Chapter 1

Getting Started with Java

Android is Google's software stack for mobile devices that includes an operating system and middleware. With help from Java, the OS runs specially designed Java applications known as *Android apps*. Because these apps are based on Java, it makes sense for you to learn about Java before you dive into the world of Android development.

> **NOTE:** This book illustrates Java concepts via non-Android Java applications.

This chapter sets the stage for teaching you the essential Java concepts that you need to understand before you embark on your Android career. I first answer the "What is Java?" question. I next show you how to install the Java SE Development Kit, and introduce you to JDK tools for compiling and running Java applications.

After showing you how to install and use the open source NetBeans and Eclipse IDEs so that you can develop these applications faster, I present an application for playing a card game that I call *Four of a Kind*. This application gives you a significant taste of the Java language, and is the centerpiece of my discussion on developing applications.

What Is Java?

Java is a language and a platform originated by Sun Microsystems. This section briefly describes this language and reveals what it means for Java to be a platform. To meet various needs, Sun organized Java into three main editions: Java SE, Java EE, and Java ME. This section also briefly explores each of these editions, along with Android.

NOTE: Java has an interesting history that dates back to December 1990. At that time, James Gosling, Patrick Naughton, and Mike Sheridan (all employees of Sun Microsystems) were given the task of figuring out the next major trend in computing. They concluded that one trend would involve the convergence of computing devices and intelligent consumer appliances. Thus was born the Green project.

The fruits of Green were *Star7*, a handheld wireless device featuring a five-inch color LCD screen, a SPARC processor, a sophisticated graphics capability, and a version of Unix; and *Oak*, a language developed by James Gosling for writing applications to run on Star7, and which he named after an oak tree growing outside of his office window at Sun. To avoid a conflict with another language of the same name, Dr. Gosling changed this language's name to Java.

Sun Microsystems subsequently evolved the Java language and platform until Oracle acquired Sun in early 2010. Check out `http://java.sun.com/` for the latest Java news from Oracle.

Java Is a Language

Java is a language in which developers express *source code* (program text). Java's *syntax* (rules for combining symbols into language features) is partly patterned after the C and C++ languages to shorten the learning curve for C/C++ developers.

The following list identifies a few similarities between Java and C/C++:

- Java and C/C++ share the same single-line and multiline comment styles. Comments let you document source code.
- Many of Java's reserved words are identical to their C/C++ counterparts (`for`, `if`, `switch`, and `while` are examples) and C++ counterparts (`catch`, `class`, `public`, and `try` are examples).
- Java also supports character, double precision floating-point, floating-point, integer, long integer, and short integer primitive types, and via the same `char`, `double`, `float`, `int`, `long`, and `short` reserved words.
- Java also supports many of the same operators, including arithmetic (+, -, *, /, and %) and conditional (`?:`) operators.
- Java also uses brace characters ({ and }) to delimit blocks of statements.

The following list identifies a few differences between Java and C/C++:

- Java supports an additional comment style known as Javadoc. (I will briefly introduce Javadoc later in this chapter.)
- Java provides reserved words not found in C/C++ (`extends`, `strictfp`, `synchronized`, and `transient` are examples).

- Java supports the byte integer type, does not provided a signed version of the character type, and does not provide unsigned versions of integer, long integer, and short integer. Furthermore, all of Java's primitive types have guaranteed implementation sizes, which is an important part of achieving portability (discussed later). The same cannot be said of equivalent primitive types in C and C++.
- Java provides operators not found in C/C++. These operators include `instanceof` and `>>>` (unsigned right shift).
- Java provides labeled break and continue statements that you will not find in C/C++.

You will learn about single-line and multiline comments in Chapter 2. Also, you will learn about reserved words, primitive types, operators, blocks, and statements (including labeled break and continue) in that chapter.

Java was designed to be a safer language than C/C++. It achieves safety in part by omitting certain C/C++ features. For example, Java does not support *pointers* (variables containing addresses) and does not let you overload operators.

Java also achieves safety by modifying certain C/C++ features. For example, loops must be controlled by Boolean expressions instead of integer expressions where 0 is false and a nonzero value is true. (Chapter 2 discusses loops and expressions.)

