9

Graphical User Interfaces

OBJECTIVES

After studying this chapter, you will

- Acquire a familiarity with the Swing component set.
- Understand the relationship between the AWT and Swing.
- Have a better understanding of Java's event model.
- Be able to design and build simple Graphical User Interfaces (GUI)s.
- Appreciate how object-oriented design principles were used to extend Java's GUI capabilities.

OUTLINE

9.1 Introduction
From the Java Library: AWT to Swing
9.2 The Swing Component Set
Object-Oriented Design: Model-View-Controller Architecture
9.3 The Java Event Model
9.4 Case Study: Designing a Basic GUI
9.5 Containers and Layout Managers
9.6 Checkboxes, Radio Buttons, and Borders
9.7 Menus and Scroll Panes
Are Computers Intelligent?
In the Laboratory: The ATM Machine
Chapter Summary
Answers to Self-Study Exercises
Exercises

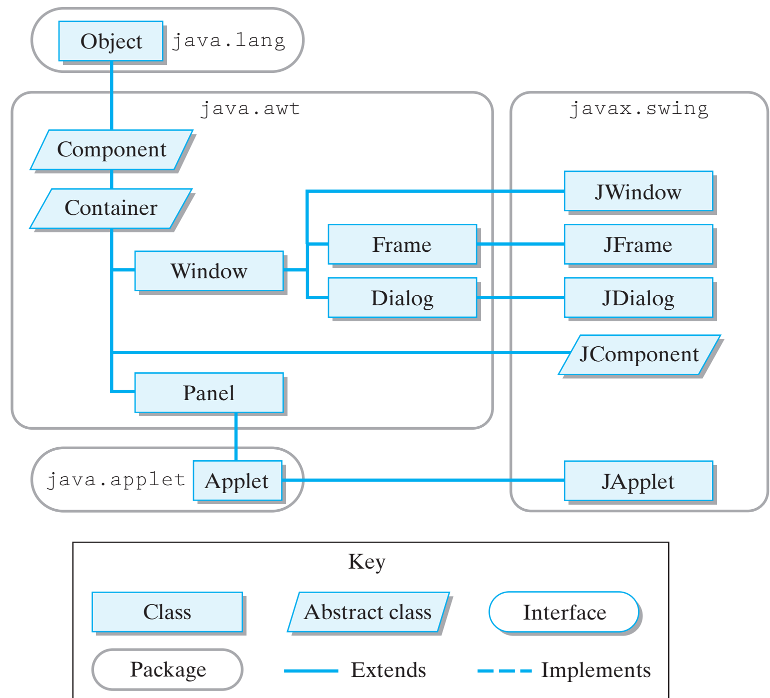


Figure 9–1 Swing classes, part 1: Relationship between the AWT and the top-level Swing windows.

windowing system, lightweight components are much more efficient than the regular AWT components.

All Swing components except for the four top-level window classes — the `JApplet`, `JDialog`, `JFrame`, and `JWindow` — are lightweight components. As you can see, Swing components that have corresponding AWT components have names that begin with "J." Figure 9–1 shows the relationship between the AWT `Container` and `Component` classes and the top-level Swing classes.

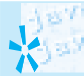
Because these four classes are derived from heavyweight components, they themselves are dependent on the native windowing system. However, note that the abstract `JComponent` class is derived directly from the `Container` class. Therefore it, and all of its Swing subclasses, are lightweight components (Figure 9–2).

The Future of the AWT

What's going to happen to the AWT in the future? It is somewhat of a misconception to assume that now that the Swing component set is available, the AWT package will be dropped. However, even if an application or applet uses Swing components (and no AWT component), that will still not break the dependence on the AWT. So despite the introduction of the Swing component set, it is clear that the AWT will remain an essential part of GUI development in Java.

Finally, all GUI applications and applets use layout managers (`java.awt.FlowLayout`), fonts (`java.awt.Font`), colors (`java.awt.Color`), and other (noncomponent) classes that are defined in the AWT. So there is no way to design a GUI without using AWT classes.

