountListener, and use this reference to access variables tfCount and count, through public getters or granting them to public access.

```
// An ordinary outer class used as ActionListener for the Button
public class BtnCountListener implements ActionListener {
  AWTCounter frame;
  public BtnCountListener(AWTCounter frame) {
    this.frame = frame;
  }

  @Override
  public void actionPerformed(ActionEvent e) {
    frame.count++;
    frame.tfCount.setText(frame.count + "");
  }
}
```

This code is messy!

An Anonymous Inner Class as Event Listener

Instead of using a *named inner class* (called BtnCountListner in the previous example), we shall use an inner class without a name, known as *anonymous inner class* as the ActionListener in this example.

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

// An AWT GUI program inherits from the top-level container java.awt.Frame
public class AWTCounterAnonymousInnerClass extends Frame {
  // This class is NOT a ActionListener, hence, it does not implement ActionListener

  // The event-handler actionPerformed() needs to access these private variables
```

```
  private TextField tfCount;
  private int count = 0;

  /** Constructor to setup the GUI */
  public AWTCounterAnonymousInnerClass () {
    setLayout(new FlowLayout());  // "super" Frame sets to FlowLayout
    add(new Label("Counter"));    // an anonymous instance of Label
    tfCount = new TextField("0", 10);
    tfCount.setEditable(false);   // read-only
    add(tfCount);                 // "super" Frame adds tfCount

    Button btnCount = new Button("Count");
    add(btnCount);                // "super" Frame adds btnCount

    // Construct an anonymous instance of an anonymous class.
    // btnCount adds this instance as a ActionListener.
    btnCount.addActionListener(new ActionListener() {
      @Override
      public void actionPerformed(ActionEvent e) {
        ++count;
        tfCount.setText(count + "");
      }
    });

    setTitle("AWT Counter");
    setSize(250, 100);
    setVisible(true);
  }

  /** The entry main method */
  public static void main(String[] args) {
    new AWTCounterAnonymousInnerClass(); // Let the constructor do the job
  }
}
```

Dissecting the Program

- Again, "this" class is NOT used as the ActionEvent listener. Hence, we remove the "implements ActionListener" from this class' definition.
- The anonymous inner class is given a name generated by the compiler, and compiled into *OuterClassName*$*n*.class, where *n* is a running number of the inner classes of this outer class.
- An anonymous instance of an anonymous inner class is constructed, and passed as the argument of the addActionListener() method as follows:

```
btnCount.addActionListener(new ActionListener() {
  @Override
  public void actionPerformed(ActionEvent e) {
    ++count;
    tfCount.setText(count + "");
  }
});
```

The above codes is equivalent to and compiled as:

```
private class N implements ActionListener
{  // N is a running number of the inner classes created
   @Override
   public void actionPerformed(ActionEvent e) {
      ++count;
      tfCount.setText(count + "");
   }
}
btnCount.addActionListener(new N());

// Or
N n = new N()
btnCount.addActionListener(n);
```

Properties of Anonymous Inner Class

1. The anonymous inner class is define inside a method, instead of a member of the outer class (class member). It is *local* to the method and cannot be marked with access modifier (such as public, private) or static, just like any local variable of a method.
2. An anonymous inner class must always extend a superclass or implement an interface. The keyword "extends" or "implements" is NOT required in its declaration. An anonymous inner class must implement all the abstract methods in the superclass or in the interface.
3. An anonymous inner class always uses the default (no-arg) constructor from its superclass to create an instance. If an anonymous inner class implements an interface, it uses the java.lang.Object().
4. An anonymous inner class is compiled into a class named OuterClassName$*n*.class, where *n* is a running number of inner classes within the outer class.
5. An instance of an anonymous inner class is constructed via this syntax:

```
new SuperClassName/InterfaceName() {  // extends superclass or implements interface
                                      // invoke the default no-arg constructor or Object[]
  // Implement abstract methods in superclass/interface
  // More methods if necessary
  ......
}
```

The created instance can be assigned to a variable or used as an argument of a method.

Using the Same Listener Instance for All the Buttons

If you use the same instance as the listener for the 3 buttons, you need to determine which button has fired the event. It is because all the 3 buttons trigger the same event-handler method.

Using ActionEvent's getActionCommand()

In the following example, we use the same instance of a named inner class as the listener for all the 3 buttons. The listener needs to determine which button has fired the event. This can be accomplished via the ActionEvent's getActionCommonad() method, which returns the button's label.

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

// An AWT GUI program inherits the top-level container java.awt.Frame
public class AWTCounter3Buttons1Listener extends Frame {
   private TextField tfCount;
   private int count = 0;

   /** Constructor to setup the GUI */
   public AWTCounter3Buttons1Listener () {
      setLayout(new FlowLayout());
      add(new Label("Counter"));
      tfCount = new TextField("0", 10);
      tfCount.setEditable(false);
      add(tfCount);

      // Create buttons
      Button btnCountUp = new Button("Count Up");
      add(btnCountUp);
      Button btnCountDown = new Button("Count Down");
      add(btnCountDown);
      Button btnReset = new Button("Reset");
      add(btnReset);

      // Allocate an instance of inner class BtnListener.
      BtnListener listener = new BtnListener();
      // Use the same listener to all the 3 buttons.
      btnCountUp.addActionListener(listener);
      btnCountDown.addActionListener(listener);
      btnReset.addActionListener(listener);

      setTitle("AWT Counter");
      setSize(400, 100);
      setVisible(true);
   }

   /** The entry main method */
   public static void main(String[] args) {
      new AWTCounter3Buttons1Listener();  // Let the constructor do the job
   }

   /**
    * BtnListener is a named inner class used as ActionEvent listener for the buttons.
    */
```

```
  private class BtnListener implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent e) {
      // Need to determine which button has fired the event.
      // getActionCommand() returns the button's label
      String btnLabel = e.getActionCommand();
      if (btnLabel.equals("Count Up")) {
        ++count;
      } else if (btnLabel.equals("Count Down")) {
        --count;
      } else {
        count = 0;
      }
      tfCount.setText(count + "");
    }
  }
}
```