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.

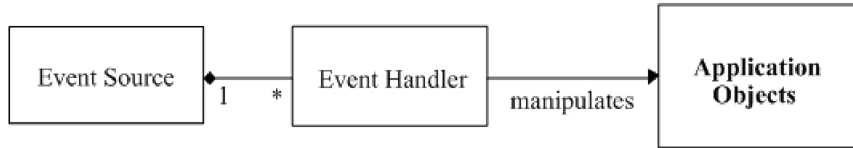

Figure 2.1: The object–oriented event processing model

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.

Figure 2.2: The user interface for `ClickMe.java`

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.

```java
1 /** ClickMe.java
2      This is a very simple event driven Java program.
3      It contains a single button, that when clicked pops open a message dialog.
4
5      Written by: Stuart Hansen
6      Date: September 2008
7 **/
8
9 import javax.swing.*;
10 import java.awt.event.*;
11
12 public class ClickMe extends JFrame {
13    JButton button;  // Our one and only button
14
15    public ClickMe () {
16       // The button is our event source
17       button = new JButton("Click me");
18
19       // We register a handler with the source
20       ActionHandler handler = new ActionHandler();
21       button.addActionListener(handler);
22
23       // We add the button to the viewable area of the window
24       getContentPane().add(button);
25
26       // We set a couple of window properties and then open the main window
27       setSize(100, 100);
28       setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
29       setVisible(true);
30    }
31    // A very simple main method
32    public static void main (String args[]) {
33       new ClickMe();
34    }
35 }
36
37 // The handler class.
38 class ActionHandler implements ActionListener {
39    public void actionPerformed (ActionEvent e) {
40       JOptionPane.showMessageDialog(null, "Ouch! That hurt.");
41    }
42 }
```

Lines	Commentary
9, 10	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.
12	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.
13	The `JButton` is the only GUI element in our window.
15–30	The `ClickMe` constructor in which we instantiate our objects, register the handler with source and set a few properties.
17	This instantiates the button.
20	`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.
21	`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.
24	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.
27	The `JFrame` is sized to 100x100 pixels.
28	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.
29	The `JFrame` is set to be visible.
32–34	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.
38–42	The `ActionHandler` class. It contains one method, `actionPerformed()` which is invoked when the button is clicked. The method opens a message window.

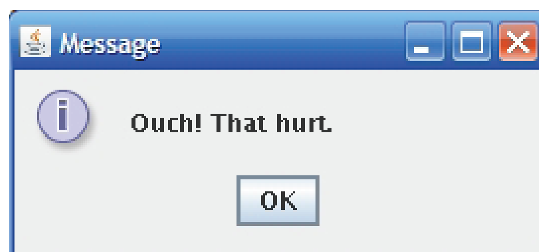

Figure 2.3: The message box opened by clicking the button.

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:

```
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 (`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.

