Event handling in Java

Like people, processes cannot always foresee the consequences of their actions.

Java divides the realm of the unpredictable into exceptions and events.

As we have seen, exceptions happen synchronously with a process; it is the immediate consequence of some action, such as division by 0.

An event happens asynchronously, often as a result of something the user does. The Java GUI is event driven; user actions initiate events.

We have already seen one kind of event—closing a window—in Lecture 13. Other events are initiated by the user, for example,

- moving the mouse,
- clicking a mouse button,
- pressing a key, or
- resizing a component.

Events are sent to a Java program by the underlying windowing system.

Information about event is stored in an instance of a subclass of EventObject. Most of the events we will see are AWTEvents.

Here is the hierarchy of AWTEvents.

```
java.lang.Object
 java.util.EventObject
  java.awt.AWTEvent
   java.awt.event.ActionEvent
   java.awt.event.AdjustmentEvent
   java.awt.ItemEvent
   java.awt.ComponentEvent
    java.awt.ContainerEvent
     java.awt.FocusEvent
```

Outline for Lecture 16

I. Event handling
   Events
   Listener interfaces
   Adapters
   Types of events
      Buttons
      Keyboard
      Mouse
      Item

II. Inner classes
   Definition
   A FixedStack
   An event listener
To get an event processed, the programmer must write code to do two things:

- register an *event listener*, and
- implement an *event handler*.

**Listener interfaces**

In our example from Lecture 13, we used

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

This could also be done by having the `HelloWorldSwing` class implement a `windowClosing` method to exit the system:

```java
public void windowClosing(WindowEvent e) {
    System.exit(0);
}
```

To do this, `HelloWorldSwing` would have to implement the `WindowListener` interface.

*Exercise:* What other changes would we have to make in the code for the example?

`WindowListener` is an example of an event-listener interface.

In general, for each leaf `Event` class there is a corresponding listener. Two exceptions are

- `MouseEvent`, which has two listeners, `MouseListener` and `MouseMotionListener`, and
- `PaintEvent`, which is a special event type used to ensure that `paint/update` method calls are serialized properly with other events.

Now, let's list the `Listener` interfaces.
Adapters

Some interfaces, e.g., `WindowListener`, have many methods:

```java
public void windowOpened(WindowEvent e)
public void windowIconified(WindowEvent e)
public void windowDeiconified(WindowEvent e)
public void windowClosed(WindowEvent e)
public void windowActivated(WindowEvent e)
public void windowDeactivated(WindowEvent e)
```

We could just implement all of these methods in a class like `HelloWorldSwing`.

But, because this is tedious, the `java.awt.event` package defines adapters for most of the `Listener` interfaces.

Adapters are classes that implement all of the methods in their corresponding interfaces, with null bodies.

If the programmer only wants to define a few methods of the interface, (s)he can extend an adapter class, and override a few of its methods.

However, as we shall see, this usually means that an object can no longer serve as a handler for its own events.

Types of events

Let us take a look at a few components and the events that they generate.

**Button events**

A button generates a *button event* when a user clicks it with a mouse.

Buttons are usually created with a one-parameter constructor. This parameter specifies the button’s label:

```java
JButton myButton = new JButton("OK");
```
An `ActionListener` receives events from the button. Here is a simple code segment that defines a button and processes its event:

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

public class OKButton extends JFrame {
    private JButton myButton;

    public void init() {
        myButton = new JButton("OK");
        myButton.addActionListener(
            new ButtonListener(this));
        add(myButton);
    }
}

class ButtonListener implements ActionListener {
    JFrame myFrame;

