Chapter 2

Event Based Programming Basics

2.1 Introduction

This chapter introduces you to the fundamentals of event based programming. Obviously, if we are going to discuss programming, we will have examples, and those examples will be implemented in some language. We chose Java. Java has a clean event model where each class or interface plays a particular role and where the various components work together nicely to form an entire application. Since, Java is (mostly) platform independent, you may work through all our examples under Windows, or Linux, or on a Mac – on almost any desktop computer.

Graphical User Interfaces (GUIs) are also a good way to introduce the nuts and bolts of event based programming. GUI components such as menus, check boxes, radio buttons, and text boxes communicate with an application via events. The purpose of this chapter is not to study GUI programming in depth. GUI programming deserves a complete textbook, and there are already many good texts on the topic [Johnson, 2000, Walrath et al., 2004]. Our purpose is to illustrate various aspects of event based programming. We use GUIs as the domain.

Java includes two packages for GUI development, the Abstract Window Toolkit (AWT) and Swing. Our examples use Swing. Swing is the newer package and the more popular of the two. By the time you finish this chapter, you will be familiar with some of the basic Swing classes, including: JFrame, JButton, JLabel, JTextField, JScrollPane, JList, and JPanel, as well as the Swing menu classes. You will also understand the events that Swing components use to interact with the application. You will not have mastered Swing, however. If you are interested in pursuing Swing programming more thoroughly, you are encouraged to pick up a good Swing programming text [Elliott et al., 2002, Deitel and Deitel, 2007, Deitel et al., 2002].

All our examples are relatively short and may be coded using any text editor. However, all source code examples may also be downloaded from the text’s web site.

2.1.1 Integrated Development Environments

If you intention is to build professional quality GUIs quickly, the authors recommend using an Integrated Development Environment (IDE). There are several good Java IDEs, NetBeans and Eclipse, for example, and a variety of plugins that allow programmers to develop Swing programs using drag and drop techniques. These are not discussed here, as they hide many of the details of event based programming in which we are interested.
2.2 The Object–Oriented Event Model

The fundamental building blocks of object–oriented programs are classes and objects. Object–oriented event based programs are no exception. The event source (an object) fires an event which is processed by the event handlers (other objects). Each handler processes the event by manipulating application objects. Figure 2.1 shows the relationship among the various pieces.

As we will see in later chapters, designing event sources poses some complex issues. We will avoid these issues in this chapter by letting the Swing classes serve as the sources. Swing components are the programmatic representations of the sources for input events such as mouse clicks and keyboard presses. Of course, a particular physical device (mouse or keyboard) is really the “source” of the event, and that this low level event propagates upward to the Swing application. We pick up the event processing when the Swing component realizes that the event has occurred. For more on the upward propagation of events, see Chapter 5.

The event handlers, which execute when the event fires, are application specific. That is, each button in each application has handlers designed explicitly for it. Assuming that you have already implemented the data structures and other application classes with which your GUI will interact, coding a Swing GUI becomes a three step process:

1. Identify the GUI components to be used and lay out the GUI interface.

2. Code the event handlers that go behind the GUI components\(^1\).

3. Register the handlers with the GUI components.

2.2.1 A Simple Example

A simple example can serve to clarify these ideas by putting them into a concrete context.

In this example there is a single button. When the button is clicked, a message box is opened. The main GUI interface is shown in Figure 2.2.

The resulting message box is shown in Figure 2.3.

\(^1\) A GUI component such as a button has a visible attribute that appears to a user on the screen, and clicking on the button activates program code (the handler) that is invisible to the user. We think of the button as hiding the handler, or that the handler is “behind” the button, away from the view.
This is a very simple event driven Java program. It contains a single button, that when clicked pops open a message dialog.

Written by: Stuart Hansen
Date: September 2008

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

public class ClickMe extends JFrame {
    JButton button; // Our one and only button

    public ClickMe () {
        // The button is our event source
        button = new JButton("Click me");

        // We register a handler with the source
        ActionListener handler = new ActionHandler();
        button.addActionListener(handler);

        // We add the button to the viewable area of the window
        getContentPane().add(button);

        // We set a couple of window properties and then open the main window
        setSize(100, 100);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }

    // A very simple main method
    public static void main (String args[]) {
        new ClickMe();
    }

    // The handler class.
    class ActionHandler implements ActionListener {
        public void actionPerformed (ActionEvent e) {
            JOptionPane.showMessageDialog(null, "Ouch! That hurt.");
        }
    }
}
The `javax.swing` package includes the Swing classes for windows, buttons, textboxes, etc. The `java.awt.event` package is also needed, as it includes many of the event classes.

The class `ClickMe` inherits from `JFrame`. `JFrame` is Swing’s window class. By inheriting from it, our program gets a main window in which we develop our GUI.

The `JButton` is the only GUI element in our window.

The `ClickMe` constructor in which we instantiate our objects, register the handler with source and set a few properties.

This instantiates the button.

`handler` is an `ActionHandler` object. The handler is responsible to responding to button clicks. `ActionHandler` is a separate class, whose source code is at the end of the file.

`handler` is registered with the button at runtime. Note that this is an example of polymorphism or dynamic binding, a central theme in event based programming.

The button is added to the `JFrame`. Swing requires that we add the button to the `contentpane` of the application rather than directly to the `JFrame`. This separates the handling of the GUI components from the other windowing responsibilities, e.g. closing and resizing.

The `JFrame` is sized to 100x100 pixels.

The application should exit when the `JFrame` closes. If this line is omitted, you may close the GUI window, but the application will continue to run in the background.

The `JFrame` is set to be visible.

The main method. As with many GUI and other event driven programs, the main method’s body shrinks to a single line of code that instantiates the application object. In our application line 33 constructs a new `ClickMe` object.

The `ActionHandler` class. It contains one method, `actionPerformed()` which is invoked when the button is clicked. The method opens a message window.

![Message Box](image.png)

Figure 2.3: The message box opened by clicking the button.

Many beginning Java programmers do not realize that it is legal, and many times desirable, to place multiple Java classes within one file, as we done here. The only restriction Java places on us is
that at most one class may be public: this must be the class containing the main() method. In our program we place the ActionHandler class within the ClickMe.java file, because it is only used by this application.

The ActionHandler class implements the ActionListener interface. Objects implementing this interface must have a method with the particular signature:

\[
\text{public void actionPerformed (ActionEvent e)}
\]

Since our handler is registered with the button (line 21), this actionPerformed method is called whenever the button is clicked. Our particular actionPerformed method displays a message dialog box saying ”‘Ouch! That hurt.’”