```
1 /** ClickMeAgain.java
2      This program illustrates the use of anonymous classes for handlers
3      Written by: Stuart Hansen
4      Date: September 2008
5 **/
6
7 import javax.swing.*;
8 import java.awt.event.*;
9
10 public class ClickMeAgain extends JFrame {
11    JButton button;  // Our one and only button
12
13    public ClickMeAgain () {
14       // The button is our event source
15       button = new JButton("Click me");
16
17       // Register the handler
18       button.addActionListener(new ActionListener() {
19          public void actionPerformed (ActionEvent e) {
20             JOptionPane.showMessageDialog(null, "I said,  \"Don't do that.\"");
21          } }
22       );
23
24       // We add the button to the viewable area of the window
25       getContentPane().add(button);
26
27       // We set a couple of window properties and then open the main window
28       setSize(100, 100);
29       setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
30       setVisible(true);
31    }
32
33    // Event driven program regularly have simple main programs
34    public static void main (String args[]) {
35       new ClickMeAgain();
36    }
37 }
```

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.

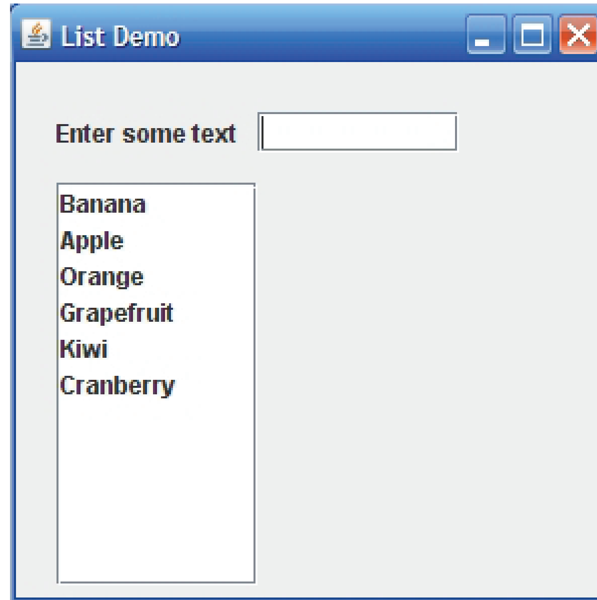

Figure 2.4: The GUI for the List of Strings program.

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 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.

```
1 import javax.swing.*;
2 import java.awt.*;
3 import java.awt.event.*;
4
5 /* ListDemo illustrates the use of JLabels, JTextFields,
6  * JScrollPanes and JLists.
7  *
8  * Written by: Stuart Hansen
9  * September 2008
10  */
11 public class ListDemo extends JFrame {
12    // These elements form the GUI of the application
13    private JLabel enterLabel;
14    private JTextField enterField;
15    private JScrollPane listPane;
16    private JList list;
17
18    // The constructor
19    public ListDemo () {
20       super ("List Demo");
21       enterLabel = new JLabel();
22       enterField = new JTextField();
23
24       // The DefaultListModel on the next line gives us the
25       // ability to add to the list
26       list = new JList(new DefaultListModel());
27
28       // The list is added to a JScrollPane, so that we may
29       // view lists larger than the display
30       listPane = new JScrollPane(list);
31
32       // setup the contentPane to use absolute coordinates
33       Container contentPane = getContentPane();
34       contentPane.setLayout(null);
35
36       // initialize and add the components to the GUI
37       enterLabel.setText("Enter some text");
38       enterLabel.setSize(100, 30);
39       enterLabel.setLocation(20, 20);
40       contentPane.add(enterLabel);
41
42       enterField.setSize(100, 20);
43       enterField.setLocation(120 ,25);
44       contentPane.add(enterField);
45       enterField.addActionListener(new InputHandler());
46
47       listPane.setSize(100, 200);
48       listPane.setLocation(20, 60);
49       contentPane.add(listPane);
```

```
50
51         // Set the windows size and close operation
52         setSize(300, 300);
53         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
54
55         // display the application window
56         setVisible(true);
57     }
58
59     // The ActionListener for the JTextField
60     // The event if fired when enter is pressed
61     // The text is added to the list
62     private class InputHandler implements ActionListener  {
63         public void actionPerformed(ActionEvent e) {
64             String text = enterField.getText();
65             DefaultListModel model = ((DefaultListModel)(list.getModel()));
66             if (!text.equals("")) {
67                 model.addElement(text) ;
68                 enterField.setText("");
69             }
70         }
71     }
72
73     // a very simple main program
74     public static void main (String args[]) {
75         new ListDemo();
76     }
77 }
```

Lines	Commentary
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. `JList`s 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 `JList`s, 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.

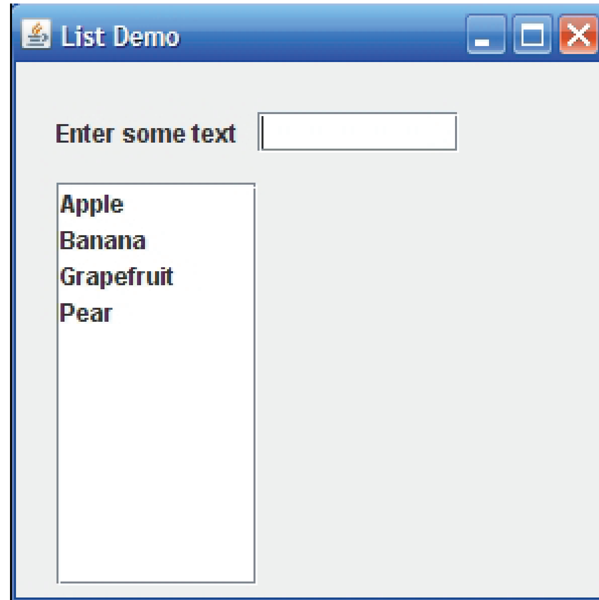

Figure 2.5: The sorted list program during a run.

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.