Suppose you must code a C/C++ while loop that repeats no more than ten times. Being tired, you specify `while (x) x++;` (assume that x is an integer-based variable initialized to 0—I discuss variables in Chapter 2) where x++ adds 1 to x's value. This loop does not stop when x reaches 10; you have introduced a *bug* (a defect).

This problem is less likely to occur in Java because it complains when it sees `while (x)`. This complaint requires you to recheck your expression, and you will then most likely specify `while (x != 10)`. Not only is safety improved (you cannot specify just x), meaning is also clarified: `while (x != 10)` is more meaningful than `while (x)`.

The aforementioned and other fundamental language features support classes, objects, inheritance, polymorphism, and interfaces. Java also provides advanced features related to nested types, packages, static imports, exceptions, assertions, annotations, generics, enums, and more. Subsequent chapters explore all of these language features.

Java Is a Platform

Java is a platform for executing programs. In contrast to platforms that consist of physical processors (such as an Intel processor) and operating systems (such as Linux), the Java platform consists of a virtual machine and associated execution environment.

The *virtual machine* is a software-based processor that presents its own instruction set. The associated *execution environment* consists of libraries for running programs and interacting with the underlying operating system.

The execution environment includes a huge library of prebuilt classfiles that perform common tasks, such as math operations (trigonometry, for example) and network communications. This library is commonly referred to as the *standard class library*.

A special Java program known as the *Java compiler* translates source code into instructions (and associated data) that are executed by the virtual machine. These instructions are commonly referred to as *bytecode*.

The compiler stores a program's bytecode and data in files having the `.class` extension. These files are known as *classfiles* because they typically store the compiled equivalent of classes, a language feature discussed in Chapter 2.

A Java program executes via a tool (such as `java`) that loads and starts the virtual machine, and passes the program's main classfile to the machine. The virtual machine uses a *classloader* (a virtual machine or execution environment component) to load the classfile.

After the classfile has been loaded, the virtual machine's *bytecode verifier* component makes sure that the classfile's bytecode is valid and does not compromise security. The verifier terminates the virtual machine when it finds a problem with the bytecode.

Assuming that all is well with the classfile's bytecode, the virtual machine's *interpreter* interprets the bytecode one instruction at a time. *Interpretation* consists of identifying bytecode instructions and executing equivalent native instructions.

> **NOTE:** *Native instructions* (also known as native code) are the instructions understood by the underlying platform's physical processor.

When the interpreter learns that a sequence of bytecode instructions is executed repeatedly, it informs the virtual machine's *Just In Time (JIT) compiler* to compile these instructions into native code.

JIT compilation is performed only once for a given sequence of bytecode instructions. Because the native instructions execute instead of the associated bytecode instruction sequence, the program executes much faster.

During execution, the interpreter might encounter a request to execute another classfile's bytecode. When that happens, it asks the classloader to load the classfile and the bytecode verifier to verify the bytecode prior to executing that bytecode.

The platform side of Java promotes *portability* by providing an abstraction over the underlying platform. As a result, the same bytecode runs unchanged on Windows-based, Linux-based, Mac OS X–based, and other platforms.

NOTE: Java was introduced with the "write once, run anywhere" slogan. Although Java goes to great lengths to enforce portability, it does not always succeed. Despite being mostly platform independent, certain parts of Java (such as the scheduling of threads, discussed in Chapter 7) vary from underlying platform to underlying platform.

The platform side of Java also promotes *security* by providing a secure environment in which code executes. The goal is to prevent malicious code from corrupting the underlying platform (and possibly stealing sensitive information).

NOTE: Because many developers are not satisfied with the Java language, but believe that the Java platform is important, they have devised additional languages (such as Groovy) that run on the Java platform. Furthermore, Java version 7 includes an enhanced virtual machine that simplifies adapting even more *dynamic programming languages* (languages that require less-rigid coding; you do not have to define a variable's type before using the variable, for example) to this platform.

