

CS1Bh Lecture Note 13

Event-driven programming

The best practice in user-interface code development is the *event-driven programming* model in which code fragments can be associated with particular *events* which may occur in code use. In object-oriented event-driven programming it is class methods which are attached to particular events.

In this lecture note we will consider some simple GUI driven applications. In order to understand their code, we first need to introduce a number of new Java programming concepts.

13.1 GUI Components and Containers

Real graphical user interface (GUI) driven application programs don't build their interfaces directly in terms of the simple graphics operations of drawing lines, boxes and so on. Instead, they exploit pre-written class libraries of GUI components such as buttons, lists and menus. Our example programs will use the facilities provided by the Swing package. These inherit many properties from classes further up the Java hierarchy. In particular, `java.awt.Component` and `java.awt.Container` respectively capture the concepts of an element of a GUI (such as a clickable button), and of those GUI elements which can themselves contain other GUI elements (such as a panel of buttons). All containers are themselves components.

13.2 GUI Layout Managers

As well as defining the logical content of a GUI (which buttons and menus it contains, and what they do), we need to specify the way in which these will be laid out on the screen. Java provides a number of Layout Manager classes which handle this aspect for us. For example, the `GridLayout` class knows how to arrange components into a rectangular style of display, while `BorderLayout` places components around the edges of a central GUI area (as might be useful in an editor interface, for example).

13.3 Events and Listeners

Events (such as mouse clicks) are represented in Java programs by objects and are subclasses of `java.util.EventObject`. Those events which are associated with Java's Abstract Windowing Toolkit (the AWT) are subclasses of `java.awt.AWTEvent`. The package `java.awt.event` makes a collection of these. When the user generates an event (for example, by clicking the mouse on a button), the run-time system creates a corresponding event object and presents it to the program.

The process by which the program deals with the event is called *event handling* and links an *event listener* (the code which knows what to do with the event) to an *event source* (the GUI component in which the event was generated). An event listener adds itself to a list maintained by the source and is notified whenever an event occurs.

Event listeners extend the interface `java.util.EventListener`. For example, the interface `ActionListener` specifies a single method `ActionPerformed` which must deal with the `ActionEvent` object it receives as its parameter. The `ActionEvent` object can be inspected via its own methods to find out various things about it (such as its source GUI component).

```
package java.awt.event; // 1
import java.util.EventListener; // 2
public interface ActionListener // 3
    extends EventListener { // 4
    void actionPerformed (ActionEvent e); // 5
} // 6
```

13.4 The ColourDemo Program

We now pull together the concepts we have seen so far into a small program. This provides the user with the ability to draw simple coloured shapes at fixed position in the GUI by clicking on buttons corresponding to each shape. There is also a button which can be clicked to clear the drawing area.

Lines 6 to 12 create the components and containers which the GUI needs. These are added to the GUI in the constructor. Notice how the individual buttons are first added to the `JPanel` component `buttons` (lines 19-20), which is then added (line 26) to the GUI. Lines 21-24 make the connection between the buttons (as potential event sources) and the `ColourDemo` object (which is itself an event listener), with a reference to **this**.

The event handling code is provided by method `actionPerformed` (as required by the `ActionListener` interface). This uses the event's `getSource` method to determine which button has been clicked, then takes appropriate action with the `Graphics` object which controls the appearance of the `Canvas` drawing area.

```

import java.awt.*; // 1
import java.awt.event.*; // 2
import javax.swing.*; // 3
public class ColourDemo extends JFrame // 4
    implements ActionListener { // 5
    Canvas canvas = new Canvas(); // 6
    JPanel buttons = new JPanel (); // 7
    JButton b1 = new JButton("Blue Square"); // 8
    JButton b2 = new JButton("Red Rectangle"); // 9
    JButton b3 = new JButton("Yellow Circle"); // 10
    JButton b4 = new JButton("Clear"); // 11
    Container pane; Graphics canvasg; // 12
    // 13
    public ColourDemo() { // 14
        super ("Some Buttons, Actions and Colours"); // 15
        pane = getContentPane(); // 16
        pane.setLayout (new GridLayout(2,1)); // 17
        buttons.setLayout (new GridLayout(1,3)); // 18
        buttons.add(b1); buttons.add(b2); // 19
        buttons.add(b3); buttons.add(b4); // 20
        b1.addActionListener(this); // 21
        b2.addActionListener(this); // 22
        b3.addActionListener(this); // 23
        b4.addActionListener(this); // 24
        canvas.setBackground(Color.green); // 25
        pane.add(buttons); pane.add(canvas); // 26
        setSize(600,400); setVisible(true); // 27
        canvasg = canvas.getGraphics(); // 28
    } // 29
    // 30
    public void actionPerformed (ActionEvent e) { // 31
        if (e.getSource() == b1) { // 32
            canvasg.setColor(Color.blue); // 33
            canvasg.fillRect(250,25,100,100); // 34
        } else if (e.getSource() == b2) { // 35
            canvasg.setColor(Color.red); // 36
            canvasg.fillRect(200,50,200,50); // 37
        } else if (e.getSource() == b3) { // 38
            canvasg.setColor(Color.yellow); // 39
            canvasg.fillOval(250,25,100,100); // 40
        } else if (e.getSource() == b4) { // 41
            canvas.repaint(); // 42
        } // 43
    } // 44
    public static void main (String[] Args) { // 45
        ColourDemo c = new ColourDemo(); // 46
        c.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // 47
    } // 48
} // 49

```

13.5 Adapter Classes

In the AWT, event listeners are Java interfaces. It is convenient to have adaptable classes which implement these interfaces but ignore all of the events because it can be easier to provide a subclass for the relevant *event adapter* than to implement the interface directly. Often this is because only one particular kind of event is of interest. In the `java.awt.event` package, the interface `MouseEventAdapter` and the class `MouseEventAdapter` form such a pair of a listener and an adapter. A mouse motion listener provides two methods. The first is to be invoked whenever the mouse is *dragged* (moved while the button is held down). The second is to be invoked whenever the mouse is moved. Other listener interfaces can specify many more methods. The corresponding adapter class (e.g. `MouseEventAdapter`) provides a “do nothing” default implementation of all methods. We can then extend this with our own class which overrides just those methods we care about, without having to write dummy “do nothing” implementations for the others.