```java
1 import javax.swing.*;
2
3 /* SortedListModel.java replaces the default list model with one
4  * that keeps the list is sorted order.
5  *
6  * Written by: Stuart Hansen
7  * September 2008
8  */
9 public class SortedListModel extends DefaultListModel {
10    // We override addElement in the DefaultListModel class
11    // so that the JList remains sorted.
12    // Note that in a more complete application, add()
13    // set() and setElementAt() should also be overridden.
14    public void addElement(Object obj) {
15       String str = obj.toString();
16       int index = getSize();
17       while (index > 0 && ((String)elementAt(index-1)).compareTo(str) >=0) {
18          index--;
19       }
20       super.add(index, obj);
21    }
22 }
```

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 `ListModelListener`s whenever the model changed.

Because our new model inherited all the infrastructure to register and fire `ListDataEvent`s 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.

`JPanel`s 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.

```
1 import javax.swing.*;
2 import javax.swing.plaf.*;
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.util.*;
6
7 /**
8  * DrawPanel implements a JPanel that can be drawn on using
9  * the mouse
10 *
11 * @author Stuart Hansen
12 * @version September 2008
13 */
14
15 // We specialize the JPanel to contain a drawing.
16 public class DrawPanel extends JPanel {
17    // The model used for the JPanel is a list of curves.
18    ArrayList<Curve> listOfCurves;
19
20    // Two additional state variables to aid us in creating the drawing
21    Curve  currentCurve;     // the current curve being drawn
22    Color  currentColor;     // the current drawing color
23
24    public DrawPanel() {
25       super();
26
27       // Initialize the model
28       reset();
29
30       // replace the view
31       setUI (new DrawPanelUI());
32
33       // add specialized control
34       addMouseListener(new MouseHandler());
35       addMouseMotionListener(new MouseMotionHandler());
36    }
37
38    // A setter for the color
39    public void setColor(Color col) {
40       currentColor = col;
41    }
42
43    // a getter for the color
44    public Color getColor() {
45       return currentColor;
46    }
47
48    // reset the drawPanel model
49    public void reset() {
50       listOfCurves = new ArrayList<Curve>();
51       currentColor = Color.BLACK;
52       repaint();
53    }
```

Lines	Commentary
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 userful 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.

```
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.

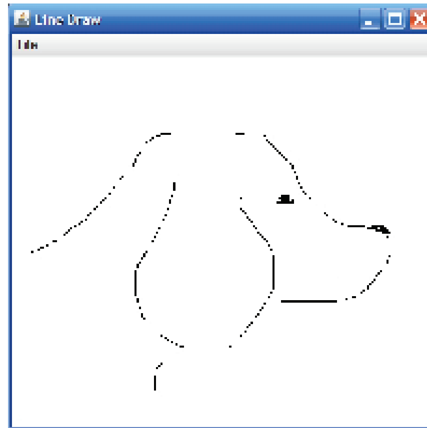

Figure 2.6: An example of a `DrawPanel`. The image was created by holding down the left mouse button and dragging the mouse.

