8 GUI Event Handling

Topics

- The Delegation Event Model
- Event Classes
- Event Listeners
  - ActionListener Method
  - MouseListener Methods
  - MouseMotionListener Methods
  - WindowListener Methods
  - Guidelines for Creating Applications Handling GUI Events

Topics

- Adapter Classes
- Inner Classes
- Anonymous Inner Classes

The Delegation Event Model

- The Delegation Event Model
  - Model used by Java to handle user interaction with GUI components
  - Describes how your program can respond to user interaction

- Three important components:
  - Event Source
  - Event Listener/Handler
  - Event Object

The Delegation Event Model

- Event Source
  - GUI component that generates the event
  - Example: button, mouse, keyboard

- Event Listener/Handler
  - Receives news of events and processes user's interaction
  - Example: displaying an information useful to the user, computing for a value

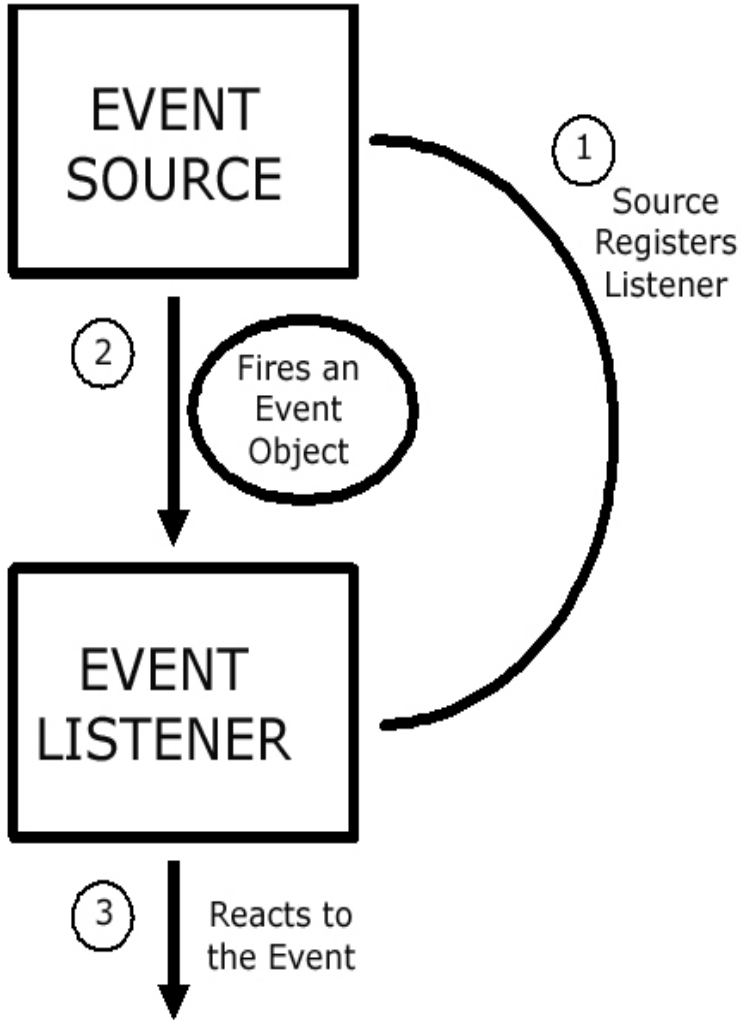
The Delegation Event Model

- Event Object
  - Created when an event occurs (i.e., user interacts with a GUI component)
  - Contains all necessary information about the event that has occurred
    - Type of event that has occurred
    - Source of the event
  - May have one of several event classes as data type

The Delegation Event Model

- A listener should be registered with a source
- Once registered, listener waits until an event occurs
- When an event occurs
  - An event object created
  - Event object is fired by the source to the registered listeners
- Once the listener receives an event object from the source
  - Deciphers the notification
  - Processes the event that occurred.

