

Java

Methods

A & AB

Object-Oriented Programming
and
Data Structures

Maria Litvin

Phillips Academy, Andover, Massachusetts

Gary Litvin

Skylight Software, Inc.

Skylight Publishing
Andover, Massachusetts

Skylight Publishing
9 Bartlet Street, Suite 70
Andover, MA 01810

web: <http://www.skylit.com>
e-mail: sales@skylit.com
support@skylit.com

Library of Congress Control Number: 2005910949

ISBN-10: 0-9727055-7-0
ISBN-13: 978-0-9727055-7-8

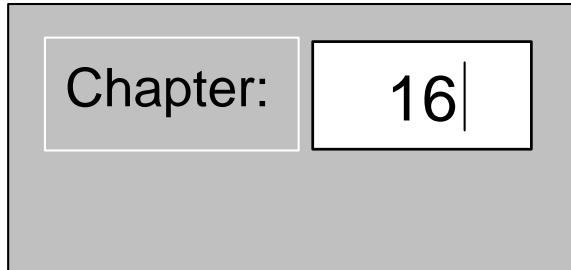
**Copyright © 2006 by Maria Litvin, Gary Litvin, and
Skylight Publishing**

This material is provided to you as a supplement to the book *Java Methods A&AB*. You may print out one copy for personal use and for face-to-face teaching for each copy of the *Java Methods A&AB* book that you own or receive from your school. You are not authorized to publish or distribute this document in any form without our permission. **You are not permitted to post this document on the Internet.** Feel free to create Internet links to this document's URL on our web site from your web pages, provided this document won't be displayed in a frame surrounded by advertisement or material unrelated to teaching AP* Computer Science or Java. You are not permitted to remove or modify this copyright notice.


* AP and Advanced Placement are registered trademarks of The College Board, which was not involved in the production of and does not endorse this book.

The names of commercially available software and products mentioned in this book are used for identification purposes only and may be trademarks or registered trademarks owned by corporations and other commercial entities. Skylight Publishing and the authors have no affiliation with and disclaim any sponsorship or endorsement by any of these products' manufacturers or trademarks' owners.

Sun, Sun Microsystems, Java, and Java logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.



GUI Components and Events

- 16.1 Prologue 2
- 16.2 Pluggable Look and Feel 3
- 16.3 Basic *Swing* Components and Their Events 6
- 16.4 Layouts 10
- 16.5 Menus 13
- 16.6 *Case Study and Lab: the Ramblecs Game* 14
- 16.7 Summary 17
- Exercises 

16.1 Prologue

In this chapter we discuss the basics of *graphical user interfaces* and *event-driven programming* in Java. Event-driven GUI is what made the OOP concept popular, and it remains the area where it's most relevant. While you can write console applications in Java, such programs won't take full advantage of Java or OOP; in most cases such programs could just as well be written in C or Pascal. The style of modern user interfaces — with many different types of control components such as menus, buttons, pull-down lists, checkboxes, radio buttons, and text edit fields — provides an arena where OOP and event-driven programming naturally excel.

Our task in this chapter and the following one is to organize more formally the bits and pieces of the *Swing* package and multimedia that you have managed to grasp from our examples so far. Overall, the Java API lists hundreds of classes and interfaces, with thousands of constructors and methods. A sizable portion of the API — more than 100 classes — deals with *Swing* GUI components, events handlers, and multimedia. The online documentation in HTML format gives you convenient access to these detailed specifications. Still, it is not very easy to find what you need unless you know exactly what to look for. In many cases it may be easier to look up an example of how a particular type of object is used than to read about it in the API spec file. In most cases you need only the most standard uses of classes and their methods, and there are thousands of examples of these in JDK demos, books, online tutorials, and other sources.

This is the approach our book has taken, too. While introducing Java's fundamental concepts and basic syntax and control structures, we have sneaked in a variety of commonly used GUI methods and “widgets.” We have added some bells and whistles here and there, just so you could see, if you cared to, how such things might be done. This chapter summarizes what you have already seen and fills in some gaps. Appendix D presents a synopsis and an index of the more commonly used GUI components that appear in the code examples in this book.