    public ButtonListener(JFrame f) {
        myFrame = f;
    }

    public void actionPerformed(ActionEvent e) {
        myFrame.setTitle("Button was pressed!");
    }
}
```

Note that the `ActionListener` interface is implemented. This is because clicking a button with a mouse generates an `ActionEvent`.

When a button is pressed and released,

- the AWT sends an instance of `ActionEvent` to the button, by calling `processEvent` on the button.
- the button’s `processEvent` method calls `processActionEvent`.
- `processActionEvent` passes the action event on to any `ActionListener`s that have registered an interest in events from this button. It does this by calling the `ActionListener`’s `actionPerformed` method.

In the code above, it is an instance of `ButtonHandler` that is interested in events from `myButton`. 
• myButton adds an instance of ButtonListener as its ActionListener.

• Conversely, ButtonListener keeps track of which frame it is listening for, by taking the frame as a parameter to its constructor.

This makes it possible for the ButtonListener to perform an action involving the frame (specifically, uses setTitle to display a string in the title bar at the top of the frame.

**Keyboard events**

Pressing or releasing a key generates a keyboard event.

The KeyListener interface recognizes three kinds of events:

• keyPressed(KeyEvent) is called when an action key is depressed. Action keys include arrow keys, function keys, Home, End, Page Up, Page Down, Num Lock, Print Screen, Scroll Lock, Caps Lock, and Pause.

• keyTyped(KeyEvent) is called when any other key is depressed.

• keyReleased(KeyEvent) is called after any key is released that has generated either of the above two events.

Four methods are useful in figuring out which key has been typed:

• int getKeyCode() returns the integer key code associated with the key in this event (only for keyPressed events).

• char getKeyChar() returns the character associated with the key (if any).

• String getKeyText(int keyCode) returns a string describing the keyCode, such as “PAUSE” or “F1”.

• String getKeyModifiersText(int modifiers) returns a string describing the modifier key(s), such as “Shift” or “Ctrl+Shift”.

A sample program that captures and processes key events is found at:
public class KeyEventDemo ... implements KeyListener ... {
    ...//where initialization occurs:
    typingArea = new JTextField(20);
    typingArea.addKeyListener(this);
    ...//** Handle the key-typed event from the text field. */
    public void keyTyped(KeyEvent e) {
        displayInfo(e, "KEY TYPED: ");
    }
    ...//**Handle the key-pressed event from the text field.*/
    public void keyPressed(KeyEvent e) {
        displayInfo(e, "KEY PRESSED: ");
    }
    ...//**Handle the key-released event from the text field.*/
    public void keyReleased(KeyEvent e) {
        displayInfo(e, "KEY RELEASED: ");
    }
    ...
    protected void displayInfo(KeyEvent e, String s) {
        ...
        char c = e.getKeyChar();
        int keyCode = e.getKeyCode();
        int modifiers = e.getModifiers();
        ...
        tmpString = KeyEvent.getKeyModifiersText(modifiers);
        ...//display information about the KeyEvent...
    }
}

Mouse events

There are two interfaces for dealing with mouse events, 
MouseListener and MouseMotionListener.

MouseListener defines these methods:

- mousePressed(MouseEvent) Invoked when a mouse button is pressed with the mouse cursor over a component.
- mouseClicked(MouseEvent) Invoked when a mouse button is pressed and released over a component without the mouse being moved.
- mouseReleased(MouseEvent) Invoked when a mouse’s button is released after the mouse has been dragged. This event is always preceded by a MouseDragged event.
• `mouseEntered(MouseEvent)` Invoked when the mouse cursor enters the bounds of a component.

• `mouseExited(MouseEvent)` Invoked when the mouse cursor moves outside the bounds of a component.

`MouseMotionListener` defines these methods:

• `mouseDragged(MouseEvent)` Invoked when a mouse button is pressed while the mouse is over a component, and then dragged. Always preceded by a `MousePressed` event.

• `mouseMoved(MouseEvent)` Invoked when the mouse is moved with the cursor is over a component, without any buttons being pressed.

What is the sequence of events when a mouse is dragged?

What is the difference between `MouseClicked` and `MouseReleased`?

Among other things, each `MouseEvent` object contains the x and y coordinates where the event occurred. These values can be retrieved via the `int getX()` and `int getY()` methods of the mouse event.

On some systems, mice have more than one button. `MouseEvent` inherits methods from `InputEvent` to allow it to distinguish among the buttons.

• `isMetaDown()` returns true when the user clicks the right button on a mouse with two or three buttons.

  How do you think this might be simulated on a one-button mouse?

• `isAltDown()` returns true when the user clicks the middle button on a three-button mouse.
How do you think this might be simulated on a one- or two-button mouse?

For examples of mouse listeners, see http://java.sun.com/docs/books/tutorial/ui/components/mouselistener.htm.

In general, documentation on events can be found at http://java.sun.com/docs/books/tutorial/uiswing/overview/event.html

**Item events**

Several of the Swing classes implement the interface ItemSelectable, which indicates that they contain a set of items from which zero or more items may be selected.

Among these classes are JButton, JCheckBox, JRadioButton, and JComboBox.

- A checkbox is a box that may be selected or deselected. Any number of checkboxes in a group may be selected.
- “Radio buttons” are a set of buttons of which only one can be selected at the same time. Such a set of buttons are members of the same ButtonGroup.
- A combo box, sometimes called a drop-down list or dropbox, allows a user to select an item from the list.

Choosing an item generates an ItemEvent. The ItemEvent class contains a method getItemSelectable() to get the ItemSelectable object that generated the item event. Whenever an item’s state changes, the itemStateChanged() of the ItemListener class is called.

Here is a way that such an event might be processed, assuming that selected is a Boolean array that tells for each item whether it has been selected:

