Events and Listeners

Java event model, semantic and low-level events, event handlers, ActionListeners/ActionEvents, WindowListeners/WindowEvents, anonymous inner classes

Java event model

- Whenever a user interacts with the GUI for an application such as moving a mouse or pressing a key, the component or hardware/ software participant "fires" an **event**.
- Events are objects that use **event listeners** to describe the source of the event to other objects. Any object that requires to be notified about an event must implement the appropriate listener interface and **register** the listener.
- The root of the event class hierarchy is EventObject. It defines a method **Object getSource()** that returns the object that fired the event. GUI event classes are in the package java.awt.event.
- Events fall into two broad categories:
  - **Low-level events:** triggered by low level input or interactions with the operating system, including key presses and mouse actions.
  - **Semantic events:** all the rest, including such things as button clicks, text entry, and moving sliders.

Event handlers

- An **event handler** is an object that can respond to an event. Every event handler must satisfy three conditions:
  1. The event handler class must either *implement a listener interface* or extend a class that implements a listener interface. For example:

     ```
     public class MyClass implements ActionListener {
     ```

  2. If a component is going to fire the event, there must be *code that registers an instance of this event handler class as a listener for the component*. For example:

     ```
     myButton.addActionListener(this);
     ```

  3. The event handler class must *implement the methods in the listener interface.* For example:

     ```
     public void actionPerformed(ActionEvent e) {
        //code that responds to the action...
     ```

ActionListener and ActionEvent

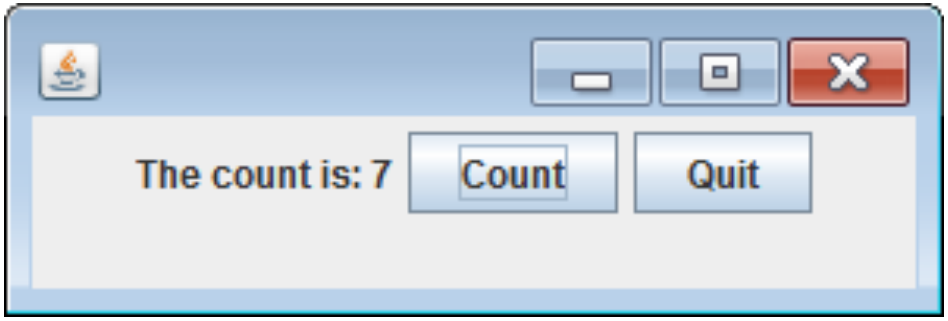
- **ActionListener** is a listener interface that extends EventListener.
- ActionListener declares only one method:

  void actionPerformed(ActionEvent e)
- An **ActionEvent** is an event (object) that indicates that a component-defined action occurred (such as pressing a button). The event is passed to every ActionListener object that registered to receive such events using the component's **addActionListener** method:
  `myButton.addActionListener(this)`
- ActionEvent has a method **getSource** that returns the source of the event (button, menu, etc).
- The implementation of actionPerformed typically follows this scheme:

```
void actionPerformed(ActionEvent e) {
   if (e.getSource().equals(myButton) {
      doSomething ();
   }
   else if (e.getSource().equals ….
```

Semantic events example

- An ActionEvent is triggered by a user action on a component -- button, list, menu item, textfield, and so on.



- The GUI here has a label and two buttons.
- Clicking *Count* increases the count total (shown in the label).
- Clicking *Quit* exits the program.

Semantic events code – Part 1

```
public class EventDemo extends JFrame implements ActionListener {

    // Variables for initializing strings
    private int count = 0;
    private final String MESSAGE = "The count is: ";
    // Components
    private JLabel lblCount = new JLabel(MESSAGE + count);
    private JButton btnCount = new JButton("Count");
    private JButton btnQuit = new JButton("Quit");

    public EventDemo(){
        Container c = getContentPane();
        c.setLayout(new FlowLayout());
        c.add(lblCount);
        c.add(btnCount);
        c.add(btnQuit);
        btnCount.addActionListener(this);
        btnQuit.addActionListener(this);
        setSize(300,100);
        setVisible(true);
    }
```

Semantic events code – Part 2

```
public class EventDemo extends JFrame
                        implements ActionListener {
    // Instance variables, constructor here

     public void actionPerformed(ActionEvent e) {
         if (e.getSource().equals(btnQuit))
             System.exit(0);
         if (e.getSource().equals(btnCount))
             lblCount.setText(MESSAGE + ++count);
     }
     public static void main(String[] args){
         new EventDemo();
     }
}
```