Knowing all the details of the latest GUI package still does not guarantee that the GUI in your application will “work.” In addition to working the way you, the programmer, intend, it must also be intuitive and convenient for the user. Designing a good user interface is a matter of experience, good sense, trial and error, paying attention to the software users, developing prototypes, and in some cases, relying on more formal “human factors” research. To become a good user interface designer, one should gain experience with a wide range of applications, observe what works

and what doesn't, and absorb the latest designs from cyberspace. Note that, strictly speaking, this skill may not be directly related to programming skills.

In this chapter we will discuss the “pluggable look and feel” and a few basic *Swing* components, some of their methods, and the events they generate. We will also get familiar with four layout managers that help to arrange GUI components on the application's window.

16.2 Pluggable Look and Feel

The phrase *look and feel* refers to the GUI aspect of a program: the appearance of windows, menus, dialog boxes, toolbars, and other GUI components; feedback for selected actions; navigation between screens; keyboard commands; sounds; and so on. Each operating system has its own look and feel, used by the system GUI and by built-in applications (such as *Windows Explorer*) as well as by other applications that choose to emulate the system's look and feel (such as *Internet Explorer* or *Microsoft Office*). An operating system may allow users to customize some aspects of the look and feel, for example, by selecting a color scheme or the appearance of file folders.

Software development tools available for a particular operating system help programmers develop applications that emulate the system look and feel. A programmer can, of course, opt for an entirely different look and feel for his program, but, in general, it is not a good idea to surprise users. An unexpected interface will make it more difficult to learn to use the program. So most programs use standard GUI components: buttons, pull-down boxes, sliders, checkboxes, radio buttons, and so on.

The look and feel differs slightly from system to system. To a large extent, this is a matter of marketing considerations and proprietary software issues among software companies. Look and feel also evolves with time, but it follows the same general trends in different operating systems.

The look and feel issue poses a dilemma for Java developers. On one hand, Java applications are supposed to be platform-independent, so they should look and feel exactly the same on different computer systems. On the other hand, users of a particular system are accustomed to the look and feel of that system and may be reluctant to work with programs that don't fit the mold. Java's response to this dilemma is *pluggable look and feel* (PLAF). The programmer or the user can choose among several look and feel configurations, including the standard Java look and feel and a system-specific look and feel.

Support for PLAF is one of the features of the *Swing* package. *Swing*'s `UIManager` class has a static method that returns an array of installed PLAFs. For example:

```
UIManager.LookAndFeelInfo[] plafs =
    UIManager.getInstalledLookAndFeels();
for (UIManager.LookAndFeelInfo plaf : plafs)
    System.out.println(plaf.getClassName());
```

The output under *Windows* might look like this:

```
javax.swing.plaf.metal.MetalLookAndFeel
com.sun.java.swing.plaf.motif.MotifLookAndFeel
com.sun.java.swing.plaf.windows.WindowsLookAndFeel
com.sun.java.swing.plaf.windows.WindowsClassicLookAndFeel
```

“Metal” is the name of the system-independent (cross-platform) Java look and feel; “Motif” is the look and feel typical for *Unix* and *Linux* systems; “Windows” is the system-specific look and feel under *Windows*.

`UIManager` provides static methods that return the names of the platform-independent and system-specific PLAFs, respectively. For example:

```
System.out.println(UIManager.getCrossPlatformLookAndFeelClassName());
System.out.println(UIManager.getSystemLookAndFeelClassName());
```

The output under *Windows* will be:

```
javax.swing.plaf.metal.MetalLookAndFeel
com.sun.java.swing.plaf.windows.WindowsLookAndFeel
```



There are three ways to set the look and feel for your Java program.

1. Setting PLAF using the `swing.properties` file

This is the most general way to affect how all Java programs look on your computer. The `swing.properties` file resides in the `<jreHome>\lib` folder, where `jreHome` means the root directory where the JRE (Java Rn-time Environment) is installed (C:\Program Files\Java\jre1.5.0_06\lib, for example). If the `swing.properties` file does not exist, you can create one. To set the default PLAF, add the `defaultlaf` attribute to the `swing.properties` file. For example:

```
swing.defaultlaf=com.sun.java.swing.plaf.windows.WindowsLookAndFeel
```

This will make all Java programs use the *Windows* LAF by default.