The programs presented in this chapter will use Swing components instead of AWT components. But they will also use layouts and other elements from the AWT. In terms of GUI components alone, Swing provides a replacement for every AWT component, as well as many new components that have no counterpart in the AWT. Although it is possible to mix and match AWT and Swing components in the same application, this is not advisable. Both sets of components use the same event model, so there are no problems on that score. But, if you are developing new software in Java, you should use the Swing components. This will allow you to take advantage of the new GUI features that come with Swing, and it will also place your programs squarely on the road to the future. That's the approach we will take in this and subsequent chapters.



PROGRAMMING TIP: Swing Documentation. Complete documentation of the Swing classes is available for downloading or browsing on Sun's Web site at:

```
http://java.sun.com/products/jdk/1.2/docs/guide/swing/index.html
```

SELF-STUDY EXERCISES

EXERCISE 9.1 What would have to be done to make a Swing-based GUI completely platform independent?

EXERCISE 9.2 Why are abstract classes, such as the `Container` and `Component` classes, not dependent on peers in the native windowing system?

9.2 The Swing Component Set

Java's Swing components are defined in a collection of packages named `javax.swing.*`, which is assumed by the code shown in this and subsequent chapters. (In JDK 1.1, these packages were named `com.sun.java.swing.*`, so if you are using JDK 1.1, you will have to change the package names in the programs that follow.) Some of the packages included under Swing include the following:

```
javax.swing.event.*
javax.swing.plaf.*
javax.swing.text.*
```

The `javax.swing.event` package defines the various Swing events and their listeners. (In the AWT, the AWT events and listeners were defined in `java.awt.event`.)

The `javax.swing.text` package contains the classes for `JTextField` and `JTextComponent`, the Swing classes that replace the AWT's `TextField` and `TextArea` classes. The Swing text components are more complex than their AWT counterparts. For example, one of their important features is the ability to undo changes made to the text they contain. This feature is crucial for building sophisticated word processing applications.

The `javax.plaf` package contains Swing's look-and-feel classes. The term *plaf* is an acronym for **pluggable look and feel**. It refers to the fact that changing an application's look and feel is a simple matter of "plugging in" a different plaf model. Changing how a program looks does not change what it does.

Swing's platform independent look and feel is achieved by placing all the code responsible for drawing a component in a separate class from the component itself. For example, in addition to `JButton`, the class which defines the button control, there will be a separate class responsible for drawing the button on the screen. The drawing class will control the button's color, shape, and other characteristics of its appearance.

There are several look-and-feel packages built into Swing. For example the `javax.swing.plaf.motif` package contains the classes that implement the Motif interface, a common Unix-based interface. These classes know how to draw each component, and how to react to mouse, keyboard and other events associated with these components. The `javax.swing.plaf.windows` package takes the same responsibility for a Windows 95 style interface.

OBJECT-ORIENTED DESIGN: Model-View-Controller Architecture

Java's Swing components have been implemented using an object-oriented design known as the **model-view-controller (MVC)** model. Any Swing component can be viewed in terms of three independent aspects: what state it's in (its model), how it looks (its view), and what it does (its controller).

For example, a button's role is to appear on the interface waiting to be clicked. When it is clicked, the button's appearance changes. It looks pushed in or it changes color briefly and then it changes back to its original (unclicked) appearance. In the MVC model, this aspect of the button is its **view**. If you were designing an interface for a button, you would need visual representations for both the clicked and the unclicked button (as well as other possible states).

When you click a button, its internal state changes from pressed to unpressed. You've also probably seen buttons that were disabled — that is, in a state where they just ignore your clicks. Whether a button is enabled or disabled, and whether it is pressed or not, are properties of its internal state. All such properties, taken together, constitute the button's **model**. Of course, a button's view — how it looks — depends on its model. When a button is pressed, it has one appearance, and when it is disabled, it has another.

Figure 9–5 An applet that handles action events on a JButton.