Semantic events code discussion

- The class declaration shows implementing the ActionListener interface.
- These two statements register the two buttons to be notified of ActionEvents:

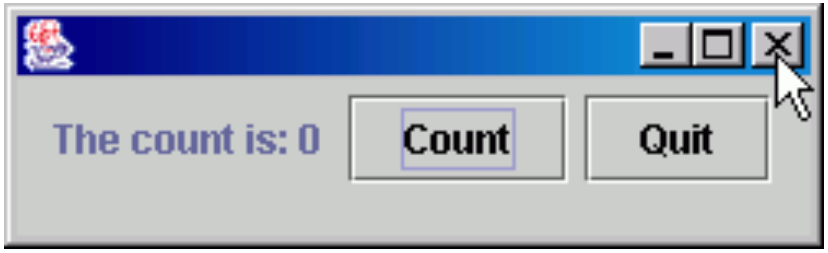
```
btnCount.addActionListener(this);
btnQuit.addActionListener(this);
```

- The method addActionListener() is defined in Component. The actual argument can be any ActionListener.
- actionPerformed() indicates what to do as a result of each event (button click for this application). It checks the source of each event to see which button was clicked.

```
if (e.getSource().equals(btnQuit))
```

Low-level events

- Low-level events are not component-generated. They include key presses or releases, mouse clicks/drag/move/release, and so on.



- Low-level event listeners include **WindowListeners** and **MouseListeners**.
- A **WindowEvent** is triggered by a user action on a window, such as closing the window, iconifying it, etc.

Low-level event code – Part 1

This code is a modification of EventDemo. It shows how to quit the program when the window is closed. The only thing different is the name of the class and WindowListener code.

```
public class WindowEventDemo extends JFrame implements
                                ActionListener, WindowListener {
    // Data declarations go here

    public WindowEventDemo() {
        // The old EventDemo constructor code is here
        addWindowListener(this);
    }
    public void actionPerformed(ActionEvent e) {
        // And this code doesn't change at all
    }
    // WindowListener methods go next
```

Low-level event code – Part 2

All we care about is what happens when the window is closed.
The rest of the methods are required to implement WindowListener.

```
public void windowClosing(WindowEvent e) {
    System.exit(0);
}
public void windowClosed(WindowEvent e) {}
public void windowActivated(WindowEvent e) {}
public void windowDeactivated(WindowEvent e) {}
public void windowOpened(WindowEvent e) {}
public void windowIconified(WindowEvent e) {}
public void windowDeiconified(WindowEvent e) {}

// Rest of program as usual here
```