2. Setting PLAF using a command-line switch

You can override the `swing.properties` file setting by using a command line switch when you run a Java application. For example:

```
C:\Mywork>java -Dswing.defaultlaf=javax.swing.plaf.metal.MetalLookAndFeel
MyProg
```

If you are using an IDE, there is usually a way to add command-line options for programs.

3. Setting PLAF in the program

The most definitive way to set a desired LAF is to do it within the code of your program. This setting overrides both the `swing.properties` file and the command line option. To set LAF in the program, call the `UIManager`'s `setLookAndFeel` method. For example:

```
String plafName = UIManager.getSystemLookAndFeelClassName();
try
{
    UIManager.setLookAndFeel(plafName);
}
catch (Exception ex)
{
    System.out.println("*** " + plafName + " PLAF not installed ***");
}
```

If LAF is not specified in the `swing.properties` file, on the command-line, or in the program, then Java sets the cross-platform “Metal” look and feel by default.

The “Metal” LAF looks quite close to the *Windows XP* “silver” style, but the `JFileChooser` component looks too bland and is hard to get used to after fancier *Windows* and *Mac* screens.

You can find a few more details on setting look and feel in Sun’s Java tutorial [\[1\]](#).

16.3 Basic *Swing* Components and Their Events

Some of the more commonly used *Swing* components are:

- `JLabel` — displays an icon or a line of text
- `JButton` — triggers an “action event” when pressed
- `JToggleButton` and `JCheckBox` — toggle an option
- `JComboBox` and `JRadioButton` — choose an option out of several possibilities
- `JSlider` — adjusts a setting
- `JTextField`, `JPasswordField`, and `JTextArea` — allow the user to enter and display or edit a line of text, a password, or a multi-line fragment of text, respectively
- `JMenuBar`, `JMenu`, `JMenuItem` — support pull-down menus.

Each of these GUI objects is created using one of its constructors. For example, one of `JLabel`'s constructors takes one `String` parameter — the line of text to be displayed. Another constructor takes one `Icon` parameter — an image to be displayed. After an object is created, you can add or change its features by calling its methods. For example, `JLabel`'s `setText` method can be used to change the line of text it displays. A GUI object must be added to your application's or applet's “content pane” or to one of the other components. We'll discuss placement of GUI components on windows and panels in Section 16.4.



Each GUI component (with the exception of `JLabel` and `JPanel`) generates certain types of events. Your program can capture these events using an appropriate type of event “listener,” an object of a class that *implements* a particular “Listener” *interface*.

Recall that a class can implement several interfaces, so the same object can serve as several types of listeners. For example:

```
public class ControlPanel extends JPanel
    implements ActionListener, ChangeListener, KeyListener
{
    ...
}
```


To capture events from a component, you need to add the appropriate type of listener object to that component. For example, a `JButton` object generates “action” events. These events can be captured by an `ActionListener` object (that is, an object of a class that implements the `ActionListener` interface). The `ActionListener` interface requires one method:

```
public void actionPerformed(ActionEvent e)
```

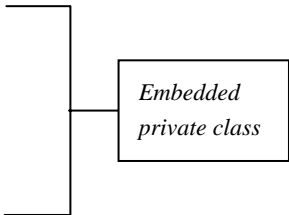
```
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JButton;

public class ControlPanel extends JPanel
{
    private EasySound bells = new EasySound("bells.wav");

    public ControlPanel()
    {
        JButton button = new JButton(" Play ");
        button.addActionListener(new PlayButtonListener());
        add(button);
    }

    private class PlayButtonListener
        implements ActionListener
    {
        public void actionPerformed(ActionEvent e)
        {
            bells.play();
        }
    }

    public static void main(String[] args)
    {
        JFrame window = new JFrame("ActionListener demo");
        window.setBounds(100, 100, 300, 100);
        window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        ControlPanel panel = new ControlPanel();
        window.getContentPane().add(panel);
        window.setVisible(true);
    }
}
```



The diagram consists of a large right-facing curly bracket on the left side of the `PlayButtonListener` class definition. A horizontal line extends from the center of this bracket to a rectangular box on the right. Inside the box, the text *Embedded private class* is written in italics.