Java SE, Java EE, Java ME, and Android

Developers use different editions of the Java platform to create Java programs that run on desktop computers, web browsers, web servers, mobile information devices (such as cell phones), and embedded devices (such as television set-top boxes):

- *Java Platform, Standard Edition (Java SE)*: The Java platform for developing *applications*, which are stand-alone programs that run on desktops. Java SE is also used to develop *applets*, which are programs that run in the context of a web browser.
- *Java Platform, Enterprise Edition (Java EE)*: The Java platform for developing enterprise-oriented applications and *servlets*, which are server programs that conform to Java EE's Servlet API. Java EE is built on top of Java SE.
- *Java Platform, Micro Edition (Java ME)*: The Java platform for developing *MIDlets*, which are programs that run on mobile information devices, and *Xlets*, which are programs that run on embedded devices.

Developers also use a special Google-created edition of the Java platform (see `http://developer.android.com/index.html`) to create Android apps that run on Android-enabled devices. This edition is known as the *Android platform*.

Google's Android platform largely consists of Java core libraries (partly based on Java SE) and a virtual machine known as *Dalvik*. This collective software runs on top of a specially modified Linux kernel.

> **NOTE:** Check out Wikipedia's "Android (operating system)" entry (`http://en.wikipedia.org/wiki/Android_%28operating_system%29`) to learn more about the Android OS, and Wikipedia's "Dalvik (software)" entry (`http://en.wikipedia.org/wiki/Dalvik_%28software%29`) to learn more about the Dalvik virtual machine.

In this book, I cover the Java language (supported by Java SE and Android) and Java SE APIs (also supported by Android). Furthermore, I present the source code (typically as code fragments) to Java SE–based applications.

Installing and Exploring the JDK

The *Java Runtime Environment (JRE)* implements the Java SE platform and makes it possible to run Java programs. The public JRE can be downloaded from the Java SE Downloads page (`http://java.sun.com/javase/downloads/index.jsp`).

However, the public JRE does not make it possible to develop Java programs. For that task, you need to download and install the *Java SE Development Kit (JDK)*, which contains development tools (including the Java compiler) and a private JRE.

> **NOTE:** JDK 1.0 was the first JDK to be released (in May 1995). Until JDK 6 arrived, JDK stood for Java Development Kit (SE was not part of the title). Over the years, numerous JDKs have been released, with JDK 7 set for release in fall or winter 2010.
>
> Each JDK's version number identifies a version of Java. For example, JDK 1.0 identifies Java version 1.0, and JDK 5 identifies Java version 5.0. JDK 5 was the first JDK to also provide an internal version number: 1.5.0.

The Java SE Downloads page also provides access to the current JDK, which is JDK 6 Update 20 at time of writing. Click the Download JDK link to download the current JDK's installer program for your platform.

> **NOTE:** Some of this book's code requires JDK 7, which is only available as a preview release (`http://java.sun.com/javase/downloads/ea.jsp`) at time of writing.

The JDK installer installs the JDK in a home directory. (It can also install the public JRE in another directory.) On my Windows XP platform, the home directory is `C:\Program Files\Java\jdk1.6.0_16`—JDK 6 Update 16 was current when I began this book.

TIP: After installing the JDK, you should add the `bin` subdirectory to your platform's `PATH` environment variable. That way, you will be able to execute JDK tools from any directory in your filesystem.

Finally, you might want to create a `projects` subdirectory of the JDK's home directory to organize your Java projects, and create a separate subdirectory within `projects` for each of these projects.

The home directory contains various files (such as `README.html`, which provides information about the JDK, and `src.zip`, which provides the standard class library source code) and subdirectories, including the following three important subdirectories:

- `bin`: This subdirectory contains assorted JDK tools, including the Java compiler tool. You will discover some of these tools shortly.
- `jre`: This subdirectory contains the JDK's private copy of the JRE, which lets you run Java programs without having to download and install the public JRE.
- `lib`: This subdirectory contains library files that are used by JDK tools. For example, `tools.jar` contains the Java compiler's classfiles—the compiler was written in Java.

You will use only a few of the `bin` subdirectory's tools in this book, specifically `javac` (Java compiler), `java` (Java application launcher), `javadoc` (Java documentation generator), and `jar` (Java archive creator, updater, and extractor).

NOTE: `javac` is not the Java compiler. It is a tool that loads and starts the virtual machine, identifies the compiler's main classfile (located in `tools.jar`) to the virtual machine, and passes the name of the source file being compiled to the compiler's main classfile.