Low-level event code discussion

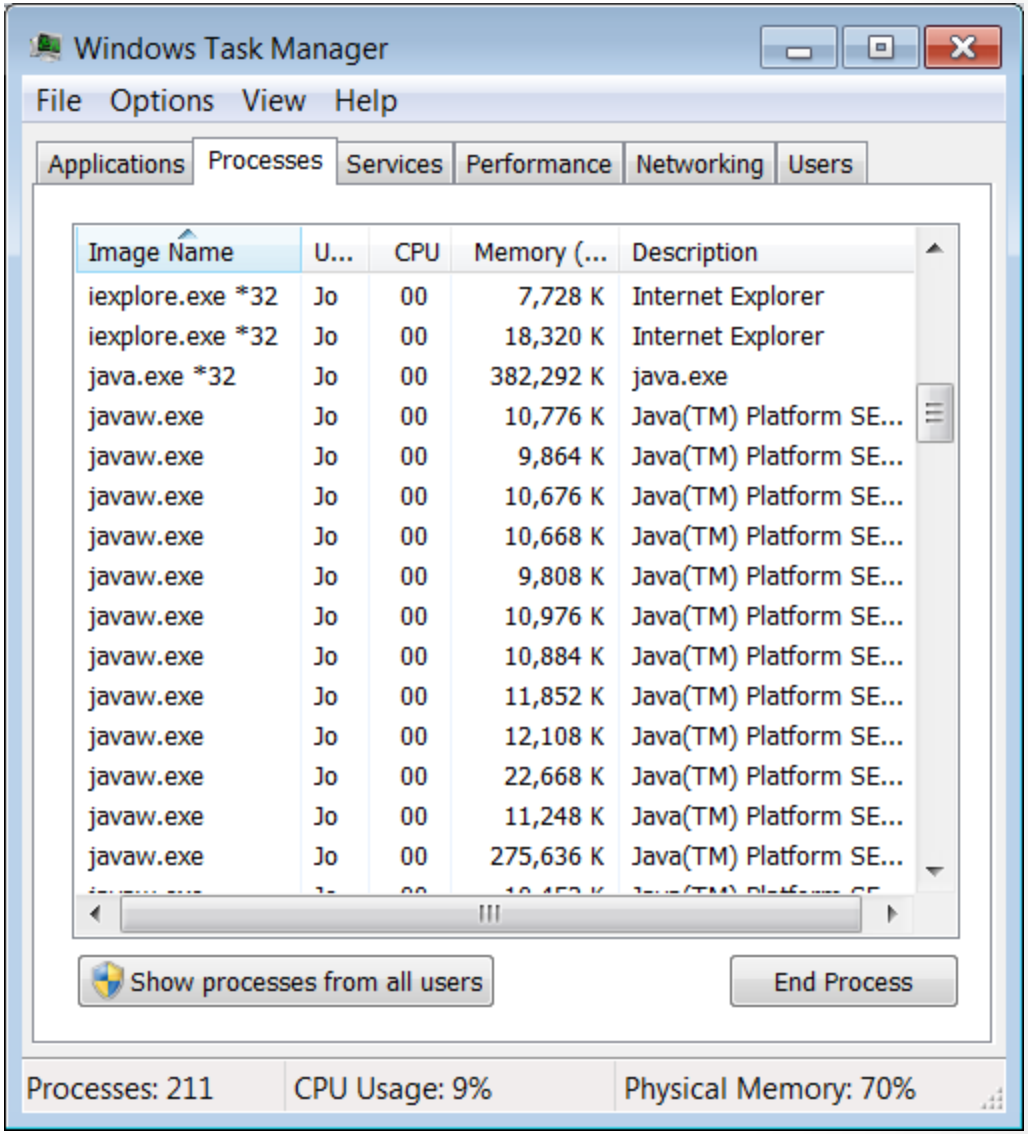
- EventWindowDemo implements WindowListener.
- The window (which is the application in this case) registers the WindowListener with the statement in the constructor:

```
addWindowListener(this);
```

- The methods windowClosed ... windowDeiconified are defined in order for WindowEventDemo not to be an abstract class.
- These extra methods are stubbed out – they do nothing.
- There is only one method of interest, windowClosing.

Why does it matter?

- Clicking the window close box doesn't automatically stop a program from running.
- The figure shows lots of java processes from lots of experiments with simple java programs.
- Except for eclipse or other well-behaved apps, there's no way to kill them nicely.



Adapters

- The code in the previous section demonstrated the problem of implementing WindowListener directly. Most of the WindowListener methods were irrelevant to the application, but they still needed to be defined.
- An **adapter** is a class that implements an interface, using stubs for all the methods that are declared in the interface. The WindowAdapter class is an adapter for the WindowListener interface. The MouseAdapter class is an adapter for the MouseListener interface.

Using adapters

- Since an adapter already defines all the methods in the interface, a class could "use" the adapter by extending it. This is uncommon, however, since that would prohibit the class from extending another class (for example JFrame).
- The following version of the application identical to the one in the previous section shows how to use an adapter by creating an **anonymous inner class**. This is a class with no name, and it is defined as part of the actual parameters for a method call.
- In this case, the anonymous inner class is the code defined in the call to addWindowListener().

Anonymous inner classes

This code has “unnamed” WindowAdapter as an anonymous inner class. It’s defined in the parameter list.

```
public class AnonymousEventDemo extends JFrame
                                 implements ActionListener {
   // Instance variables go here
   public AnonymousEventDemo() {
     // The original constructor contents here
     addWindowListener(
        new WindowAdapter() {
           public void windowClosing(WindowEvent e) {
              System.exit(0);
           }
        });
   } // End of constructor
   // Rest of code goes here
```

Anonymous inner class discussion

- Notice that the three "characterizations" for listeners appear absent from this code. In particular:
  - AnonymousEventDemo does *not* implement WindowListener.
  - There are no additional WindowListener methods defined outside the constructor.
- The code new **WindowAdapter()** indicates that what follows is the definition of an unnamed class that ***is-a*** WindowAdapter.
- The anonymous inner class overrides the WindowAdapter definition of windowClosing, which is just a stub.

A quick and clean alternative

If all you want to do is make the GUI program quit execution when its window is closed, simply put this statement in the constructor for its main JFrame:

```
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```