While this example is very simple, it illustrates many of the basic features of Java event based systems. It has an event source (the JButton) and an event handler (the ActionHandler). It registers the handler with the source at runtime. The handler contains a method with a specific signature (\text{actionPerformed(ActionEvent e)}) that is called when the button is clicked. We should also note that the event object, ActionEvent e, is always passed to the handler, but in our example e is never used.

2.3 Java Language Features to Support Event Based Programming

There are several features of Java that support event based programming. Among them are anonymous classes and inner classes. The next several examples show how these features can be used to make our programs more elegant.

2.3.1 Anonymous Classes

Most Java programming students are familiar with anonymous objects. An anonymous object is one that is instantiated, but never assigned to a variable. For example, in the ClickMe program above, the main() method instantiates a ClickMe object, which starts the entire program running, but that object is never assigned. It remains anonymous.

Anonymous classes are similar. A class is created, but never given a name. Two conditions should be true before using an anonymous class:

1. There must be only one place in the code where an object of this type is instantiated. We will define the anonymous class and instantiate objects at this location.

2. The anonymous class should only contain one or at most two short methods that are being defined or overridden. If the class is longer, the code will be much more readable if defined as a named class.

Anonymous classes are useful to define event handlers, because handlers generally meet these conditions. Often an event handler is only referenced when it is instantiated and registered with the source (condition 1). Similarly, a handler only needs to define the methods expected by the listener interface. In our above example, this was actionPerformed(). Thus, event handlers make ideal candidates for anonymous classes.
ClickMe using Anonymous Classes

Here is the ClickMe program again, using an anonymous handler.

```java
public class ClickMeAgain extends JFrame {
    JButton button; // Our one and only button

    public ClickMeAgain () {
        // The button is our event source
        button = new JButton("Click me");

        // Register the handler
        button.addActionListener(new ActionListener() {
            public void actionPerformed (ActionEvent e) {
                JOptionPane.showMessageDialog(null, "I said, \"Don’t do that.\"");
            }
        });

        // We add the button to the viewable area of the window
        getContentPane().add(button);

        // We set a couple of window properties and then open the main window
        setSize(100, 100);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }

    public static void main (String args[]) {
        new ClickMeAgain();
    }
}
```

**Lines Commentary**

18–22 An anonymous instance of an anonymous handler. We construct an `ActionListener` followed by a pair of braces, { and }, and we define the handler methods within them. In this program, the only method defined this way is `actionPerformed()`. Note that `ActionListener` is an interface, but because we define `actionPerformed()`, we can instantiate it. Anonymous classes can also be derived from other classes. This is useful if we want to specialize a previously defined handler by overriding one or two methods.

The remainder of the code is virtually unchanged from the previous example. The only difference is that the named class, `ActionHandler`, is gone – replaced by the anonymous class shown above.
2.3.2 Inner Classes

Many modern object-oriented languages, including C++, C# and Java allow a programmer to define inner classes. These are classes defined inside of another class. Inner classes are useful in several different situations. If class \( A \) takes complete responsibility for all instances of class \( B \), it is appropriate to define class \( B \) as a private inner class, inside of \( A \). This prevents other classes from attempting to access \( B \) directly. For example, linked lists contain nodes for which the list is completely responsible. Nodes should not be seen outside of the list. They serve no purpose outside of the list. We can define the Node class as a private inner class to the LinkedList class. The linked list will have complete access to the nodes, but nodes will be completely invisible outside the list.

Inner classes also provide improved scoping. The scope of a variable or method is where it is visible. In object-oriented languages, instance variables and methods are declared private, public, or protected. This determines the variable's visibility. Good software engineering practice tells us to declare our variables as private so that they are only visible within the class where they are declared. We then grant access to them via public methods that manipulate them only in controlled ways.

In modern object-oriented languages, instances of an inner classes have access to all members of the encapsulating class, including private members. This means that an event handler defined as an inner class can call private methods and access private data in the encapsulating/outer class, as needed.

2.3.3 List Example using Inner Classes

This example illustrates inner classes, as well as several additional Swing classes. The objective of the program is to display a list of strings that the user enters. The GUI is shown in Figure 2.4. The user types text in the input field following the prompt. When she presses enter, the text is added to the list.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

// ListDemo illustrates the use of JLabels, JTextFields, JScrollPanes and JLists.
// Written by: Stuart Hansen
// September 2008

public class ListDemo extends JFrame {
    // These elements form the GUI of the application
    private JLabel enterLabel;
    private JTextField enterField;
    private JScrollPane listPane;
    private JList list;

    // The constructor
    public ListDemo () {
        super ("List Demo");
        enterLabel = new JLabel();
        enterField = new JTextField();

        // The DefaultListModel on the next line gives us the
        // ability to add to the list
        list = new JList(new DefaultListModel());

        // The list is added to a JScrollPane, so that we may
        // view lists larger than the display
        listPane = new JScrollPane(list);

        // setup the contentPane to use absolute coordinates
        Container contentPane = getContentPane();
        contentPane.setLayout(null);

        // initialize and add the components to the GUI
        enterLabel.setText("Enter some text");
        enterLabel.setSize(100, 30);
        enterLabel.setLocation(20, 20);
        contentPane.add(enterLabel);

        enterField.setSize(100, 20);
        enterField.setLocation(120, 25);
        contentPane.add(enterField);
        enterField.addActionListener(new InputHandler());

        listPane.setSize(100, 200);
        listPane.setLocation(20, 60);
        contentPane.add(listPane);
    }
}
// Set the windows size and close operation
setSize(300, 300);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// display the application window
setVisible(true);
}

// The ActionListener for the JTextField
// The event if fired when enter is pressed
// The text is added to the list
private class InputHandler implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        String text = enterField.getText();
        DefaultListModel model = ((DefaultListModel)(list.getModel()));
        if (!text.equals("")) {
            model.addElement(text);
            enterField.setText("");
        }
    }
}

// a very simple main program
public static void main(String args[]) {
    new ListDemo();
}
13–16 Declare the various Swing components for this example.

13 Declares the JLabel that prompts the user.
14 Declares the JTextField where the user enters the strings.
15–16 The components on lines 15 and 16 work together. JLists are not scrollable, so when using a JList it is almost always best to wrap it in a JScrollPane before adding it to the application. Scroll bars will then appear automatically if the list grows larger than its viewable area.

19–57 Define the application’s constructor.

21–30 Instantiate the various GUI components.