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.

```
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
93             // We iterate across each curve drawing it
94             Curve curve = curveIterator.next();
95             Iterator<Point> pointIterator = curve.iterator();
96
97             // Set the color for this curve
98             g.setColor(curve.getColor());
99
100            // Iterate across the Points rendering the line segments
101            Point p1 = pointIterator.next();
102
103            while (pointIterator.hasNext()) {
104               Point p2 = pointIterator.next();
105               g.drawLine((int)p1.getX(), (int)p1.getY(),
106                          (int)p2.getX(), (int)p2.getY());
107               p1 = p2;
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.

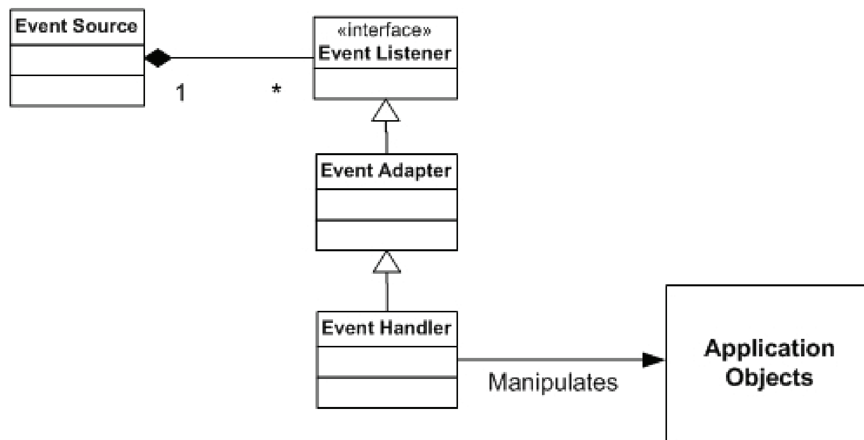

Figure 2.7: Java achieves decoupling between the event source and event handler by placing an interface and an adapter between them. The event listener specifies the methods that the event source expects in all handlers. The event adapter implements all of the methods of the interface with empty method bodies.

```
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.
- `JMenu`s are added to the menu bar.
- `JMenuItem`s are added to the menus. `JMenuItem`s 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'

```java
 1 import javax.swing.*;
 2 import javax.swing.plaf.*;
 3 import java.awt.*;
 4 import java.awt.event.*;
 5 import java.util.*;
 6
 7 /**
 8  * This class implements a drawing program in java
 9  *
10  * @author Stuart Hansen
11  * @version September 2008
12  */
13
14 public class DrawProgram extends JFrame {
15    // These elements form the GUI of the application
16    DrawPanel drawPanel = new DrawPanel();
17    JMenuBar menuBar = new JMenuBar();
18    JMenu menu = new JMenu ("File");
19    JMenuItem colorItem = new JMenuItem("Color");
20    JMenuItem clearItem = new JMenuItem("Clear");
21    JMenuItem exitItem = new JMenuItem ("Exit");
22
23    public DrawProgram () {
24       super ("Line Draw");
25
26       // add the menu to the application Frame
27       setJMenuBar (menuBar);
28       menu.add(colorItem);
29       menu.add(clearItem);
30       menu.add(exitItem);
31       menuBar.add(menu);
32
33       // setup the drawPanel
34       getContentPane().add(drawPanel);
35       drawPanel.setBackground(Color.white);
36
37       // Set the current Color
38       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.