You execute JDK tools at the *command line*, passing *command-line arguments* to a tool. Learn about the command line and arguments via Wikipedia's "Command-line interface" entry (`http://en.wikipedia.org/wiki/Command-line_interface`).

Now that you have installed the JDK and know something about its tools, you are ready to explore a small `DumpArgs` application that outputs its command-line arguments to the standard output device.

NOTE: The standard output device is part of a mechanism known as *Standard I/O*. This mechanism, which consists of Standard Input, Standard Output, and Standard Error, and which originated with the Unix operating system, makes it possible to read text from different sources (keyboard or file) and write text to different destinations (screen or file).

Text is read from the standard input device, which defaults to the keyboard but can be redirected to a file. Text is output to the standard output device, which defaults to the screen but can be redirected to a file. Error message text is output to the standard error device, which defaults to the screen but can be redirected to a file that differs from the standard output file.

Listing 1–1 presents the `DumpArgs` application source code.

Listing 1–1. *Dumping command-line arguments via `main()`'s `args` array to the standard output device*

```
public class DumpArgs
{
   public static void main(String[] args)
   {
      System.out.println("Passed arguments:");
      for (int i = 0; i < args.length; i++)
         System.out.println(args[i]);
   }
}
```

Listing 1–1's `DumpArgs` application consists of a class named `DumpArgs` and a method within this class named `main()`, which is the application's entry point and provides the code to execute. (You will learn about classes and methods in Chapter 2.)

`main()` is called with an array of *strings* (character sequences) that identify the application's command-line arguments. These strings are stored in `String`-based array variable `args`. (I discuss method calling, arrays, and variables in Chapter 2.)

NOTE: Although the array variable is named `args`, there is nothing special about this name. You could name this variable anything you want.

`main()` first executes `System.out.println("Passed arguments:");`, which calls `System.out`'s `println()` method with the `"Passed arguments:"` string. This method call outputs `Passed arguments:` to the standard output device and then terminates the current line so that subsequent output is sent to a new line immediately below the current line. (I discuss `System.out` in Chapter 7.)

NOTE: `System.out` provides access to a family of `println()` methods and a family of `print()` methods for outputting different kinds of data (such as sequences of characters and integers). Unlike the `println()` methods, the `print()` methods do not terminate the current line; subsequent output continues on the current line.

Each `println()` method terminates a line by outputting a line separator string, which is defined by system property `line.separator`, and which is not necessarily a single newline character (identified in source code via character literal `'\n'`). (I discuss system properties in Chapter 7, `line.separator` in Chapter 10, and character literals in Chapter 2.) For example, on Windows platforms, the line separator string is a carriage return character (whose integer code is 13) followed by a line feed character (whose integer code is 10).

`main()` uses a for loop to repeatedly execute `System.out.println(args[i]);`. The loop executes `args.length` times, which happens to identify the number of strings that are stored in `args`. (I discuss for loops in Chapter 2.)

The `System.out.println(args[i]);` method call reads the string stored in the `i`th entry of the `args` array—the first entry is located at *index* (location) 0; the last entry is stored at index `args.length - 1`. This method call then outputs this string to standard output.

Assuming that you are familiar with your platform's command-line interface and are at the command line, make `DumpArgs` your current directory and copy Listing 1–1 to a file named `DumpArgs.java`. Then compile this source file via the following command line:

```
javac DumpArgs.java
```

Assuming that that you have included the `.java` extension, which is required by `javac`, and that `DumpArgs.java` compiles, you should discover a file named `DumpArgs.class` in the current directory. Run this application via the following command line:

```
java DumpArgs
```

If all goes well, you should see the following line of output on the screen:

```
Passed arguments:
```

For more interesting output, you will need to pass command-line arguments to `DumpArgs`. For example, execute the following command line, which specifies `Curly`, `Moe`, and `Larry` as three arguments to pass to `DumpArgs`:

```
java DumpArgs Curly Moe Larry
```

This time, you should see the following expanded output on the screen:

```
Passed arguments:
Curly
Moe
Larry
```

You can redirect the output destination to a file by specifying the greater than angle bracket (>) followed by a filename. For example, `java DumpArgs Curly Moe Larry >out.txt` stores the `DumpArgs` application's output in a file named `out.txt`.