26 Pass an instance of DefaultListModel to the JList’s constructor. It sounds silly, but DefaultListModel is not the default list model for JLists, so we need to pass one to the constructor, if we want to use it in place of the real default. In our application, the main advantage of using the DefaultListModel is that it makes it easier to add objects to the list.

30 Wrap the JList inside of the JScrollPane as discussed above.

33–34 Set the JFrame’s contentpane and set its layout manager to null. Layout managers control the location and size of components in a GUI. There are a number of good layout managers, but any discussion of them is beyond the scope of this chapter. Instead, by setting the layout manager to null, we tell the application to use the sizes and locations that we set explicitly in the code.

36–49 Set the sizes, locations and a few other properties of the various GUI components and add each to the JFrame.

52–56 Set a few JFrame properties and show the application window.

62–71 Define the handler for the JTextField. An ActionEvent is generated when the user presses enter while the focus is on the field. As in the previous example, this handler implements the ActionListener interface. Conceptually, the actionPerformed() method is straightforward. We get the text from enterField, and if it isn’t null add it to the list. There are a number of details we should note:

- The handler is defined as a private inner class. This means that the class and instance of the class are not visible beyond the ListDemo class.

- actionPerformed() accesses data and methods in the ListDemo class. It calls two methods from enterField and one method from list. These are private data members of ListDemo, but are visible to the handler because it is an inner class.

- All Swing components, including JList use a model–view–controller (MVC) architecture. In the next section we discuss MVC in more detail. For now, note that the data is stored in the model portion of the component. Therefore the handler must call getModel() before it can add to (or delete from) the list.
2.4 Inheritance

Inheritance is the primary way of reusing existing code while specializing it to the needs of a specific application. A derived class inherits from a base class. It automatically gets all the data and methods already defined in the base class. The programmer can then add more data and methods, and override methods that already exist.

Inheritance can play a major role in developing GUIs and other event base programs. Languages come with large GUI APIs that contain buttons and sliders and all the types of components that go into a typical GUI. Each has a standard appearance and user interactions, e.g. a GUI button looks like a button, and computer users all know that you click it to make things happen. By contrast, a label displays information to the user and the user doesn’t expect anything to happen if they click on it.

An easy way to build complex GUIs is to extend existing GUI classes so they appear and behave the way you need. Let’s consider two examples:

- In our previous Java example, we displayed a list of strings. The list was displayed in the order the strings were entered. A simple specialization would be to maintain the list in alphabetical order.

- As another example, consider developing a drawing program. The user will create a picture by dragging the mouse. The program displays the figure in real time, as it is created.

In both cases, we start with an existing class and extend it for some particular purpose. We will develop Java implementations of each of these after a bit more discussion.

2.4.1 Model–View–Controller

GUI components are developed using a Model–View–Controller (MVC) approach. MVC is way of thinking about GUI components and programs that divides their implementation into three parts:

- a model – containing the data,
- a view – visually presenting the data, and
- control – through which other objects and the user interact with the data.

The component wraps the model, view and controller into a single object.

The model, view and controller are tightly coupled, because the component can’t exist without all three, but the coupling occurs in very specific ways.

- Control updates the model and on occasion may directly tweak the view.
- The view depends on the model for the data to display.
- Control code, at least in Java, frequently does not exist as an independent object, but consists of the methods and handlers within the component that change the component’s state.

The model, view and controller remain loosely coupled in the sense that each one has distinct responsibilities and can be modified or replaced without requiring changes to the other two. Model, view and control each have specific responsibilities, and as long as they live up to their responsibilities their internal functioning is independent of the other two. For example, in the ListDemo code above, the JList was assigned a new model, the DefaultListModel, without requiring any changes to the list’s view or control.
The Sorted List Example

This example is identical to the previous one, except that the elements in the list are maintained in lexicographic order. In the earlier example we showed how to replace a JList’s model with an instance of DefaultListModel. In this example we will use a specialized list model of our own design.

ListModel is a Java interface. Any object that implements the interface could be used as the model for a JList. Our SortedListModel extends DefaultListModel which, in turn, implements ListModel. This way we will only need to override methods of interest. The rest we inherit from DefaultListModel. We choose to only override the addElement() method, modifying the code so that the list is maintained in sorted order. As noted in the code’s comments, better code would also override all other methods that can add data to the list, or modify elements already in the list, but for the sake of brevity this is not done.
import javax.swing.*;

/* SortedListModel.java replaces the default list model with one
that keeps the list is sorted order.
* 
* Written by: Stuart Hansen
* September 2008
*/

public class SortedListModel extends DefaultListModel {
    // We override addElement in the DefaultListModel class
    // so that the JList remains sorted.
    // Note that in a more complete application, add() 
    // set() and setElementAt() should also be overridden.

    public void addElement(Object obj) {
        String str = obj.toString();
        int index = getSize();
        while (index > 0 && ((String) elementAt(index - 1)).compareTo(str) >= 0) {
            index--;
        }
        super.add(index, obj);
    }
}

Lines    Commentary
14–21    The new addElement() method uses a linear search to find the correct lexicographic
location for the element. The while loop starts at the end of the list and moves
backwards toward the first element, until it finds the correct location or reaches the list’s
beginning. The call to super.add() is to the add method in the DefaultListModel
which inserts the element at that location.

We do not show the remainder of the code for this example, because only one other line is
changed, the line that instantiates the JList. A new SortedListModel() is passed to the JList
constructor instead of the DefaultListModel. Figure 2.5 shows the modified program during a run.

ListModelEvents

The astute reader will notice that our revised code did not change the events that fired or the control
code. Only the model changed. The external events of GUI components control the interaction
between the user and application. Our modified program did not touch this code. We only changed
the model, which modified how the data was stored and indirectly how it was displayed.

How then did the JList know that it needed to update its view when an element was added?
The answer is that it uses internal events. An internal event is one whose source and handler both
reside in the same component. Only the model, view and control of the component need be aware
of these.

In our example, the list model fired a ListDataEvent to the ListModelListeners whenever the
model changed.

Because our new model inherited all the infrastructure to register and fire ListDataEvents from
DefaultListModel. Our new addElement() method fired the event within its call to super.add().
No explicit event firing was needed on our part.
The JList’s view listened for the events and updated itself appropriately. This approach also works well if we have multiple views of the same model, or when changes to the model propagate to other non-view objects\(^2\).

### A Drawing Application

In this section we present another example of events working with MVC, creating a drawing application. The user draws in a window by pressing the left button and dragging the mouse within the window, see Figure 2.6.