```java
class ItemEventHandler implements ItemListener {
    ...
    public void itemStateChanged(ItemEvent e) {
        Choice chosen = (Choice) e.getItemSelectable();
```
selected[chosen.getSelectedIndex()] = true;
}

**Inner classes**

Languages like Algol, PL/1 and Pascal allow a programmer to nest procedures within procedures.

The idea behind this was that certain names in the program need to be known only in the vicinity where they are defined.

From the beginning, Java adopted this idea of *limited scope* for variables and methods, but not for packages.

In Java 1.0, all classes were *top level*; all classes were defined at the same level. If a class was known anywhere in a source file, it was known throughout the file.

Java 1.1 allows classes to be nested within other classes. A class defined inside another class is known as an *inner class*.

Inner classes are useful for at least two reasons:

- The name of the inner class is known only within its scope. Thus, it does not “pollute” the namespace of the package.
- The code of an inner class can refer directly to names from enclosing scopes, including both class and instance variables and methods, and local variables of enclosing blocks.

**A Fixed Stack**

Here is an example taken from the java tutorial documentation, a class `FixedStack` which implements a stack, and defines an enumerator of elements from the stack, from top to base:

```java
public class FixedStack {
    Object array[];
    int top = 0;
    FixedStack(int fixedSizeLimit) {
        array = new Object[fixedSizeLimit];
    }
```
public void push(Object item) {
    array[top++] = item;
}
public boolean isEmpty() {
    return top == 0;
}
// other stack methods go here...

/** This adapter class is defined as part of its
   * target class. It is placed alongside the
   * variables it needs to access.
   */
class Enumerator implements java.util.Enumeration {
    int count = top;
    public boolean hasMoreElements() {
        return count > 0;
    }
    public Object nextElement() {
        if (count == 0)
            throw new NoSuchElementException("FixedStack");
        return array[--count];
    }
    public java.utilEnumeration elements() {
        return new Enumerator();
    }
}

The interface java.util.Enumeration is used to communicate a series of values to a client.

Note that FixedStack does not directly implement the Enumeration interface. Why not?

Thus, a separate adapter class is used to return the elements one by one.

Note that the adapter class needs to access the array containing the stack elements.

It can directly refer to instance variables of the stack, since the adapter is defined inside FixedStack.
Inner classes for adapters

Since an AWT component and its event listener are closely related, it is helpful to use inner classes for event adapters too. A useful example is found at http://java.sun.com:81/docs/books/faq/src/awt/EventListenerExample.java.

It contains code for implementing a WindowListener and a MouseMotionListener. Here are some excerpts:
import java.awt.*;
import java.awt.event.*;
import java.swing.*;

public class EventListener11Example {

    /** An inner class that closes the window and
     *  quits the app. */
    class WindowQuitter extends WindowAdapter {
        public void windowClosing(WindowEvent e) {
            e.getWindow().dispose();
            System.exit(0);
        }
    }

    Frame mainFrame = new JFrame("EventListener11Example");

    /** Performs initialization, builds the interface. */
    public void init() {
        /* Initialize event handlers. */
        mainFrame.addWindowListener(new WindowQuitter());

        /* Show the main window. */
        mainFrame.setBounds(100, 100, 500, 200);
        mainFrame.show();
    }
}

Sometimes it is not necessary to define a named class (like
WindowQuitter) at all. If the code for a class is very simple, it can be
declared in line with the add...Listener method.

Here is an example of how this could be done in the HelloWorldSwing
example from the start of Lecture 13:

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