```
39         // Change the drawing color
40         colorItem.addActionListener (
41            new ActionListener () {
42               public void actionPerformed (ActionEvent e) {
43                  Color oldColor = drawPanel.getColor();
44                  Color newColor = JColorChooser.showDialog(null,
45                                      "Choose a new color", oldColor);
46                  if (newColor != null)
47                     drawPanel.setColor(newColor);
48               }
49            }
50         );
51
52         // Clear the drawing by replacing the DrawingPanel
53         clearItem.addActionListener (
54            new ActionListener () {
55               public void actionPerformed (ActionEvent e) {
56                  drawPanel.reset();
57               }
58            }
59         );
60
61         // Exit the system elegantly
62         exitItem.addActionListener (
63            new ActionListener () {
64               public void actionPerformed (ActionEvent e) {
65                  System.exit(0);
66               }
67            }
68         );
69
70         // Set the application window's properties
71         setSize (400, 400);
72         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
73
74         // display the application window
75         setVisible(true);
76      }
77
78      // a very simple main program
79      public static void main (String args[]) {
80         new DrawProgram();
81      }
82   }
```

40–68 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`.

40–50 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.

53–59 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.

62–69 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.

71–75 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.

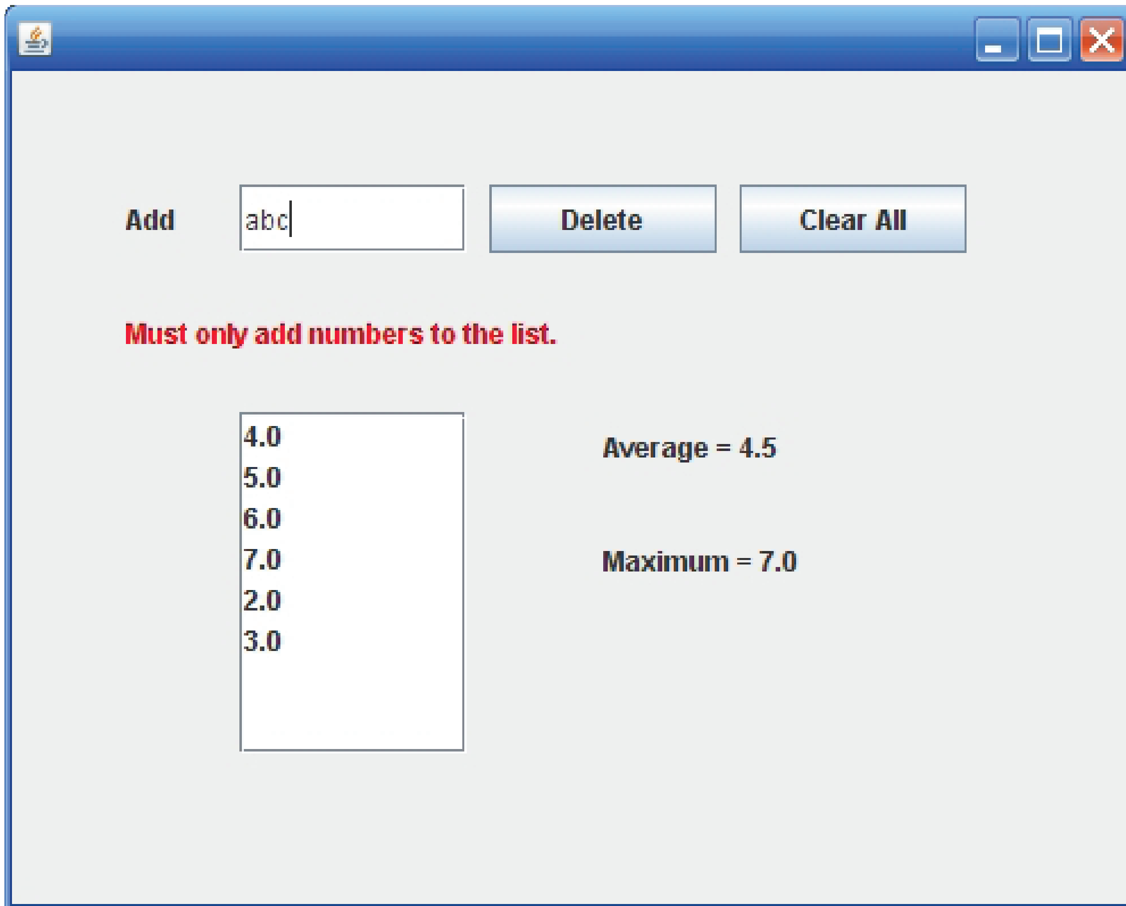

Figure 2.8: The user may add and delete numbers from the list. The application automatically updates the displayed average and maximum using event based techniques.

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 `Double`s may be added to the list.