The application is built around a specialized JPanel, named DrawPanel. Developing the DrawPanel provides an excellent example of extending a Swing component with redefined and augmented model, view and control.

JPanels are containers. A container is a component whose primary role is to hold other components. That is, programmers add buttons, textboxes and other components to them, and then later add the container to a window/frame. Containers don’t have much specialized functionality, making them an ideal base class for extending when creating a new specialized component.

The more general question of which component a programmer should extend to create a new, specialized component depends a lot on the programmer’s needs and the language/API being used. The programmer should try to maximize the amount of code that can be reused. If you want to be able to click on the component, a button is an obvious choice. If you want to be able to type text into the component, a textbox is the obvious choice. If you want to define something completely new and different, then choosing a component with minimal pre-existing functionality, e.g. a JPanel in Java, makes good sense.

We start by presenting our DrawPanel, then show how it can be used in a complete application.

---

\(^2\)It is possible in Java to register other listeners for ListDataEvents. This is illustrated in the List of Doubles example, later in this chapter.
import javax.swing.*;
import javax.swing.plaf.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

/**
* DrawPanel implements a JPanel that can be drawn on using
* the mouse
* @author Stuart Hansen
* @version September 2008
*/

// We specialize the JPanel to contain a drawing.
public class DrawPanel extends JPanel {
    // The model used for the JPanel is a list of curves.
    ArrayList<Curve> listOfCurves;

    // Two additional state variables to aid us in creating the drawing
    Curve currentCurve;  // the current curve being drawn
    Color currentColor;  // the current drawing color

    public DrawPanel() {
        super();

        // Initialize the model
        reset();

        // replace the view
        setUI(new DrawPanelUI());

        // add specialized control
        addMouseListener(new MouseHandler());
        addMouseMotionListener(new MouseMotionHandler());
    }

    // A setter for the color
    public void setColor(Color col) {
        currentColor = col;
    }

    // a getter for the color
    public Color getColor() {
        return currentColor;
    }

    // reset the drawPanel model
    public void reset() {
        listOfCurves = new ArrayList<Curve>();
        currentColor = Color.BLACK;
        repaint();
    }
}
16 Declare `DrawPanel` to extend `JPanel`. By extending `JPanel` we get all the functionality already associated with it. The other advantage of inheriting from `JPanel` is that it also lets us use a `DrawPanel` wherever a `JPanel` would be permitted. This makes it easy to add `DrawPanels` to `JFrames` or other containers.

18–22 Declare the model variables for the `DrawPanel`. The drawing is made up of a collection of curved lines (poly-lines), which we store in an `ArrayList`. `Curve` is a private inner class which is discussed below.

The other two variables, `currentCurve` and `currentColor`, are useful while creating the drawing. Each time a new curve is started `currentCurve` is updated. `currentColor` is updated when `setColor()` is called.

24–36 The constructor follows a fairly standard pattern for components that extend Swing components. It does base class initialization, then initializes the model, view and control.

25 `super()` initializes the base class, in this case `JPanel`. It is always a good idea to explicitly initialize the base class with a call to `super`. It is required in Java if you want to pass parameters to the base class constructor.

28 `reset()` initializes the data model portion of the component. The model initialization code is placed in a separate method, `reset()`, so that the drawing can also be re-initialized later, not just when the application is started.

31 `setUI()` sets the user interface to our specialized view. `DrawPanelUI` is discussed below.

34–35 Mouse handlers are added specifying how the mouse will be used to make a drawing. `MouseHandler` and `MouseMotionHandler()` are discussed separately below.

39–46 `setColor()` and `getColor()` are self explanatory.

49–53 `reset()` re-initializes the model portion of the `DrawPanel`. The `listOfCurves` is set to an empty list and the `currentColor` is reset to black. The call to `repaint()` directs Java to repaint the `DrawPanel`. Repainting in Swing is asynchronous, much like event handling. That is, `repaint()` does not do the repainting. Instead, it directs Java to repaint the component at its earliest convenience. There are ways to force immediate repainting, but `repaint()` is almost always the more appropriate method to call.

Note that this approach is distinct from the event based approach used in the previous example. We do not fire a `JPanelDataEvent` as that event does not exist. The result of calling `repaint()` is similar, however. The `DrawPanel` is redrawn on the screen.

The `DrawPanel` class is short because it delegates the responsibility for doing much of the work to private the model, view and control. For example, a programmer using the `DrawPanel` class does not need to know how a curve is represented, so the `Curve` class is naturally a private inner class. Similarly, the low-level details of displaying the drawing are best kept private. Again, a private inner class is ideal.
Modeling a Curve

Curve is a private inner class to DrawPanel. It is used to model individual curved lines in the drawing. Each curve contains a color and a list of points on the curve.

```java
55 // Each curve has a color and a list of points.
56 // The points form a series of line segments, so it is really a poly-Line,
57 // not a true curve.
58 // We use an inner class to model the curve.
59 private class Curve {
60     private Color color; // the color of the curve
61     private ArrayList<Point> points; // the points on the curve
62
63     // the constructor initializes the color and the list
64     public Curve (Color c, Point p) {
65         color = c;
66         points = new ArrayList<Point>();
67         points.add(p);
68     }
69
70     // get the Color
71     Color getColor() {
72         return color;
73     }
74
75     // returns an iterator over the points
76     public Iterator<Point> iterator() {
77         return points.iterator();
78     }
79
80     // adds a Point
81     public void add (Point p) {
82         points.add(p);
83     }
84 }
```

60–61 The Curve class contains two data elements, the curve’s color and a list of points. Joining the points forms a series of very short line segments, a.k.a. a poly-line. The segments are so short, however, that the curve appears smooth to the naked eye when rendered.

64–68 The constructor initializes the color and the ArrayList. Because it creates a new curve when the mouse button is pressed, there will be one point in the curve when it is constructed, the location where the initial button press occurred.

70–83 All of the methods in the class get or modify data values. This is very common in model classes, as their primary purpose is to hold the data.

Populating our model with curves is the business of the control code, discussed later.
Figure 2.6: An example of a DrawPanel. The image was created by holding down the left mouse button and dragging the mouse.