> **NOTE:** Instead of specifying `System.out.println()`, you could specify `System.err.println()` to output characters to the standard error device. (`System.err` provides the same families of `println()` and `print()` methods as `System.out`.) However, you should only switch from `System.out` to `System.err` when you need to output an error message so that the error messages are displayed on the screen, even when standard output is redirected to a file.

Congratulations on successfully compiling your first application source file and running the application! Listing 1–2 presents the source code to a second application, which echoes text obtained from the standard input device to the standard output device.

Listing 1–2. *Echoing text read from standard input to standard output*

```
public class EchoText
{
   public static void main(String[] args) throws java.io.IOException
   {
      System.out.println("Please enter some text and press Enter!");
      int ch;
      while ((ch = System.in.read()) != 13)
         System.out.print((char) ch);
      System.out.println();
   }
}
```

After outputting a message that prompts the user to enter some text, `main()` introduces `int` variable `ch` to store each character's integer representation. (You will learn about `int` and integer in Chapter 2.)

`main()` now enters a while loop (discussed in Chapter 2) to read and echo characters. The loop first calls `System.in.read()` to read a character and assign its integer value to `ch`. The loop ends when this value equals 13 (the integer value of the Enter key).

> **NOTE:** When standard input is not redirected to a file, `System.in.read()` returns 13 to indicate that the Enter key has been pressed. On platforms such as Windows, a subsequent call to `System.in.read()` returns integer 10, indicating a line feed character. Whether or not standard input has been redirected, `System.in.read()` returns -1 when there are no more characters to read.

For any other value in `ch`, this value is converted to a character via `(char)`, which is an example of Java's cast operator (discussed in Chapter 2). The character is then output via `System.out.println()`. The final `System.out.println();` call terminates the current line without outputting any content.

NOTE: When standard input is redirected to a file and `System.in.read()` is unable to read text from the file (perhaps the file is stored on a removable storage device that has been removed prior to the read operation), `System.in.read()` fails by throwing an object that describes this problem. I acknowledge this possibility by appending `throws java.io.IOException` to the end of the `main()` method header. I discuss `throws` in Chapter 4 and `java.io.IOException` in Chapter 10.

Compile Listing 1–2 via `javac EchoText.java`, and run the application via `java EchoText`. You will be prompted to enter some text. After you input this text and press Enter, the text will be sent to standard output. For example, consider the following output:

```
Please enter some text and press Enter!
Hello Java
Hello Java
```

You can redirect the input source to a file by specifying the less than angle bracket (<) followed by a filename. For example, `java EchoText <EchoText.java` reads its text from `EchoText.java` and outputs this text to the screen.

Run this application and you will only see `EchoText.java`'s first line of text. Each one of this file's lines ends in a carriage return character (13) (followed by a line feed character, 10, on Windows platforms), and `EchoText` terminates after reading a carriage return.

In addition to downloading and installing the JDK, you might want to download the JDK's companion documentation archive file (`jdk-6u18-docs.zip` is the most recent file at time of writing).

After downloading the documentation archive file from the same Java SE Downloads page (`http://java.sun.com/javase/downloads/index.jsp`), unzip this file and move its `docs` directory to the JDK's home directory.

To access the documentation, point your web browser to the documentation's start page. For example, after moving `docs` to my JDK's home directory, I point my browser to `C:\Program Files\Java\jdk1.6.0_16\docs\index.html`. See Figure 1–1.

Scroll a bit down the start page and you discover the "API, Language, and Virtual Machine Documentation" section, which presents a Java 2 Platform API Specification link. Click this link and you can access the standard class library's documentation.

TIP: You can read the online documentation by pointing your web browser to a link such as `http://download.java.net/jdk6/archive/b104/docs/`, which provides the online documentation for JDK 6.