```
1 import javax.swing.DefaultListModel;
2
3 /**
4  * DoubleListModel is the model for the DoubleList class.
5  * It overrides a couple of methods so that only Doubles are
6  * placed into it.
7  *
8  * @author Stuart Hansen
9  * @version September 2008
10  */
11
12 public class DoubleListModel extends DefaultListModel {
13     // Overrides add element so that only Doubles are added
14     public void addElement(Object obj) {
15         Double d = Double.parseDouble(obj.toString());
16         addElement(d);
17     }
18
19     // A specialized addElement for Doubles
20     public void addElement(Double d) {
21         super.addElement(d);
22     }
23
24     // Overrides toArray() so that the returned Array contains Doubles
25     public Double[] toArray() {
26         Object[] tempArr = super.toArray();
27         Double[] dArr = new Double[tempArr.length];
28         for (int i=0; i<tempArr.length; i++)
29             dArr[i] = (Double) tempArr[i];
30         return dArr;
31     }
32 }
```

Lines	Commentary
12	The `DoubleListModel` class extends the `DefaultListModel` class overriding several methods in it.
14–17	`addElement(Object obj)` is overridden. The method converts its `Object` parameter first to a `String` and then to a `Double`. In Java all objects have a `toString()` method, so this conversion is guaranteed to succeed. It then proceeds to convert the `String` to a `Double`. This code works correctly if the user has entered a valid `Double`. While not explicit in the code, Java throws a `ClassCastException` occurs if the conversion fails. Finally, we add the newly created `Double` to the list, using the overloaded `addElement(Double d)` method.
20–22	We also implement a specialized `addElement(Double d)` method that adds `Double`s to the list. Note that we are still using the list data structure maintained by the super class, `DefaultListModel`. 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 `Double`s. `DefaultListModel` contains several other methods that add or modify values in the list. These include: `add()`, `insertElementAt()`, `set()` and `setElementAt()`. A more complete `DoubleListModel` class would also override these methods. Since they aren't necessary for our example, they are ignored here.
25–31	Finally, we override `DefaultListModel`'s `toArray()` method so that the array returned is an array of `Double`s. We will use `toArray()` to get the values for calculating the average and maximum. Overriding `toArray()` keeps us from having to cast each element to a `Double` after accessing it in the original array. As with the previous note, there are several other "getter" methods that could be overridden, including: `elementAt()`, `elements()`, `firstElement()`, `get()`, `getElemetnAt()`, `lastElement()`, and `remove()`. Again, these are ignored, since they are not used by our application.

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.

```
1 import java.beans.*;
2
3 /**
4  *  The Average class finds and keeps track of the average
5  *  of an array of Doubles.
6  *
7  *  @author Stuart Hansen
8  *  @version September 2008
9  */
10 public class Average {
11    // We maintain the average in order to have an old
12    // average for the property change event
13    private Double average;
14
15    // We use pcs to facilitate our property change events
16    private PropertyChangeSupport pcs;
17
18    // The default constructor
19    public Average() {
20       average = new Double (Double.NaN);
21       pcs = new PropertyChangeSupport (this);
22    }
23
24    // Return the current Average
25    public Double getAverage() {
26       return average;
27    }
28
29    // Return the current Average as a String
30    public String toString () {
31       return average.toString();
32    }
33
34    // Find the average of an enumeration of Doubles
35    public Double findAverage(Double[] dArr) {
36       Double oldAverage = average;
37
38       if (dArr.length > 0) {
39          double sum = 0.0;
40          int count = 0;
41          for (Double d : dArr) {
42             sum += d;
43             count++;
44          }
45          average = new Double (sum/count);
46       }
47       else
48          average = Double.NaN;
49
50       firePropertyChange (new PropertyChangeEvent (this, "average",
51                                                   oldAverage, average));
52       return average;
53    }
```

Lines	Commentary
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.