Replacing the View

There are several ways to update the view of a component in Java. An elegant and object-oriented way is to update the view object, a.k.a. the ComponentUI for the component. Swing delegates responsibility for rendering a component to its ComponentUI object. To create a new view, we specialize ComponentUI and override its paint() method. As seen in the constructor above, we then assign the component a new ComponentUI using the setUI() method.

```java
86 // The DrawPanelUI class knows how to display the drawing
87 private class DrawPanelUI extends ComponentUI {
88     public void paint (Graphics g, JComponent c) {
89         // We iterate across the list of curves, drawing each
90         Iterator<Curve> curveIterator = listOfCurves.iterator();
91         while (curveIterator.hasNext()) {
92             // We iterate across each curve drawing it
93             Curve curve = curveIterator.next();
94             Iterator<Point> pointIterator = curve.iterator();
95             while (pointIterator.hasNext()) {
96                 // Set the color for this curve
97                 g.setColor(curve.getColor());
98                 // Iterate across the Points rendering the line segments
99                 Point p1 = pointIterator.next();
100                while (pointIterator.hasNext()) {
101                    Point p2 = pointIterator.next();
102                    g.drawLine((int)p1.getX(), (int)p1.getY(),
103                                 (int)p2.getX(), (int)p2.getY());
104                    p1 = p2;
105                }
106         }
107     }
108 }
109 }
110 }
111 }
```
This is the most complex code of the entire application. The drawing is rendered using nested loops. The outer loop iterates across all the curves in the drawing. Each curve is rendered by first setting its color and then iterating across its points drawing the poly-line as we go.

88 Override ComponentUI’s paint() method. It takes two parameters, a Graphics object and a reference to the component we are painting. The details of working with Graphics objects are beyond the scope of this text. The class contains over 40 different methods, most of which are related to rendering.

98, 105–106 Our paint() method only uses two methods from Graphics, setColor() on line 98, and drawLine() on lines 105–106. The semantics of each of these should be intuitively clear. Further documentation on the Graphics class can be found in the Java API documentation.

Note that there are several places in the view code that directly access the DrawingPanel’s model, both when getting the iterators and when invoking setColor() and drawLine(). Again, because DrawPanelUI is a private inner class, it has direct access to these data and methods.

Mouse Input Handlers

Our final private inner classes are the event handlers. Java separates mouse events into those associated with moving the mouse and those associated with pressing mouse buttons, so we need two handlers for the mouse input.

An event adapter implements all the methods of an event listener with empty method bodies. The notion of an empty method might strike some as strange. The method is called, does nothing and returns. However, adapters are useful because a handler may inherit from the adapter class and override only the subset of the methods needed for the particular application. Figure 2.7 shows how adapters fit into the basic event handling architecture.

Some event sources only fire one type of event, e.g. a JMenuItem only fires an ActionEvent. In this case, the interface only contains one method declaration and there is no need for an adapter because there are no "extra" methods.

The need for adapters arises because the Java event classes sometimes represent multiple closely related events. For example, the MouseEvent class represents a number of different mouse related events, including: mouse entered, mouse exited, mouse pressed, mouse released and mouse clicked. The MouseListener interface contains a method declaration for each of these. If a component is only interested in mouse clicked events, its mouse handler inherits from the MouseAdapter class and overrides the mouseClicked() method. When a mouse clicked event is fired by the source it is handled by the mouse handler. Other mouse events are passed up the inheritance hierarchy and handled by the mouse adapter’s empty methods.

Both the mouseListener and mouseMotionListener interfaces specify several methods, so we use adapter classes for both our handlers.
Figure 2.7: Java achieves decoupling between the event source and event handler by placing an interface and an adapter between them. The event listener specifies the methods that the event source expects in all handlers. The event adapter implements all of the methods of the interface with empty method bodies.

```java
113 // The following event handlers are part of the JPanel's control
114 // This handler adds points to the current vector
115 private class MouseMotionHandler extends MouseMotionAdapter {
116     public void mouseDragged (MouseEvent e) {
117         if (SwingUtilities.isLeftMouseButton(e)) {
118             currentCurve.add(e.getPoint());
119             repaint();
120         }
121     }
122 }
```

While the mouse is dragged, we add points to the curve.

115 declares our handler to be a subclass of MouseMotionAdapter. MouseMotionAdapter defines empty methods related to moving the mouse. We just override the one in which we are interested mouseDragged().

116-121 mouseDragged() is called when the user holds down any mouse button and moves the mouse. We only want to draw if it is the left mouse button, so the code includes an if statement checking this condition. The result is that points are added to the current curve when we drag with the left button pressed.
123 // When the mouse is first pressed a new curve is started
124 private class MouseHandler extends MouseAdapter {
125     public void mousePressed (MouseEvent e) {
126         if (SwingUtilities.isLeftMouseButton(e)) {
127             currentCurve = new Curve(currentColor, e.getPoint());
128             currentCurve.add(e.getPoint());
129             listOfCurves.add(currentCurve);
130         }
131     }
132 }
133 }

124 MouseHandler extends MouseAdapter for the same reasons as discussed above.
125–131 Our handler only overrides one method, mousePressed(). This handler begins a new curve.

2.4.2 The Drawing Application

The previous section developed a specialized component, DrawPanel. The DrawPanel is not a complete program, however. It must be added to a JFrame before it can be displayed. No special code is needed. It is just added to the JFrame in the same way we previously added buttons and textboxes. As we saw when developing the DrawPanel, however, it contains some public functionality like changing drawing colors and clearing the panel, that is only available by calling its methods. Dropdown menus are the ideal way to access these methods.

In this section complete our develop of a drawing program, using a JFrame with dropdown menus to display and manipulate a DrawPanel.

Swing Menu Classes

There are a number of classes associated with dropdown menus in Swing.

- There is a single JMenuBar per JFrame. It contains the menus and menu items.

- JMenus are added to the menu bar.

- JMenuItem are added to the menus. JMenuItem fire ActionEvents.

- Event handlers are registered with the menu items.

The menu for the application consists of

- a single menu, labeled 'File'

- three menu items in the File menu: 'Color', 'Clear' and 'Exit'
import javax.swing.*;
import javax.swing.plaf.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

/**
 * This class implements a drawing program in java
 *
 * @author Stuart Hansen
 * @version September 2008
 */

group public class DrawProgram extends JFrame {
    // These elements form the GUI of the application
    DrawPanel drawPanel = new DrawPanel();
    JMenuBar menuBar = new JMenuBar();
    JMenu menu = new JMenu("File");
    JMenuItem colorItem = new JMenuItem("Color");
    JMenuItem clearItem = new JMenuItem("Clear");
    JMenuItem exitItem = new JMenuItem("Exit");

    public DrawProgram () {
        super("Line Draw");

        // add the menu to the application Frame
        setJMenuBar (menuBar);
        menu.add(colorItem);
        menu.add(clearItem);
        menu.add(exitItem);
        menuBar.add(menu);

        // setup the drawPanel
        getContentPane().add(drawPanel);
        drawPanel.setBackground(Color.white);

        // Set the current Color
        drawPanel.setColor(Color.black);
    }

Lines    Commentary

16–21 Declare and instantiate all the components for the application, including the
        DrawPanel and all parts of the menu.