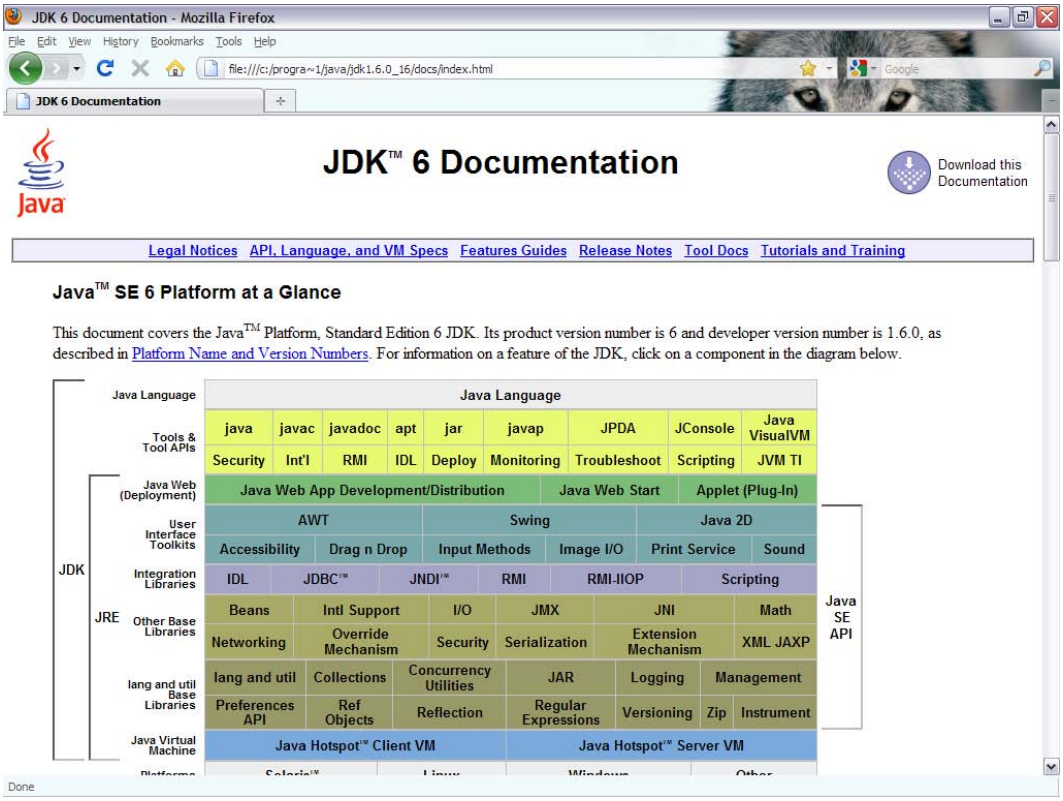

Figure 1–1. *The first part of the Java documentation's start page*

Installing and Exploring Two Popular IDEs

Working with the JDK's tools at the command line is probably okay for small projects. However, this practice is not recommended for large projects, which are hard to manage without the help of an integrated development environment (IDE).

An IDE consists of a project manager for managing a project's files, a text editor for entering and editing source code, a debugger for locating bugs, and other features. Two popular IDEs are NetBeans and Eclipse.

> **NOTE:** For convenience, I use JDK tools throughout this book, except for this section where I use NetBeans IDE and Eclipse IDE.

NetBeans IDE

NetBeans IDE is an open source, Java-based IDE for developing programs in Java and other languages (such as PHP, Ruby, C++, Groovy, and Scala). Version 6.8 is the current version of this IDE at time of writing.

You should download and install NetBeans IDE 6.8 (or a more recent version) to follow along with this section's NetBeans-oriented example. Begin by pointing your browser to `http://netbeans.org/downloads/` and accomplishing the following tasks:

1. Select an appropriate IDE language (such as English).
2. Select an appropriate platform (such as Linux).
3. Click the Download button underneath the leftmost (Java SE) column.

After a few moments, the current page is replaced by another page that gives you the opportunity to download an installer file. I downloaded the approximately 47MB `netbeans-6.8-ml-javase-windows.exe` installer file for my Windows XP platform.

> **NOTE:** According to the "NetBeans IDE 6.8 Installation Instructions" (`http://netbeans.org/community/releases/68/install.html`), you must install JDK 5.0 Update 19 or JDK 6 Update 14 or newer on your platform before installing NetBeans IDE 6.8. If you do not have a JDK installed, you cannot install the NetBeans IDE.