Figure 16-1. An `ActionListener` implemented as an inner class

To add an “action listener” to a button, you have to call that button’s `addActionListener` method. Figure 16-1 gives an example. Here, the `PlayButtonListener` class is embedded into `ControlPanel` as a *private inner class*. We have tried to avoid inner classes in this book, and we could have made `PlayButtonListener` a separate public class instead. However, with many GUI components and a separate listener class for each of them, the number of source files would become quite large. Also, a listener may need access to fields in the class or object that creates it (for example, `bells` in the Figure 16-1 example).

When implementing an event listener, programmers often use a private inner class that has access to all the fields of the surrounding public class.

In simple cases, the object that creates a GUI component can also serve as its listener. Then you can just use this as a parameter to the `addActionListener` (or `add<Whatever>Listener`) method. For example:

```
public class ControlPanel extends JPanel
    implements ActionListener
{
    private EasySound bells = new EasySound("bells.wav");

    public ControlPanel()
    {
        JButton button = new JButton(" Play ");
        button.addActionListener(this);
        add(button);
    }

    public void actionPerformed(ActionEvent e)
    {
        bells.play();
    }
}
```

The advantage of the latter approach is simpler code. The disadvantage is that if your object has created several buttons, then its `actionPerformed` method has to sort them out and take different actions depending on which one was clicked. For example:

```
public void actionPerformed(ActionEvent e)
{
    JButton button = (JButton)e.getSource();
    if (button == myButton1)
        < ... do one thing >
    else if (button == myButton2)
        < ... do another thing >
}
```

A listener for a particular component is an object, not a class.

The *SnackBar* program, for example (Section 9.9), creates three similar objects of the *VendingMachine* class, and each machine becomes the action listener for its own buttons. Very convenient.



You do not have to capture every event from GUI components. With the exception of *JButton*, an event signals that the state of the component has changed. Sometimes you may prefer to retrieve the current state of the component later, when you need it, not right at the moment when it changes. (For instance, you may not care what options a user has chosen until he clicks “OK.”) All components provide methods for getting state information from them. For example, *JTextField* and *JTextArea* have the *getText* method, *JComboBox* has the *getSelectedItem* and *getSelectedItemIndex* methods, and *JCheckBox* has the *isSelected* method. In the *Benchmarks* program in Chapter 13, *JComboBox*’s and *JTextField*’s events are ignored while *JButton*’s events are captured and processed by an “action listener.” Then the listener’s *actionPerformed* method retrieves the information from the *JComboBox* and *JTextField* components. For *JButtons* you pretty much have to capture their events (unless it’s just a clicking exercise), because a button’s state does not change after it is clicked.

A common error is to create a GUI object but forget to attach a listener to it. Of course, then there is no way to capture its events.



In a Java program, events are represented by objects of special types: *ActionEvent*, *ItemEvent*, *KeyEvent*, *MouseEvent*, and so on. If you do capture an event, the event itself carries information that the listener’s method can use. You can find out which component caused the event by calling the event’s *getSource* method: it returns an *Object*, a reference to the object that caused the event. You can cast this returned *Object* into whatever type of component this listener is processing events for (see the example for *JButton* above).

For action events you can also retrieve an “action command” by calling the event’s `getActionCommand` method. It returns a string associated with the component. By default it returns the text written on the button or in a `JTextField` component, but you can set it yourself by calling your component’s `setActionCommand` method. This may be useful, for instance, if the same button should trigger different actions at different times (for example, “Go” / “Stop”).

The same component may generate different types of events captured by different types of listeners. Your program may choose to capture a certain type of events and ignore other types. Most components generate `ActionEvents` that are captured by an action listener.

Appendix D summarizes the basic *Swing* components, their event listeners, and their most commonly used constructors and methods. Most importantly, it gives references to some examples of their use.