24 The call to super() initializes the JFrame.

25–30 Setup the menu for the application. We add the menu bar to the application, add
        the menu items to the menu, and and the menu to the menu bar.

33–38 Add a DrawPanel to the application and set a couple of its initial properties.
// Change the drawing color
colorItem.addActionListener ( 
    new ActionListener () { 
        public void actionPerformed (ActionEvent e) { 
            Color oldColor = drawPanel.getColor (); 
            Color newColor = JColorChooser.showDialog (null, 
                "Choose a new color", oldColor); 
            if (newColor != null) 
                drawPanel.setColor (newColor); 
        } 
    }; 
}

// Clear the drawing by replacing the DrawingPanel
clearItem.addActionListener ( 
    new ActionListener () { 
        public void actionPerformed (ActionEvent e) { 
            drawPanel.reset (); 
        } 
    }; 
)

// Exit the system elegantly
exitItem.addActionListener ( 
    new ActionListener () { 
        public void actionPerformed (ActionEvent e) { 
            System.exit (0); 
        } 
    } 
)

// Set the application window’s properties
setSize (400, 400); 
setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE); 

// display the application window
setVisible (true); 

// a very simple main program
public static void main (String args []) { 
    new DrawProgram (); 
}
As with buttons, menu items fire ActionEvents. We register an ActionListener with each menu item. In this example, we use anonymous inner classes for each of the ActionListeners.

The code within each handler method is relatively short. The handler for the Color menu opens a JColorChooser. If the user chooses a new color, that color is passed along to the drawPanel. If the user cancels, null is returned to the handler, in which case no new color is set.

The Clear handler clears the drawPanel’s model. Note that reset() contains a call to repaint(), so that when the model is cleared, the display is also cleared.

The handler for Exit, exits the application. System.exit() takes an integer parameter that encodes why the application terminated. Zero is the standard argument to mean that the application terminated normally.

The end of the constructor sets some main window parameters for the JFrame. The main method starts the program running by creating a new object of type DrawProgram.

### 2.5 List of Doubles

This section presents a complete, longer Java application that maintains a list of floating point numbers, Doubles in Java. The interface to the application is shown in Figure 2.8.

There are several very simple use cases:

- The user enters a number in the input field and presses Enter. The number is added to the list.

- The user may also select an element in the list and delete it by clicking Delete.

- The user may clear the entire list by clicking Clear All.

Whenever the list is updated, both the average and the maximum are recalculated and updated, as well. If the list becomes empty, the the average and the maximum are set to NaN, which stands for "Not a Number".

While this is an admittedly contrived application designed for pedagogic purposes, simple statistical applications similar to this one have many uses. Unfortunately for us, a spreadsheet will provide the needed functionality and more.

The example illustrates several of the more advanced points mentioned earlier in this chapter.

- Events are propagated through the system. E.g. pressing enter on the textfield updates the list, which in turn fires events that update the average and maximum. New averages and maximums update the display by again firing events.

- Like most programs, there are many things that can go wrong. We handle exceptions within our event handlers, printing error messages as needed. Figure 2.8 shows the error displayed when a user tries to enter non-numeric data into the list.
Figure 2.8: The user may add and delete numbers from the list. The application automatically updates the displayed average and maximum using event based techniques.
2.5.1 Double List Model

In the previous example we saw how we could specialize a JList's model. In that example we specialized it so that the list was maintained in lexicographic order.

In this example we also specialize the model, but in a different way. Here, only Doubles may be added to the list.

```java
import java.awt.DefaultListModel;

public class DoubleListModel extends DefaultListModel {
    // Overrides add element so that only Doubles are added
    public void addElement(Object obj) {
        Double d = Double.parseDouble(obj.toString());
        addElement(d);
    }

    // A specialized addElement for Doubles
    public void addElement(Double d) {
        super.addElement(d);
    }

    // Overrides toArray() so that the returned Array contains Doubles
    public Double[] toArray() {
        Object[] tempArr = super.toArray();
        Double[] dArr = new Double[tempArr.length];
        for (int i=0; i<tempArr.length; i++)
            dArr[i] = (Double) tempArr[i];
        return dArr;
    }
}
```
<table>
<thead>
<tr>
<th>Lines</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>The <strong>DoubleListModel</strong> class extends the <strong>DefaultListModel</strong> class overriding several methods in it.</td>
</tr>
<tr>
<td>14-17</td>
<td><strong>addElement(Object obj)</strong> is overridden. The method converts its <strong>Object</strong> parameter first to a <strong>String</strong> and then to a <strong>Double</strong>. In Java all objects have a <strong>toString()</strong> method, so this conversion is guaranteed to succeed. It then proceeds to convert the <strong>String</strong> to a <strong>Double</strong>. This code works correctly if the user has entered a valid <strong>Double</strong>. While not explicit in the code, Java throws a <strong>ClassCastException</strong> occurs if the conversion fails. Finally, we add the newly created <strong>Double</strong> to the list, using the overloaded <strong>addElement(Double d)</strong> method.</td>
</tr>
<tr>
<td>20-22</td>
<td>We also implement a specialized <strong>addElement(Double d)</strong> method that adds <strong>Doubles</strong> to the list. Note that we are still using the list data structure maintained by the super class, <strong>DefaultListModel</strong>. That list is a list of objects. By controlling access to it by overriding methods, we limit the values that may be added to just <strong>Doubles</strong>. <strong>DefaultListModel</strong> contains several other methods that add or modify values in the list. These include: <strong>add()</strong>, <strong>insertElementAt()</strong>, <strong>set()</strong> and <strong>setElementAt()</strong>. A more complete <strong>DoubleListModel</strong> class would also override these methods. Since they aren’t necessary for our example, they are ignored here.</td>
</tr>
<tr>
<td>25-31</td>
<td>Finally, we override <strong>DefaultListModel</strong>’s <strong>toArray()</strong> method so that the array returned is an array of <strong>Doubles</strong>. We will use <strong>toArray()</strong> to get the values for calculating the average and maximum. Overriding <strong>toArray()</strong> keeps us from having to cast each element to a <strong>Double</strong> after accessing it in the original array. As with the previous note, there are several other ”getter” methods that could be overridden, including: <strong>elementAt()</strong>, <strong>elements()</strong>, <strong>firstElement()</strong>, <strong>get()</strong>, <strong>getElementAt()</strong>, <strong>lastElement()</strong>, and <strong>remove()</strong>. Again, these are ignored, since they are not used by our application.</td>
</tr>
</tbody>
</table>
2.5.2 Average and Max Classes

The **Average** and **Max** classes model the two statistics displayed in the user interface. Whenever the average or the max is recalculated they fire a data changed event that notifies all listeners that they should take appropriate action. Note that from a practical point of view, we could have the average and maximum calculations done by methods within the application, not in separate classes, but that is not in the spirit of demonstrating data changed events.