Start the installer file and follow the instructions. You will need to agree to the NetBeans license, and are given the options of providing anonymous usage data and registering your copy of NetBeans when installation finishes.

Assuming that you have installed the NetBeans IDE, start this Java application. You should discover a splash screen identifying this IDE, followed by a main window similar to that shown in Figure 1–2.

The NetBeans user interface is based on a main window that consists of a menu bar, a toolbar, a workspace area, and a status bar. The workspace area initially presents a Start Page tab, which provides NetBeans tutorials as well as news and blogs.

To help you get comfortable with the NetBeans user interface, I will show you how to create a `DumpArgs` project containing a single `DumpArgs.java` source file with Listing 1–1's source code. You will also learn how to compile and run this application.

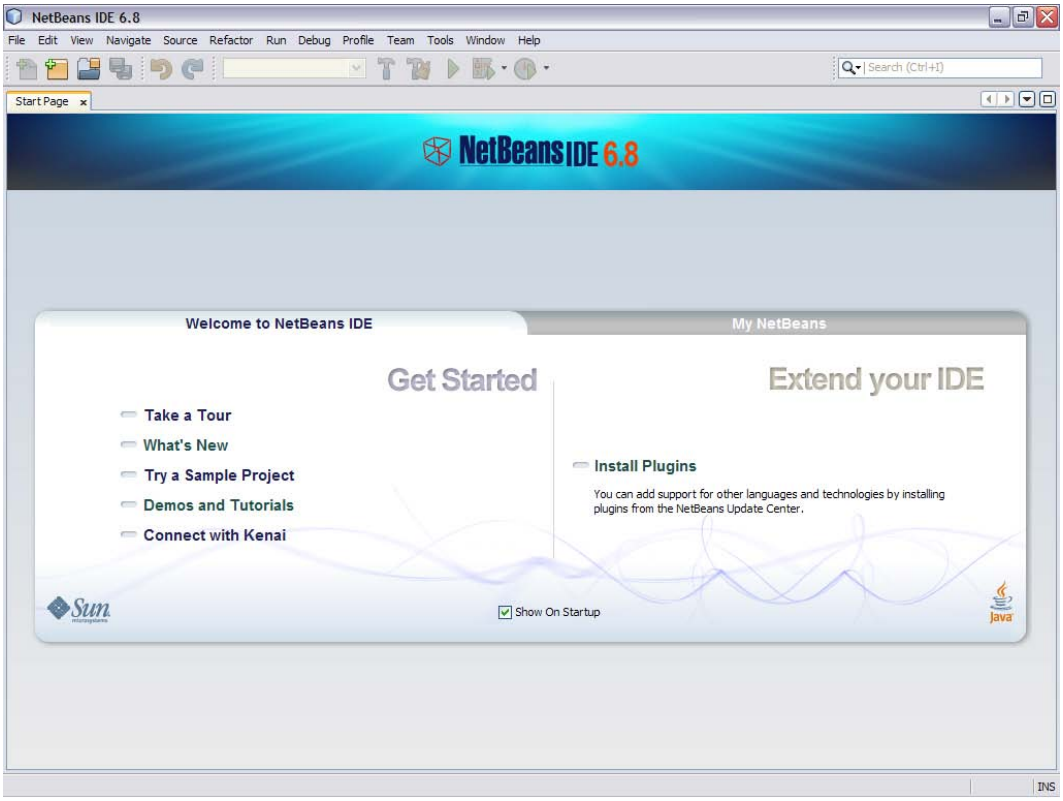

Figure 1–2. *The NetBeans IDE 6.8 main window*

Complete the following steps to create the `DumpArgs` project:

1. Select New Project from the File menu.
2. On the resulting New Project dialog box's initial pane, make sure that Java is the selected category in the Categories list, and Java Application is the selected project in the Projects list. Click the Next button.
3. On the resulting pane, enter **DumpArgs** in the Project Name text field. You will notice that dumpargs.Main appears in the text field to the right of the Create Main Class check box. Replace dumpargs.Main with **DumpArgs** and click Finish. (dumpargs names a package, discussed in Chapter 4, and Main names a class stored in this package.)

After a few moments, you will see a workspace similar to that shown in Figure 1–3.