```
55     // Fire a property change
56     public void firePropertyChange (PropertyChangeEvent e) {
57         pcs.firePropertyChange(e);
58     }
59
60     // Add a property change listener
61     public void addPropertyChangeListener(PropertyChangeListener pcl) {
62         pcs.addPropertyChangeListener(pcl);
63     }
64
65     // Delete a property change listener
66     public void removePropertyChangeListener(PropertyChangeListener pcl) {
67         pcs.removePropertyChangeListener(pcl);
68     }
69 }
```

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.

```
1 import java.awt.*;
2 import javax.swing.*;
3 import java.awt.event.*;
4 import javax.swing.event.*;
5
6 /**
7  * DoubleListMain puts together an application that records numbers
8  * into a list and reports their average and max.
9  *
10 * @author Stuart Hansen
11 * @version September 2008
12 */
13
14 public class DoubleListMain extends JFrame {
15    private JScrollPane pane;           // scrollpane for the list
16    private JList list;                 // the list's display
17    private DoubleListModel model;      // the model holding the list
18
19    private JLabel add;                 // the label for the add field
20    private JTextField inputField;      // where the numbers are entered
21    private JLabel errorLabel;          // a label to display error messages
22
23    private JLabel avgLabel;            // where the average is displayed
24    private Average avg;                // calculates the average
25
26    private JLabel maxLabel;            // where the max is displayed
27    private Max maximum;                // finds the maximum
28
29    private JButton del;                // deletes the selected element
30    private JButton clear;              // clears the entire list
31
32    // The constructor "wires" together the application
33    public DoubleListMain () {
34       Container cPane = getContentPane();
35       cPane.setLayout(null);
36
37       // Set up the list and its model
38       model = new DoubleListModel();
39       list = new JList();
40       list.setModel(model);
41       pane = new JScrollPane(list);
42       pane.setSize(100, 150);
43       pane.setLocation (100, 150);
44       cPane.add(pane);
45
46       // Set up the label for the add field
47       add = new JLabel("Add");
48       add.setLocation(50, 50);
49       add.setSize(40,30);
50       cPane.add(add);
51
52       // Set up the input textField
53       inputField = new JTextField();
54       inputField.setLocation(100, 50);
55       inputField.setSize(100, 30);
56       inputField.addActionListener(new AddHandler());
57       cPane.add(inputField);
```

Lines	Commentary
17, 24, 27	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.
15–30	The GUI consists of the usual assortment of buttons, labels, textfields and lists.
38–44	The model is placed in the list. The list is placed in the scrollpane. The scrollpane is added to the contentPane.
47–50	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.
53–57	We add the textfield used for input to the application.