16.4 Layouts

Learning about GUI components is only half the trick. The second half is learning how to place them on the screen.

In AWT and *Swing*, GUI components are added to containers.

A *container* is an object of the Java class `Container`. One container is the `JFrame`’s “content pane.” A reference to it can be obtained by calling the `getContentPane` method. Other containers include boxes (objects of the `Box` class) and panels (objects of the `JPanel` class). In fact, all *Swing* components have `Container` as an ancestor, so all of them are “containers.” But it is boxes and panels to which other components are usually added. You can have nested containers: boxes within boxes, boxes within panels, panels within boxes, and so on.

Repainting all the components in an application is a good opportunity to use recursion: for each component, first its picture is repainted, then (recursively) all components contained in it are repainted. Recursion is a perfect tool for dealing with such nested structures.

Java applications try to be platform-independent and to some extent scalable. To achieve this, Java gives up the possibility of precise placement of components based on specified pixel coordinates. Instead, components are placed with the help of “layout managers.” A layout manager implements a certain strategy for placing

components. In this section we will consider four kinds of layout managers: `FlowLayout`, `GridLayout`, `BorderLayout` and `BoxLayout`. (You can look up other layouts in the Java tutorial [1] and the API specs.) Each type of container has a default layout manager, but you can choose a different one by calling the container's `setLayout` method. A layout manager is an object, and as such it must be created before it can be used in a container. A typical idiom for setting a layout manager may look like this:

```
JPanel panel = new JPanel();
panel.setLayout(new FlowLayout());
panel.add(...);    // Add a component
< ... etc. >
```

By default, the content pane uses `BorderLayout`, a `Box` uses `BoxLayout`, and a `JPanel` uses `FlowLayout`.

Let us consider these three plus `GridLayout`, typically used with panels.

Flow Layout

`FlowLayout` is the most automatic and least precise of layout managers. It places components in the order they are added, starting from the top row of the container area and filling it as long as components fit, then starting a next row, and so on. There is a way to center the components or left- or right-justify them. For example:

```
Container c = getContentPane();
FlowLayout layout = new FlowLayout();
layout.setAlignment(FlowLayout.LEFT);
c.setLayout(layout);
c.add(...);    // Add a component
< ... etc. >
```

It is convenient to use `FlowLayout` when you want to put together a little program with a couple of GUI components quickly. A `FlowLayout` manager spaces the rows and the components within them in a reasonable way. We have used `FlowLayout` several times (see Appendix D).

Grid Layout

`GridLayout` is the opposite of `FlowLayout`: it is the most rigidly controlled. The grid occupies the whole area of the container, and all grid cells are the same size. `GridLayout`'s constructor takes two parameters, the number of rows and the number of columns in the grid. This constructor creates no gaps between the grid cells. Another constructor takes the number of rows and the number of columns in the grid plus two more parameters, the horizontal gap and the vertical gap between

the grid cells (in pixels). Components are added starting from the upper-left corner of the grid, filling the first row, then the next row, and so on. There is no way to skip a cell unless you put a dummy object into it (for example, an empty panel). We have used a `GridLayout` in the *Benchmarks* program in Chapter 13 and in the *Puzzle* program in Chapter 15.

Border Layout

`BorderLayout` splits the container area into five regions (Figure 16-2) and lets you add one component to each region. A border region can expand to reasonably fit the component in that region. If necessary, you can set the size of the component by calling its `setPreferredSize` method. When you add a component to a container with a “border” layout, you have to specify explicitly which region it goes into. For example:

```
Container c = getContentPane();  
c.add(crapsTable, BorderLayout.CENTER);  
c.add(controlPanel, BorderLayout.SOUTH);
```

We have used `BorderLayout` in many programs.