The Delegation Event Model



Registration of Listeners

- Event source registering a listener:

  ```
  void add<Type>Listener(<Type>Listener listenerObj)
  ```

  where,

  - *<Type>* depends on the type of event source
    - Can be *Key*, *Mouse*, *Focus*, *Component*, *Action* and others
  - One event source can register several listeners

- Registered listener being unregistered:

  ```
  void remove<Type>Listener(<Type>Listener listenerObj)
  ```

Event Classes

- An event object has an event class as its reference data type

- The *EventObject* class
  - Found in the *java.util* package

- The *AWTEvent* class
  - An immediate subclass of *EventObject*
  - Defined in *java.awt* package
  - Root of all AWT-based events
  - Subclasses follow this naming convention:

```
<Type>Event
```

Event Classes

Event Class	*Description*
ComponentEvent	Extends *AWTEvent*. Instantiated when a component is moved, resized, made visible or hidden.
InputEvent	Extends *ComponentEvent*. The abstract root event class for all component-level input event classes.
ActionEvent	Extends *AWTEvent*. Instantiated when a button is pressed, a list item is double-clicked, or a menu item is selected.
ItemEvent	Extends *AWTEvent*. Instantiated when an item is selected or deselected by the user, such as in a list or a checkbox.
KeyEvent	Extends *InputEvent*. Instantiated when a key is pressed, released or typed.
MouseEvent	Extends *InputEvent*. Instantiated when a mouse button is pressed, released, or clicked (pressed and released), or when a mouse cursor enteres or exits a visible part of a component.
TextEvent	Extends *AWTEvent*. Instantiated when the value of a text field or a text area is changed.
WindowEvent	Extends *ComponentEvent*. Instantiated when a *Window* object is opened, closed, activated, deactivated, iconified, deiconified, or when focus is transferred into or out of the window.

Event Listeners

- Classes that implement the *<Type>Listener* interfaces

- Common *<Type>Listener* interfaces:

Event Listeners	***Description***
ActionListener	Receives action events.
MouseListener	Receives mouse events.
MouseMotionListener	Receives mouse motion events, which include dragging and moving the mouse.
WindowListener	Receives window events.

ActionListener Method

- Contains exactly one method

ActionListener Method
`public void actionPerformed(ActionEvent e)`
Contains the handler for the *ActionEvent e* that occurred.

MouseListener Methods

MouseListener Methods
`public void mouseClicked(MouseEvent e)`
Contains the handler for the event when the mouse is clicked (i.e., pressed and released).
`public void mouseEntered(MouseEvent e)`
Contains the code for handling the case wherein the mouse enters a component.
`public void mouseExited(MouseEvent e)`
Contains the code for handling the case wherein the mouse exits a component.
`public void mousePressed(MouseEvent e)`
Invoked when the mouse button is pressed on a component.
`public void mouseReleased(MouseEvent e)`
Invoked when the mouse button is released on a component.

MouseMotionListener Methods

MouseListener Methods
`public void mouseDragged(MouseEvent e)`
Contains the code for handling the case wherein the mouse button is pressed on a component and dragged. Called several times as the mouse is dragged.
`public void mouseMoved(MouseEvent e)`
Contains the code for handling the case wherein the mouse cursor is moved onto a component, without the mouse button being pressed. Called multiple times as the mouse is moved.

WindowListener Methods

WindowListener Methods
`public void windowOpened(WindowEvent e)`
Contains the code for handling the case when the *Window* object is opened (i.e., made visible for the first time).
`public void windowClosing(WindowEvent e)`
Contains the code for handling the case when the user attempts to close *Window* object from the object's system menu.
`public void windowClosed(WindowEvent e)`
Contains the code for handling the case when the *Window* object was closed after calling dispose (i.e., release of resources used by the source) on the object.
`public void windowActivated(WindowEvent e)`
Invoked when a *Window* object is the active window (i.e., the window in use).
`public void windowDeactivated(WindowEvent e)`
Invoked when a *Window* object is no longer the active window.
`public void windowIconified(WindowEvent e)`
Called when a *Window* object is minimized.
`public void windowDeiconified(WindowEvent e)`
Called when a *Window* object reverts from a minimized to a normal state.