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

public class MyApplet extends JApplet implements ActionListener {
    private JButton clickme = new JButton("ClickMe");

    public void init() {
        getContentPane().add(clickme);   // Add clickme to the applet
        clickme.addActionListener(this); // Register it with a listener
    } // init()

    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == clickme) {
            showStatus("clickme was clicked");
            System.out.println( e.toString() );
        }
    } // actionPerformed()
} // MyApplet
```

In order to complete the event handling code, the applet must implement the `ActionListener` interface. As Figure 9–5 shows, implementing an interface is a matter of declaring the interface in the class heading, and implementing the methods contained in the interface, in this case the `actionPerformed()` method.

Now that we have implemented the code in Figure 9–5, whenever the user clicks on `clickme`, that action is encapsulated within an `ActionEvent` object and passed to the applet's `actionPerformed()` method. This method contains Java code that will handle the user's action in an appropriate way. For this example, it just prints a message in the applet's status bar and displays a string representation of the event.

The methods used to handle the `ActionEvent` are derived from the `java.util.EventObject` class, the root class for all events:

```
public class EventObject {
    public EventObject(Object source);
    public Object getSource();
    public String toString();
}
```

Our example (Figure 9–5) uses the `getSource()` method to get a reference to the object that generated the event. It also uses the `toString()` method to get a string representation of the event that was generated. Here's what it displays:

```
java.awt.event.ActionEvent[ACTION_PERFORMED,cmd=ClickMe]
  on javax.swing.JButton[,58,5,83x27,
  layout=javax.swing.OverlayLayout]
```

As you can see, the event generated was an `ACTION_PERFORMED` event, in response to the `ClickMe` command. The source of the event was the `JButton`.

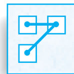
9.4 Case Study: Designing a Basic GUI:

What elements make up a basic user interface? If you think about all of the various interfaces you've encountered — and don't just limit yourself to computers — they all have the following elements:

- Some way to provide help/guidance to the user.
- Some way to allow input of information.
- Some way to allow output of information.
- Some way to control the interaction between the user and the device.

Think about the interface on a beverage machine. Printed text on the machine will tell you what choices you have, where to put your money, and what to do if something goes wrong. The coin slot is used to input money. There's often some kind of display to tell you how much money you've inserted. And there's usually a bunch of buttons and levers that let you control the interaction with the machine.

These same kinds of elements make up the basic computer interface. Designing a graphical user interface is primarily a process of choosing components that can effectively perform the tasks of input, output, control, and guidance.



EFFECTIVE DESIGN: User Interface. A user interface must effectively perform the tasks of input, output, control, and guidance.

In the programs we designed in the earlier chapters, we used two different kinds of interfaces. In the *command-line* interface, we used printed prompts to inform the user, typed commands for data entry and user control, and printed output to report results. Our applet interfaces used `Labels` to guide and prompt the user, `TextFields` and `TextAreas` as basic input and output devices, and either `Buttons` or `TextFieldss` for user control.

Up to this point, all of our GUIs have taken the form of Java applets. So let's begin by building a basic GUI in the form of a Java application. To keep the example as close as possible to the applet interfaces we've used, we'll build it out of the following Swing components: `JLabel`, `JTextField`, `JTextArea`, and `JButton`.

9.4.1 The Metric Converter Application

Suppose the coach of the cross country team asks you to write a Java application that can be used to convert miles to kilometers. The program should let the user input a distance in miles, and it should report the equivalent distance in kilometers.

Before we design the interface for this, let's first define a `MetricConverter` class that can be used to perform the conversions. For now at least, this class's only task will be to convert miles to kilometers, for

Second, note the statements used to set the layout and to add components directly to the `JFrame`. Instead of adding components directly to the `JFrame`, we must add them to its content pane:

```
getContentPane().add(input);
```

A **content pane** is a `JPanel` that serves as the working area of the `JFrame`. It contains all of the frame's components. Java will raise an exception if you attempt to add a component directly to a `JFrame`.

DEBUGGING TIP: A `JFrame` cannot directly contain GUI elements. Instead they must be added to its content pane, which can be retrieved using the `getContentPane()` method.

Unlike their AWT counterparts, `JFrame` and all the other top-level Swing windows have an internal structure made up of several distinct objects that can be manipulated by the program. Because of this structure, GUI elements can be organized into different layers within the window, making possible all sorts of sophisticated layouts. Also, one layer of the structure makes it possible to associate a menu with the frame. Thus, the use of a content pane represents a major advance beyond the functionality available with `java.awt.Frame`.

Finally, note how the `Converter` frame is instantiated, made visible, and eventually exited in the application's `main()` method:

```
public static void main(String args[]) {
    Converter f = new Converter();
    f.setSize(400, 300);
    f.setVisible(true);
    f.addWindowListener(new WindowAdapter() {      // Quit the application
        public void windowClosing(WindowEvent e) {
            System.exit(0);
        }
    });
} // main()
```

It is necessary to set both the size and visibility of the frame, since these are not set by default. Because we are using a `FlowLayout`, it's especially important to give the frame an appropriate size. Failure to do so may cause the components to be arranged is a confusing way and may even cause some components to not show up at all in the window. These are limitations we will fix when we learn how to use some of the other layout managers.

9.4.2 Inner Classes and Adapter Classes

Inner classes

Note also the code that's used to quit the `Converter` application. The program provides a listener that listens for window closing events. When such an event occurs, it exits the application by calling `System.exit()`.

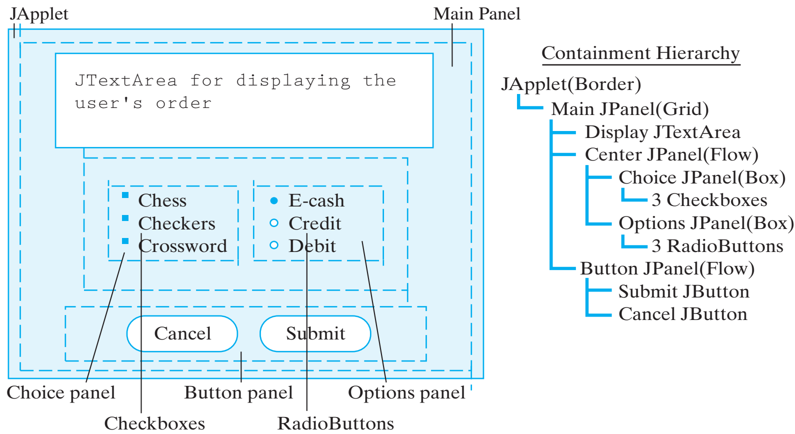


Figure 9–16 A design for an on-line order form interface.

Again, the advantage of this design is that it simplifies the instantiation and initialization of the buttons:

```
for(int k = 0; k < titles.length; k++) {
    titles[k] = new JCheckBox(titleLabels[k]);
    titles[k].addItemListener(this);
    choicePanel.add(titles[k]);
}
```

The only difference between this array of checkboxes and the keypad array of buttons that we used in the `Converter` program is that checkboxes generate `ItemEvents` instead `ActionEvents`. Therefore, each checkbox must be registered with an `ItemListener` (and of course the applet itself must implement the `ItemListener` interface). We'll show how `ItemEvents` are handled below.

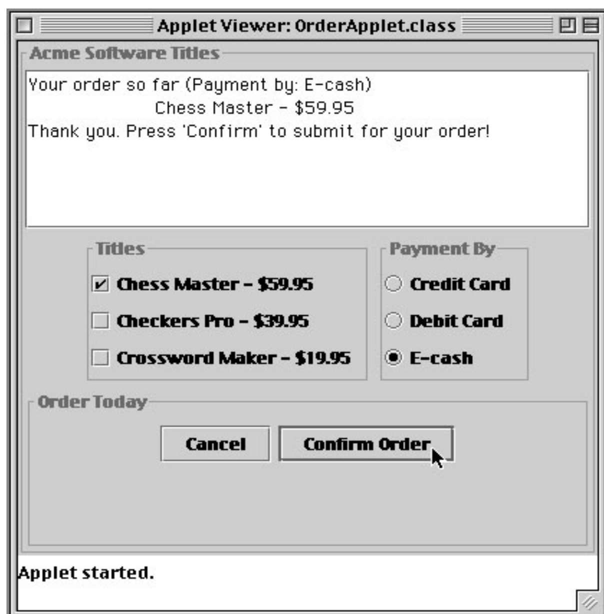


Figure 9–17 Borders around containers help make them stand out more.

area where computers still fall somewhat short is in speech recognition. However, just recently an American company demonstrated a telephone that could translate between English and German (as well as some other languages) in real time. The device's only limitation was that its discourse was limited to the travel domain. As computer processing speeds improve, this limitation is expected to be only temporary. Thus, we may be closer than we think to having our "conversational user interface."

Natural language understanding, speech recognition, learning, perception, chess playing, and problem solving are the kinds of problems addressed in AI, one of the major applied areas of computer science. Almost every major research group in AI has a Web site that describes their work. To find some of these, just do a search for "artificial intelligence" and then browse through the links that are found.

GUI Specifications

The Graphical User Interface (GUI) should contain a numeric *keyPad* — a 4 x 3 array of buttons — and a *commandPad*, a collection of buttons for the various functions that one finds on an ATM machine. It should contain a `JTextArea` which displays the result of clicking one of the buttons. Feel free to design your own layout of these components!

Designing the ATM Layout

The demo applet consists of a `JTextArea` that is used as the ATM's display screen. It displays all I/O during an ATM session. The only other interface components are the 12 key pad and 5 function `JButtons`. The demo uses several `JPanels` to achieve its overall layout (Figure 9–27). First there

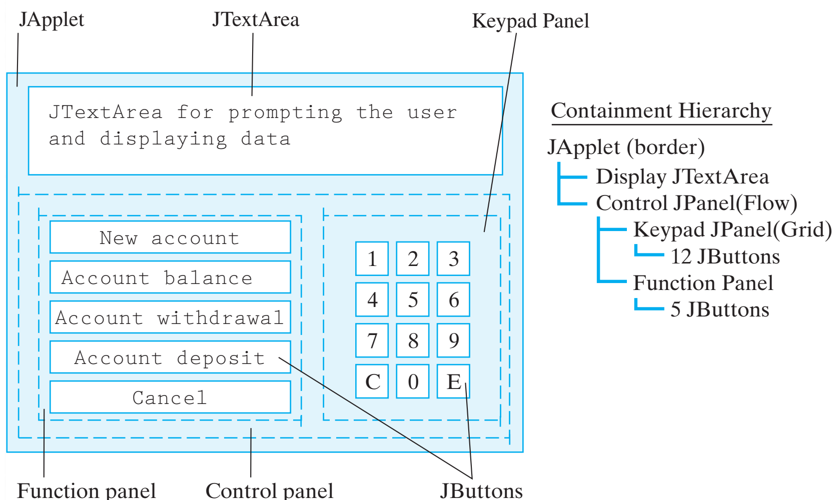


Figure 9–27 The containment hierarchy for the components used in the ATM machine GUI.

CHAPTER SUMMARY

Technical Terms

content pane	lightweight component	event model
layout manager	Model-View-Controller (MVC)	listener
model	widget hierarchy	peer model
view		
controller		

New Java Keywords

`super`	`this`

Java Library Classes

`ActionEvent`	`ActionListener`	`Applet`
`BasicButtonUI`	`BorderFactory`	`BorderLayout`
`Border`	`BoxLayout`	`Box`
`ButtonGroup`	`Button`	`CaretEvent`
`Choice`	`Component`	`Container`
`DefaultButtonModel`	`FlowLayout`	`FocusEvent`
`Frame`	`GridLayout`	`ItemEvent`
`ItemListener`	`JApplet`	`JBox`
`JButton`	`JComponent`	`JDialog`
`JFrame`	`JLabel`	`JList`
`JMenuBar`	`JMenuItem`	`JMenu`
`JPanel`	`JPopupMenu`	`JRadioButton`
`JScrollPane`	`JTable`	`JTextArea`
`JTextField`	`JTree`	`JWindow`
`KeyEvent`	`Label`	`ListDataEvent`
`MouseEvent`	`Object`	`String`
`TableColumnModelEvent`	`TextArea`	`TextComponent`
`TextField`	`TreeExpansionEvent`	`TreeSelectionEvent`
`UndoableEditEvent`	`Vector`	

Java Library Methods

`actionPerformed()`	`add()`	`addActionListener()`
`createTitledBorder()`	`elementAt()`	`getActionCommand()`
`getContentPane()`	`getItem()`	`getSelectedText()`
`getSource()`	`init()`	`insertElementAt()`
`isSelected()`	`main()`	`removeAll()`
`setBorder()`	`setLayout()`	