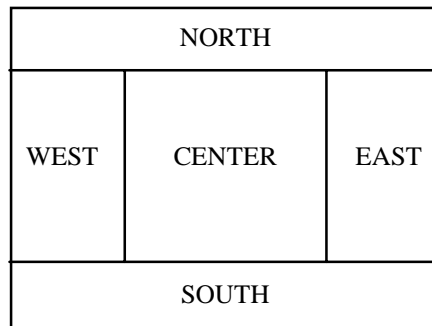


Figure 16-2. The border layout

Box Layout

`BoxLayout` can be used with panels, but it is usually used with boxes, in which it is the default layout. You can create a `Box` container using a constructor with one `int` parameter with the value `BoxLayout.X_AXIS` or `BoxLayout.Y_AXIS`. For some reason, there are also two static methods `Box.createHorizontalBox()` and `Box.createVerticalBox()` that return a reference to a new box, and it is common to use them instead of the constructors. In general, the idiom for working with boxes is different: it relies more on `Box`'s static methods.

“Horizontal” and “vertical” boxes are not defined by their dimensions but rather by how components are placed in them. In a horizontal box, components are added left to right. Stretchable components, such as panels, take the full height of the box. You can also add a horizontal “strut,” an invisible spacer that inserts a fixed amount of space between components. For example:

```
Box b = Box.createHorizontalBox();
b.add(...); // add a component
b.add(Box.createHorizontalStrut(10)); // unused space, 10 pixels
b.add(...); // add another component
< ... etc. >
```

In a vertical box, components are added starting from the top and fill the whole width if they can. For a vertical box you can add a vertical “strut” If you add a horizontal strut to a vertical box, it will set the minimum width of the box. We have used both horizontal and vertical boxes in the *Snack Bar* program in Chapter 9.

16.5 Menus

Any respectable program has a menu bar with pull-down menus. If nothing else, that's where the copyright message goes, under “Help / About...”

You can add a menu bar to a `JApplet` or a `JFrame` object, or any object of a class derived from one of them. A menu bar is a `JMenuBar` class object; it is added by calling the `setJMenuBar` method. You can add `JMenu` objects to your menu bar. To `JMenu` objects you can add `JMenuItems`, `JRadioButtonMenuItems`, `JCheckBoxMenuItems`, and more `JMenus` (submenus). You can split a menu into groups by calling `JMenu`'s `addSeparator` method. You can add “action listeners” to `JMenuItems`, checkboxes, and radio buttons. The code for all this is quite straightforward but verbose and repetitive. Visual development tools let programmers design menus and other GUI components interactively, then generate most of the Java code automatically. Still, you have to know what's going on, and

the best way to learn it is by working through an example. Sun's *How to Use Menus* tutorial [1] is a good starting point. It describes different types of components that can go into menus and provides several simple examples.

16.6 Case Study and Lab: The Ramblecs Game

Figure 16-3 shows a snapshot from the *Ramblecs* game program. In this game, a user navigates and rotates falling “letter cubes” in such a way that the letters form words in the bottom row. Run the program by clicking on the `Ramblecs.jar` file in `JM\Ch16\Ramblecs`. The help screen explains the rules.

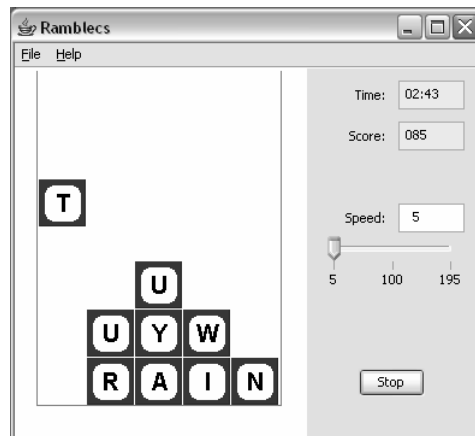


Figure 16-3. The *Ramblecs* game

Figure 16-4 shows a class diagram for *Ramblecs*. The `LetterCube` class represents a cube with 6 random letters assigned to its faces; one of the letters is designated as the front letter. The cube can be rotated, so that each of the letters can take the front position. (Rotation is logical in the array of six letters — it does not emulate cube rotation in three dimensions.) `FallingCube`, a subclass of `LetterCube`, adds `x`, `y` coordinates to the cube and a method to move it. The `RamblecsCharMatrix` class is similar to the `CharMatrix` class that you wrote for *Chomp* in Chapter 12; it represents a 2-D array of characters. Two methods are specific to *Ramblecs*: one makes and returns a string of characters in the bottom row; another shifts all the rows down by one. The `RamblecsDictionary` class holds a dictionary of 3-, 4-, and 5-letter words. You might recall that the source code for this class was generated from a words file by a program in the Chapter 14 lab. We have added a method to

the `RamblecsDictionary` class that checks whether a given word is in the dictionary with the help of `Arrays.binarySearch`.