Creating GUI Applications with Event Handling

- Guidelines:
  1. Create a GUI class
     - Describes and displays the appearance of your GUI application
  2. Create a class implementing the appropriate listener interface
     - May refer to the same class as step 1
  3. In the implementing class
     - Override ALL methods of the appropriate listener interface
     - Describe in each method how you would like the event to be handled
     - May give empty implementations for methods you don't need
  4. Register the listener object with the source
     - The object is an instantiation of the listener class in step 2
     - Use the *add<Type>Listener* method

Mouse Events Example

```
1  import java.awt.*;
2  import java.awt.event.*;
3  public class MouseEventsDemo extends Frame implements
                    MouseListener, MouseMotionListener {
4      TextField tf;
5      public MouseEventsDemo(String title){
6          super(title);
7          tf = new TextField(60);
8          addMouseListener(this);
9      }
10 //continued...
```

Mouse Events Example

```
11    public void launchFrame() {
12        /* Add components to the frame */
13        add(tf, BorderLayout.SOUTH);
14        setSize(300,300);
15        setVisible(true);
16    }
17    public void mouseClicked(MouseEvent me) {
18        String msg = "Mouse clicked.";
19        tf.setText(msg);
20    }
21 //continued...
```

Mouse Events Example

```
22    public void mouseEntered(MouseEvent me) {
23        String msg = "Mouse entered component.";
24        tf.setText(msg);
25    }
26    public void mouseExited(MouseEvent me) {
27        String msg = "Mouse exited component.";
28        tf.setText(msg);
29    }
30    public void mousePressed(MouseEvent me) {
31        String msg = "Mouse pressed.";
32        tf.setText(msg);
33    }
34 //continued...
```

Mouse Events Example

```
35    public void mouseReleased(MouseEvent me) {
36        String msg = "Mouse released.";
37        tf.setText(msg);
38    }
39    public void mouseDragged(MouseEvent me) {
40        String msg = "Mouse dragged at " + me.getX()
41                          + "," + me.getY();
42        tf.setText(msg);
43    }
44 //continued...
```

Mouse Events Example

```
45      public void mouseMoved(MouseEvent me) {
46          String msg = "Mouse moved at " + me.getX()
47                       + "," + me.getY();
48          tf.setText(msg);
49      }
50      public static void main(String args[]) {
51          MouseEventsDemo med =
52                  new MouseEventsDemo("Mouse Events Demo");
53          med.launchFrame();
54        }
55  }
```

Close Window Example

```
1  import java.awt.*;
2  import java.awt.event.*;
3
4  class CloseFrame extends Frame
5                    implements WindowListener {
6     Label label;
7     CloseFrame(String title) {
8        super(title);
9        label = new Label("Close the frame.");
10       this.addWindowListener(this);
11    }
12 //continued...
```

Close Window Example

```
13     void launchFrame() {
14         setSize(300,300);
15         setVisible(true);
16     }
17     public void windowActivated(WindowEvent e) {
18     }
19     public void windowClosed(WindowEvent e) {
20     }
21     public void windowClosing(WindowEvent e) {
22         setVisible(false);
23         System.exit(0);
24     }
25 //continued...
```

Close Window Example

```
26      public void windowDeactivated(WindowEvent e) {
27      }
28      public void windowDeiconified(WindowEvent e) {
29      }
30      public void windowIconified(WindowEvent e) {
31      }
32      public void windowOpened(WindowEvent e) {
33      }
34      public static void main(String args[]) {
35          CloseFrame cf =
36                  new CloseFrame("Close Window Example");
37          cf.launchFrame();
38       }
39  }
```

Adapter Classes

- Why use Adapter classes?
  - Implementing all methods of an interface takes a lot of work
  - Interested in implementing some methods of the interface only

- Adapter classes
  - Built-in in Java
  - Implement all methods of each listener interface with more than one method
  - Implementations of the methods are all empty