```
59        // Set up the label for error messages
60        errorLabel = new JLabel("");
61        errorLabel.setForeground(Color.red);
62        errorLabel.setSize(400, 30);
63        errorLabel.setLocation(50, 100);
64        cPane.add(errorLabel);
65
66        // Set up the delete button
67        del = new JButton("Delete");
68        del.setSize (100, 30);
69        del.setLocation(210, 50);
70        del.addActionListener(new DeleteHandler());
71        cPane.add(del);
72
73        // Set up the clear button
74        clear = new JButton("Clear All");
75        clear.setSize (100, 30);
76        clear.setLocation(320, 50);
77        clear.addActionListener(new ClearHandler());
78        cPane.add(clear);
79
80        // Set up the average value and label
81        avg = new Average();
82        avgLabel = new JLabel("Average = NaN");
83        avgLabel.setSize(200, 30);
84        avgLabel.setLocation(260, 150);
85        model.addListDataListener(new AverageAdapter());
86        avg.addPropertyChangeListener(new AvgPropHandler());
87        cPane.add(avgLabel);
88
89        // Set up the maximum value and label
90        maximum = new Max();
91        maxLabel = new JLabel("Maximum = NaN");
92        maxLabel.setSize(200, 30);
93        maxLabel.setLocation(260, 200);
94        model.addListDataListener(new MaximumAdapter());
95        maximum.addPropertyChangeListener(new MaxPropHandler());
96        cPane.add(maxLabel);
97
98        // Set a few windowing parameters and show the frame.
99        setSize(500, 400);
100       setDefaultCloseOperation(EXIT_ON_CLOSE);
101       setVisible(true);
102   }
```

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.

```
104    // This class is the handler for adding numbers
105    private class AddHandler implements ActionListener {
106       public void actionPerformed (ActionEvent e) {
107          try {
108             String str = inputField.getText();
109             model.addElement(str);
110             inputField.setText("");
111             errorLabel.setText("");
112          }
113          catch (Exception ex) {
114             errorLabel.setText("Must only add numbers to the list.");
115          }
116       }
117    }
118
119    // This class is the handler for removing numbers
120    private class DeleteHandler implements ActionListener {
121       public void actionPerformed (ActionEvent e) {
122          try {
123             int index = list.getSelectedIndex();
124             model.remove(index);
125             errorLabel.setText("");
126          }
127          catch (Exception ex) {
128             errorLabel.setText("Error! Use mouse to select element to remove");
129          }
130       }
131    }
132
133    // This class is the handler for clearing the list
134    private class ClearHandler implements ActionListener {
135       public void actionPerformed (ActionEvent e) {
136          model.clear();
137       }
138    }
```

105–117 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.

120–131 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 getSelectedItem() 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.

```
140     // This class updates the average
141     private class AverageAdapter implements ListDataListener {
142        public void contentsChanged(ListDataEvent e) {}
143        public void intervalAdded(ListDataEvent e) {
144           Double[] temp = model.toArray();
145           avg.findAverage(temp);
146        }
147
148        public void intervalRemoved(ListDataEvent e) {
149           Double[] temp = model.toArray();
150           avg.findAverage(temp);
151        }
152     }
153
154     // This class updates the maximum
155     private class MaximumAdapter implements ListDataListener {
156        public void contentsChanged(ListDataEvent e) {}
157        public void intervalAdded(ListDataEvent e) {
158           Double [] temp = model.toArray();
159           maximum.findMax(temp);
160        }
161
162        public void intervalRemoved(ListDataEvent e) {
163           Double [] temp = model.toArray();
164           maximum.findMax(temp);
165        }
166     }
167
168     // This handler updates the JLabel when the average changes
169     private class AvgPropHandler implements PropertyChangeListener {
170        public void propertyChange (PropertyChangeEvent e) {
171           avgLabel.setText("Average = " + avg.getAverage());
172        }
173     }
174
175     // This handler updates the JLabel when the maximum changes
176     private class MaxPropHandler implements PropertyChangeListener {
177        public void propertyChange (PropertyChangeEvent e) {
178           maxLabel.setText("Maximum = " + maximum.getMaximum());
179        }
180     }
181
182
183     // A one line main program
184     public static void main (String [] args) {
185        new DoubleListMain();
186     }
187 }
```

134–138 The `ClearHandler` clears the entire list.

141–152 The `AverageAdapter` recalculates the average when data is added or removed from the list.

155–166 The `MaximumAdapter` recalculates the maximum value when data is added or removed from the list.

169–180 The property change handlers update the `JLabel`s 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`s.

184–186 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.