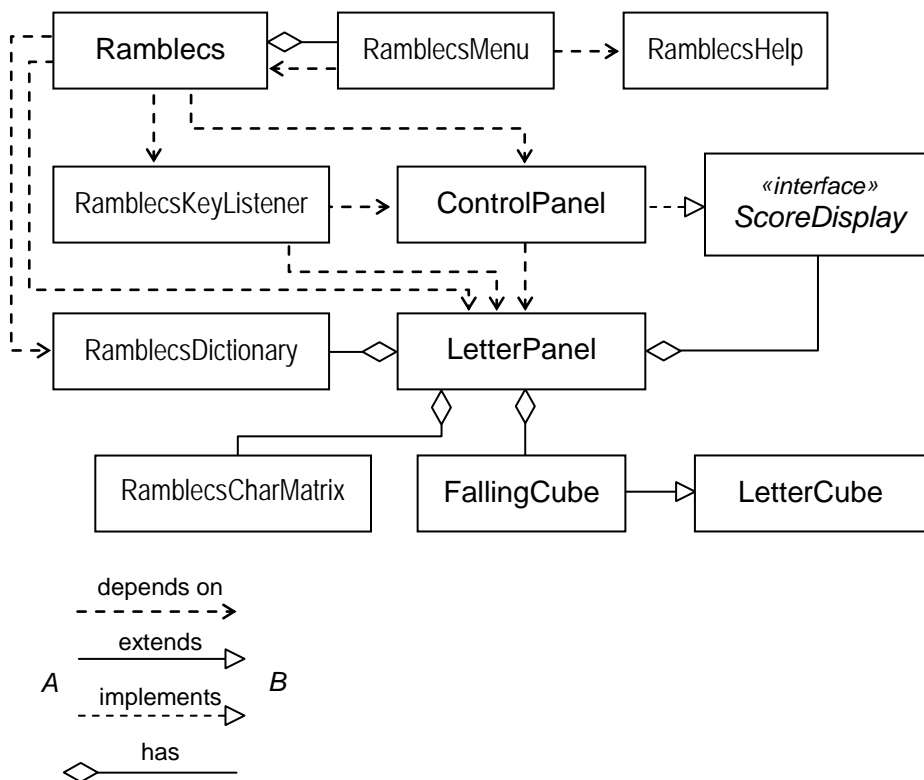


Figure 16-4. Classes in the *Ramblecs* program

The rest of the classes deal with graphics and GUI: this is a GUI-intensive application. `Ramblecs` is the main class; it creates a menu bar, a keyboard handler, a control panel, a “letter panel” on which the cubes fall, and a dictionary. `RamblecsKeyListener` handles keyboard input. `ControlPanel` holds the speed slider and the “Go” button and also serves as the display for the elapsed time and score. `LetterPanel` has methods for moving the falling cube and for drawing the letter grid and the cube. `RamblecsHelp` provides two static methods that display help screens.

GUI implementation can be challenging in two ways. First, it is hard to know ahead of time how things will look and feel in the program; you may need to do some prototyping or rely on the trial-and-error approach. Second, *Swing*’s strategies for

laying out components are complex and not fully documented; again, you may need to go through a few iterations to get it right.



Your task in this lab is to fill in the blanks in the `Ramblecs`, `RamblecsMenu`, and `ControlPanel` classes. We have intentionally left out the description of the GUI components involved and how they work: we want you to explore Appendix D and the Java API documentation [1] and tutorials [1, 2]. There are also hints in the source code comments.

1. The `Ramblecs` class:

Add code to `main` to set system-specific PLAF.