13.6 Anonymous local classes

It is common to want to write an event handler which is specific to events generated by exactly one component in a GUI (and so which will be instantiated only once). Rather than requiring us to give such one-off classes names, and define them elsewhere in the file, Java provides a mechanism which allows these to be defined where they are used (i.e. *locally*) and without names (i.e. *anonymously*), so that they definitely can't be instantiated anywhere else. The syntax for this manoeuvre is illustrated in the following example, and is one of the most commonly recurring patterns in event-driven programming.

13.7 The Scribbler Example

We consider an adaptation of the scribble program from the O'Reilly book *Java in a Nutshell*. This implements a very simple drawing program, which allows the user to drag the mouse to draw in a graphics area, and to click a button to start again. The “Clear” button is created and handled by the code between lines 30 and 39 and added to the GUI on line 43.

The listener is created by asking for a new `ActionListener` object (even though this is actually an interface rather than a class). Our intentions are made clear by the following block which defines our method to override `actionPerformed`, clearing away the previous scribble. All of this takes place within the parameter to the call of `addActionListener` which links the button to this new handler.

Similar code (extending appropriate adapter classes) is used on lines 10 to 14 (to capture the mouse press which indicates the start of a new scribble) and lines 18 to 28 (to handle a mouse drag which must be matched by marks on the drawing area).

Murray Cole, 17th February 2003.

```

import java.awt.*; // 1
import java.awt.event.*; // 2
import javax.swing.*; // 3
// 4

public class Scribble extends JFrame { // 5
    int last_x, last_y; Container pane; // 6
    // 7

    public Scribble () { // 8
        super ("Scribble Pad Demo"); // 9
        this.addMouseListener(new MouseAdapter() { // 10
            public void mousePressed(MouseEvent e) { // 11
                last_x = e.getX(); last_y = e.getY(); // 12
            } // 13
        }); // 14
        // 15

        // Define, instantiate and register // 16
        // a MouseMotionAdapter object // 17
        this.addMouseMotionListener(new MouseMotionAdapter() { // 18
            public void mouseDragged(MouseEvent e) { // 19
                // Draw a line on the screen if the mouse has // 20
                // been dragged. Ignore other mouse movement. // 21
                Graphics g = getGraphics(); // 22
                int x = e.getX(); int y = e.getY(); // 23
                g.setColor(Color.blue); // 24
                g.drawLine(last_x, last_y, x, y); // 25
                last_x = x; last_y = y; // 26
            } // 27
        }); // 28
        // 29

        Button b = new Button("Clear"); // 30
        b.addActionListener(new ActionListener() { // 31
            public void actionPerformed(ActionEvent e) { // 32
                // clear the scribble // 33
                Graphics g = getGraphics(); // 34
                g.setColor(getBackground()); // 35
                g.fillRect(0, 0, getSize().width, // 36
                    getSize().height); // 37
            } // 38
        }); // 39
        pane = getContentPane(); // 40
        pane.setBackground(Color.white); // 41
        pane.setLayout (new FlowLayout()); // 42
        pane.add(b); setSize(400,400); setVisible(true); // 43
    } // 44

    public static void main (String[] Args) { // 45
        Scribble s = new Scribble(); // 46
        s.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // 47
    } // 48
} // 49
// 50

```

Here is another demo program which illustrates that things are not always what they might appear to be....

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

public class ClickToWin extends JFrame {
    int centreX=400, centreY=400;
    Font bigFont;

    public ClickToWin () {
        super ("Annoying Demo");
        this.addMouseListener(new MouseMotionAdapter() {
            public void mouseMoved (MouseEvent e) {
                boolean moved = false;
                while (e.getX()>centreX-200 && e.getX()<centreX+200 &&
                    e.getY()>centreY-50 && e.getY()<centreY+50) {
                    centreX = (int) (200 + (500 * Math.random()));
                    centreY = (int) (50 + (500 * Math.random()));
                    moved = true;
                }
                if (moved) repaint();
            }
        });
        bigFont = new Font ("TimesRoman", Font.ITALIC, 36);
        setSize(800,800); setVisible(true);
    }

    public void paint (Graphics g) {
        g.setColor(Color.white);
        g.fillRect(0, 0, 800, 800);
        g.setColor(Color.yellow);
        g.fillRect(centreX-125, centreY-25, 250, 50);
        g.setFont(bigFont);
        g.setColor(Color.black);
        g.drawString("Click to Win!", centreX-110, centreY+16);
        g.drawRect(centreX-125, centreY-25, 250, 50);
        g.drawRect(centreX-120, centreY-20, 240, 40);
    }

    public static void main (String[] Args) {
        ClickToWin s = new ClickToWin();
        s.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```