The developers of Java Beans realized the importance of this type of event and included support for them in the Java Beans package. This support includes:

- **The PropertyChangeEvent Class**
  The `PropertyChangeEvent` class is used to encapsulate the information needed to process property changes. Each instance includes four pieces of information, the event source, the name of the property changed, the original value of the property, and the new value of the property. In java event objects are passive, containing the information needed to handle the events.

- **The PropertyChangeListener Interface**
  All classes wanting to receive `PropertyChangeEvent` s must implement the `PropertyChangeListener` interface. The interface contains a single method, `void propertyChange(PropertyChangeEvent evt)`. Event handling code goes in this method.

- **The PropertyChangeSupport Class**
  There are still two pieces of functionality missing for us to be able to use `PropertyChange` events. We need to be able to register handlers with sources and fire the events to each handler when the event occurs. The `PropertyChangeSupport` class contains numerous methods, but the two that are central to our discussion are: `addPropertyChangeListener()` and `firePropertyChange()`.

  To use this class we instantiate an instance of it for each property of interest. We then delegate responsibility for registering listeners and firing events to that instance.
import java.beans.*;

/**
 * The Average class finds and keeps track of the average of an array of Doubles.
 */
@Author Stuart Hansen
@Version September 2008
*/

public class Average {
    // We maintain the average in order to have an old average for the property change event
    private Double average;

    // We use pcs to facilitate our property change events
    private PropertyChangeSupport pcs;

    // The default constructor
    public Average() {
        average = new Double (Double.NaN);
        pcs = new PropertyChangeSupport (this);
    }

    // Return the current Average
    public Double getAverage() {
        return average;
    }

    // Return the current Average as a String
    public String toString () {
        return average.toString();
    }

    // Find the average of an enumeration of Doubles
    public Double findAverage(Double[] dArr) {
        Double oldAverage = average;
        if (dArr.length > 0) {
            double sum = 0.0;
            int count = 0;
            for (Double d : dArr) {
                sum += d;
                count++;
            }
            average = new Double (sum/count);
        } else
            average = Double.NaN;
        firePropertyChange (new PropertyChangeEvent (this, "average", oldAverage, average));
        return average;
    }
}
1 Import java.beans.*. This is the package that contains the PropertyChange classes.

16 Declare the PropertyChangeSupport object. The PropertyChangeSupport class contains methods that allow us to register listeners and fire events to them whenever the average is updated.

19–22 The constructor is quite simple. It initializes the average and the PropertyChangeSupport object.

35–53 Calculate the average of an array of Doubles.

50–51 An interesting part of this method is where it fires the PropertyChangeEvent. The PropertyChangeSupport class was designed for this purpose. We included the pcs object in our class and then use it to register listeners and fire events. The PropertyChangeEvent constructor takes four parameters, the event source, this; the name of the property changed, in our case average; the old value and the new value.

```java
// Fire a property change
public void firePropertyChange(PropertyChangeEvent e) {
    pcs.firePropertyChange(e);
}

// Add a property change listener
public void addPropertyChangeListener(PropertyChangeListener pcl) {
    pcs.addPropertyChangeListener(pcl);
}

// Delete a property change listener
public void removePropertyChangeListener(PropertyChangeListener pcl) {
    pcs.removePropertyChangeListener(pcl);
}
```

56–68 Place wrapper methods around some of the PropertyChangeSupport methods, facilitating public access to them.

The Max class directly parallels the code in the Average class and is omitted for the sake of brevity.

### 2.5.3 The Main Class

The DoubleListMain class builds the application from the previous classes, Swing components and handlers.
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import javax.swing.event.*;

/**
 * DoubleListMain puts together an application that records numbers
 * into a list and reports their average and max.
 *
 * @author Stuart Hansen
 * @version September 2008
 */

public class DoubleListMain extends JFrame {

    private JScrollPane pane; // scrollpane for the list
    private JList list; // the list's display
    private DoubleListModel model; // the model holding the list

    private JLabel add; // the label for the add field
    private JTextField inputField; // where the numbers are entered
    private JLabel errorLabel; // a label to display error messages

    private JLabel avgLabel; // where the average is displayed
    private Average avg; // calculates the average

    private JLabel maxLabel; // where the max is displayed
    private Max maximum; // finds the maximum

    private JButton del; // deletes the selected element
    private JButton clear; // clears the entire list

    // The constructor "wires" together the application
    public DoubleListMain() {
        Container cPane = getContentPane();
        cPane.setLayout(null);

        // Set up the list and its model
        model = new DoubleListModel();
        list = new JList();
        list.setModel(model);
        pane = new JScrollPane(list);
        pane.setSize(100, 150);
        pane.setLocation(100, 150);
        cPane.add(pane);

        // Set up the label for the add field
        add = new JLabel("Add");
        add.setLocation(50, 50);
        add.setSize(40, 30);
        cPane.add(add);

        // Set up the input textField
        inputField = new JTextField();
        inputField.setLocation(100, 50);
        inputField.setSize(100, 30);
        inputField.addActionListener(new AddHandler());
        cPane.add(inputField);
The application’s model consists of `model` – which contains the data from the list, `avg` – which contains the average of the data, and `max` – which contains the maximum of the data.

The GUI consists of the usual assortment of buttons, labels, textfields and lists.

The model is placed in the list. The list is placed in the scrollpane. The scrollpane is added to the contentPane.

We add a label and a textfield to the window. We register an `AddHandler` with the textfield as a method of adding to the list. All the handler code appears below.