2. The `RamblecsMenu` class:

Experiment with the executable program and write a constructor and event handlers that work the same way. Note that the “Play Sound” menu item under File / Preferences does not generate events: a boolean method `soundEnabled` is provided instead. Sound should be enabled when the program is started. All other menu items generate events that are captured by an `ActionListener`: “New Game” calls `game`’s `newGame` method, “Exit” calls `System.exit`, and the Help menu items call `RamblecsHelp`’s static methods `showHelp` and `showAbout`, respectively.

3. The `ControlPanel` class:

This class represents the *Ramblecs* control panel. It contains three text fields with respective labels, a speed slider, and a “Go” button. Experiment with the executable program to see how they work. Note that the “Go” button toggles between “Go” and “Stop.” The time and score text fields are non-editable, but the user can enter an integer in the speed field, and then the speed setting and the speed slider are adjusted accordingly. Don’t let the program crash if the user input is invalid — instead, just set the field to the current speed slider setting.

Set up a project with the three classes that you have completed and `Ramblecs.jar`, and test the program thoroughly.

16.7 Summary

The `javax.swing` package supports platform-independent implementation of GUI components and pluggable look and feel. The default cross-platform look and feel is known as the “Metal” look and feel. However, a programmer can opt for a system-specific look and feel (one that looks like *Windows*, or *Mac OS*, or *UNIX Motif*) using *Swing*’s pluggable look and feel feature.

Appendix D presents a synopsis of several *Swing* GUI components and an index of their use in the case studies, labs, and exercises in this book.

Events generated by *Swing* components are captured by different event “listeners.” An event listener is an object of a class that implements `ActionListener`, `ItemListener`, `ChangeListener`, or another “listener” interface. Listener classes are often implemented as private inner classes. An event listener object can be added to a component by calling its `addActionListener`, `addItemListener`, or `addChangeListener` method. The object that creates a component can also be its event listener, in which case the parameter to the component’s `add<Whatever>Listener` method is `this`.

Most components generate action events that can be captured by the `actionPerformed` method of an action listener object. `JToggleButton`, `JCheckBox`, `JRadioButton`, and `JComboBox` components also generate “item events” that can be captured by the `itemStateChanged` method of an “item listener.” `JSlider` components generate “change events” that can be captured by the `stateChanged` method of a `ChangeListener` object.

Swing components are added to *containers* with the help of *layout managers*. One container is a *content pane*. To obtain a reference to it, call `JApplet`’s or `JFrame`’s `getContentPane` method. Its default layout manager has the `BorderLayout` type. Examples of other containers are boxes (objects of the `Box` class) and panels (objects of the `JPanel` class). For boxes, the default layout manager type is `BoxLayout`; for panels, `FlowLayout`. You can choose a different layout by calling the container’s `setLayout` method.

In `FlowLayout`, components are added in one row as long as they fit, then in the next row, and so on. Components are reasonably spaced and can be centered (default), right aligned, or left aligned. In `GridLayout`, components are placed on a rectangular grid that covers the container’s area and has all cells of the same size. You can specify horizontal and vertical gaps between the cells. Components are added starting at the upper-left corner of the grid and filling the first row, then the

next row, and so on. `BorderLayout` can accommodate up to five components, one for each of the five regions: north, south, east, west, and center. In `BoxLayout`, components fill a horizontal or vertical box. In a horizontal box, components are placed left to right; in a vertical box, components are placed top to bottom.

In all layouts, a component can be a panel or a box with its own components in it. There is no easy uniform way of placing components in Java, and one has to resort to a bag of tricks to get the layout right.

To add a menu bar (a `JMenuBar` object) to an applet or application, call its `setMenuBar` method. You can add `JMenu` objects to the program's menu bar and `JMenuItem`s, `JCheckBoxMenuItem`s, `JRadioButtonMenuItem`s, or `JMenus` (submenus) to any `JMenu`. You can split a menu into groups of items by calling `JMenu`'s `addSeparator` method. To make the menu work, add action listeners to `JMenuItem`s, checkboxes, and radio buttons.

Exercises

The exercises for this chapter are in the book (*Java Methods A and AB: Object-Oriented Programming and Data Structures, AP Edition*, ISBN 0-9727055-7-0, Skylight Publishing, 2006 [[1](#)]).