Adapter Classes: Close Window Example

```
1  import java.awt.*;
2  import java.awt.event.*;
3
4  class CloseFrame extends Frame{
5     Label label;
6     CFListener w = new CFListener(this);
7
8     CloseFrame(String title) {
9        super(title);
10       label = new Label("Close the frame.");
11       this.addWindowListener(w);
12    }
13 //continued...
```

Adapter Classes: Close Window Example

```
14      void launchFrame() {
15          setSize(300,300);
16          setVisible(true);
17      }
18
19      public static void main(String args[]) {
20          CloseFrame cf =
21                  new CloseFrame("Close Window Example");
22          cf.launchFrame();
23       }
24  }
25  //continued...
```

Adapter Classes: Close Window Example

```
25 class CFListener extends WindowAdapter {
26      CloseFrame ref;
27      CFListener( CloseFrame ref ){
28         this.ref = ref;
29      }
30
31      public void windowClosing(WindowEvent e) {
32            ref.dispose();
33            System.exit(1);
34     }
35 }
```

Inner Classes

- Class declared within another class
- Why use inner classes?
  - Help simplify your programs
  - Especially in event handling

Inner Classes: Close Window Example

```
1  import java.awt.*;
2  import java.awt.event.*;
3
4  class CloseFrame extends Frame{
5     Label label;
6
7     CloseFrame(String title) {
8        super(title);
9        label = new Label("Close the frame.");
10       this.addWindowListener(new CFListener());
11    }
12 //continued...
```

Inner Classes: Close Window Example

```
13     void launchFrame() {
14        setSize(300,300);
15        setVisible(true);
16     }
17
18     class CFListener extends WindowAdapter {
19        public void windowClosing(WindowEvent e) {
20           dispose();
21           System.exit(1);
22        }
23     }
24  //continued...
```

Inner Classes: Close Window Example

```
25    public static void main(String args[]) {
26        CloseFrame cf =
27                new CloseFrame("Close Window  Example");
28        cf.launchFrame();
29    }
30 }
```

Anonymous Inner Classes

- Unnamed inner classes
- Why use anonymous inner classes?
  - Further simplify your codes
  - Especially in event handling

Anonymous Inner Classes: Close Window Example

```
1  import java.awt.*;   import java.awt.event.*;
2  class CloseFrame extends Frame{
3     Label label;
4     CloseFrame(String title) {
5        super(title);
6        label = new Label("Close the frame.");
7        this.addWindowListener(new WindowAdapter(){
8           public void windowClosing(WindowEvent e){
9              dispose();
10             System.exit(1);
11          }
12       });
13    }
```

Anonymous Inner Classes: Close Window Example

```
14      void launchFrame() {
15          setSize(300,300);
16          setVisible(true);
17      }
18
19      public static void main(String args[]) {
20          CloseFrame cf =
21                  new CloseFrame("Close Window Example");
22          cf.launchFrame();
23       }
24  }
```

Summary

- The Delegation Event Model
  - Register listeners

    ```
    void add<Type>Listener(<Type>Listener listenerObj)
    ```

  - Listeners wait for an event to occur
  - When event occurs:
    - Event object created
    - Object is fired by source to registered listeners
  - When listener receives event object:
    - Deciphers notification
    - Processes the event

Summary

- The Delegation Event Model Components
  - Event Source
  - Event Listener/Handler
  - Event Object

- Event Classes
  - The *EventObject* Class
  - The *AWTEvent* Class
    - Root of all AWT-based events
    - Subclasses follow this naming convention:

      `<Type>Event`

Summary

- Event Listeners
  - *ActionListener* Method
  - *MouseListener* Methods
  - *MouseMotionListener* Methods
  - *WindowListener* Methods

Summary

- Creating GUI Applications with Event Handling
  1. Create a GUI class
  2. Create a class implementing the appropriate listener interface
  3. In the implementing class
     - Override ALL methods of the appropriate listener interface
     - Describe in each method how you would like the event to be handled
  4. Register the listener object with the source
     - Use the *add<Type>Listener* method

- Simplifying your code:
  - Adapter Classes
  - Inner Classes
  - Anonymous Inner Classes