We add the textfield used for input to the application.

```java
// Set up the label for error messages
errorLabel = new JLabel(" ");
errorLabel.setForeground(Color.red);
errorLabel.setSize(400, 30);
errorLabel.setLocation(50, 100);
cPane.add(errorLabel);

// Set up the delete button
del = new JButton("Delete");
del.setSize(100, 30);
del.setLocation(210, 50);
del.addActionListener(new DeleteHandler());
cPane.add(del);

// Set up the clear button
clear = new JButton("Clear All");
clear.setSize(100, 30);
clear.setLocation(320, 50);
clear.addActionListener(new ClearHandler());
cPane.add(clear);

// Set up the average value and label
avg = new Average();
avgLabel = new JLabel("Average = NaN");
avgLabel.setSize(200, 30);
avgLabel.setLocation(260, 150);
model.addListDataListener(new AverageAdapter());
avg.addPropertyChangeListener(new AvgPropHandler());
cPane.add(avgLabel);

// Set up the maximum value and label
maximum = new Max();
maxLabel = new JLabel("Maximum = NaN");
maxLabel.setSize(200, 30);
maxLabel.setLocation(260, 200);
model.addListDataListener(new MaximumAdapter());
maximum.addPropertyChangeListener(new MaxPropHandler());
cPane.add(maxLabel);

// Set a few windowing parameters and show the frame.
setSize(500, 400);
setDefaultCloseOperation(EXIT_ON_CLOSE);
setVisible(true);
```
60–64 We place an error message label under the input field. The message is empty, unless an error occurs.

67–78 We add two buttons to the application, one to delete individual items from the list and one to clear the entire list. The registered handlers are defined below.

81–96 We add the Average and Max objects to the application, along with their GUI representations.

85, 96 Add the ListDataListeners to the model. Events are fired from the model to these handlers when data is added or deleted from the model, updating the average and the maximum.

99–101 We set a few main window parameters and open the window.

2.5.4 Event Handler Classes

All the handlers are defined as inner classes to the application class. This gives them access to all the various application components they need to carry out their activities.

```java
// This class is the handler for adding numbers
private class AddHandler implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        try {
            String str = inputField.getText();
            model.addElement(str);
            inputField.setText(" ");
            errorLabel.setText(" ");
        }
        catch (Exception ex) {
            errorLabel.setText("Must only add numbers to the list.");
        }
    }
}

// This class is the handler for removing numbers
private class DeleteHandler implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        try {
            int index = list.getSelectedIndex();
            model.remove(index);
            errorLabel.setText(" ");
        }
        catch (Exception ex) {
            errorLabel.setText("Error! Use mouse to select element to remove");
        }
    }
}

// This class is the handler for clearing the list
private class ClearHandler implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        model.clear();
    }
}
```
The `addHandler` is called when the enter key is pressed in the textfield. The data in the textfield is added to the list. Note the `try -- catch` block. As we saw earlier, if the data we try to add, `str`, cannot be parsed into a `Double`, `addElement()` will throw an exception. This exception is caught and an appropriate error message is printed to the `errorLabel`.

The `DeleteHandler` deletes individual items from the list. We choose the item to delete by clicking on it with the mouse. We then click the `Delete` button, with which this handler is registered. The `getSelectedIndex()` method in the `JList` class returns the index of the item that has been clicked. We then remove it from the model. If no item has been clicked an exception is raised, which again is handled by printing a message to the `errorLabel`. 

46
private class AverageAdapter implements ListDataListener {
    public void contentsChanged(ListDataEvent e) {}
    public void intervalAdded(ListDataEvent e) {
        Double[] temp = model.toArray();
        avg.findAverage(temp);
    }
    public void intervalRemoved(ListDataEvent e) {
        Double[] temp = model.toArray();
        avg.findAverage(temp);
    }
}

private class MaximumAdapter implements ListDataListener {
    public void contentsChanged(ListDataEvent e) {}
    public void intervalAdded(ListDataEvent e) {
        Double[] temp = model.toArray();
        maximum.findMax(temp);
    }
    public void intervalRemoved(ListDataEvent e) {
        Double[] temp = model.toArray();
        maximum.findMax(temp);
    }
}

private class AvgPropHandler implements PropertyChangeListener {
    public void propertyChange(PropertyChangeEvent e) {
        avgLabel.setText("Average = " + avg.getAverage());
    }
}

private class MaxPropHandler implements PropertyChangeListener {
    public void propertyChange(PropertyChangeEvent e) {
        maxLabel.setText("Maximum = " + maximum.getMax());
    }
}

// A one line main program
public static void main(String[] args) {
    new DoubleListMain();
}
The ClearHandler clears the entire list.

The AverageAdapter recalculates the average when data is added or removed from the list.

The MaximumAdapter recalculates the maximum value when data is added or removed from the list.

The property change handlers update the JLabels in the GUI when the average or the maximum change. The handlers that make the changes to average and maximum could also set the values in the labels. We use this method to illustrate cascading events, and to demonstrate how to use PropertyChangeEvent.

The main method starts the application running.

### 2.5.5 Event Propagation

This example contains a significant amount of code. While all the small pieces of it hopefully make sense, it is worth looking at the big picture, too, to see the cascade of changes that takes place when a number is added to (or deleted from) the list.

To add a number to the list:

1. The number is entered in the inputField.
2. Enter is pressed which causes the inputField to fire an ActionEvent.
3. The AddHandler runs, updating the model with the new value.
4. The model fires a ListDataEvent which is handled by both the AverageAdapter and the MaximumAdapter. After updating their internal values, each of these in turn fires a PropertyChangeEvent.
5. The AvgPropHandler receives the PropertyChangeEvent from the AverageAdapter and updates the GUI’s average label.
6. The MaxPropHandler receives the PropertyChangeEvent from the MaximumAdapter and updates the GUI’s maximum label.

A total of four events and five handlers are used to add a single value to the list. An equal number is needed to deleted an element!

### 2.6 Procedural Event Programming

Throughout this chapter we have discussed event based programming using the language of objects. Our event sources, handlers and application entities were all objects. Event based programming can also be done using a procedural model.

In procedural programming objects don’t exist. We have data and procedures a.k.a. functions that manipulate the data. Functions replace methods. They exist at the top level, not within a class or object. An event handler is a function, typically called a callback function. The callback function is still registered dynamically. That is, during a run, not at compile time, we setup what function is called when an event occurs. This is still polymorphism, just based on functions rather than objects.

In languages like C and C++ functions may be passed as parameters to other functions. A function parameter is known as a function pointer. GUI libraries like the GL Utility Toolkit (GLUT) contain registration functions that take a function pointer as a parameter and register it to respond to events.
2.7 Summary

In this chapter you have seen several examples of event based programs in Java that have illustrated many of the main features of the paradigm. You are now ready to start developing your own GUI applications. There are dozens of more Swing classes and hundreds of more event classes, but they all work together following the same paradigm. You should be able to read the documentation pages and work with them easily.