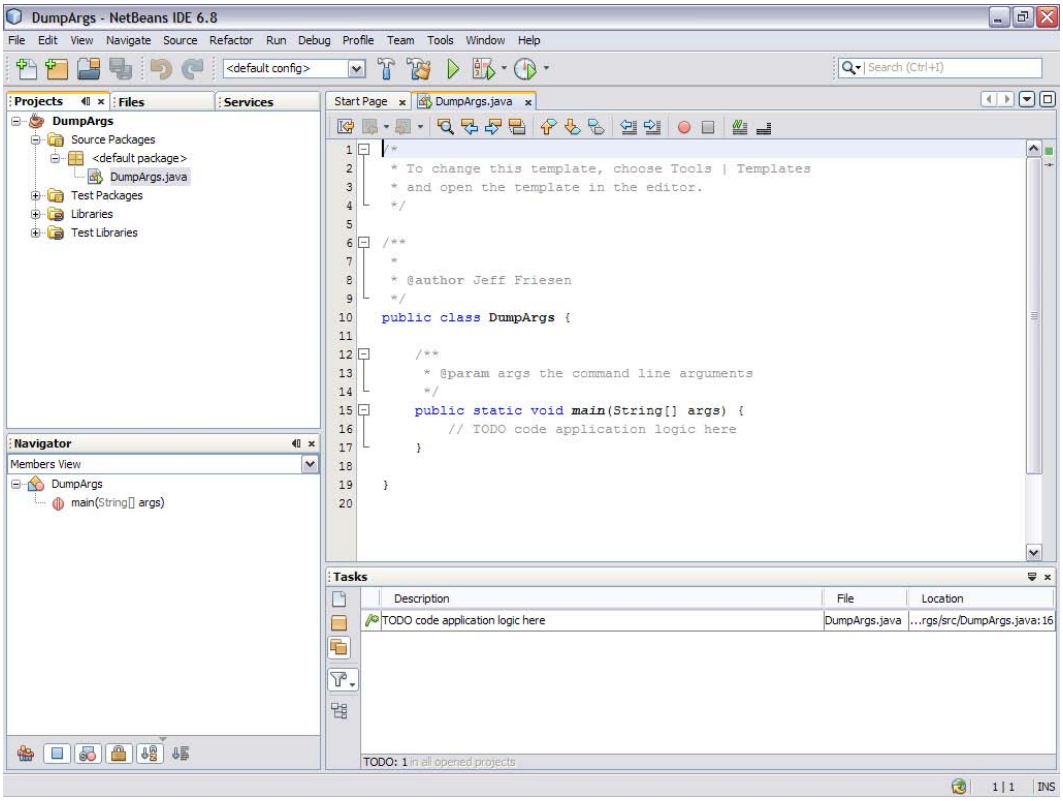

Figure 1–3. *The workspace is divided into multiple work areas.*

After creating the `DumpArgs` project, you will discover that NetBeans has organized the workspace into four main areas: projects, editor, navigator, and tasks.

The *projects area* helps you manage your projects. This window is divided into Projects, Files, and Services tabs:

- The Projects tab is the main entry point to your project's source and resource files. It presents a logical view of important project contents.
- The Files tab presents a directory-based view of your projects, including any files and folders that are not displayed on the Projects tab.
- The Services tab is the main entry point to runtime resources. It shows a logical view of important runtime resources such as the servers, databases, and web services that are registered with the IDE.

The *editor area* helps you edit a project's source files. Each file has its own tab, labeled with the file's name.

Figure 1–3 reveals a single DumpArgs.java tab, which provides access to skeletal source code. You will shortly replace this source code with Listing 1–1.

The skeletal source code reveals single-line and multiline comments (discussed in Chapter 2) and Javadoc comments (discussed later in this chapter).

The *navigator area* reveals the Navigator tab, which presents a compact view of the currently selected file and simplifies navigation between different parts of the file.

The *tasks area* reveals the Tasks tab, which presents a to-do list of items for the project's various files that need to be resolved.

Replace the skeletal DumpArgs.java source code with Listing 1–1, and select Run Main Project from the Run menu to compile and run this application. Figure 1–4 shows this application's results.

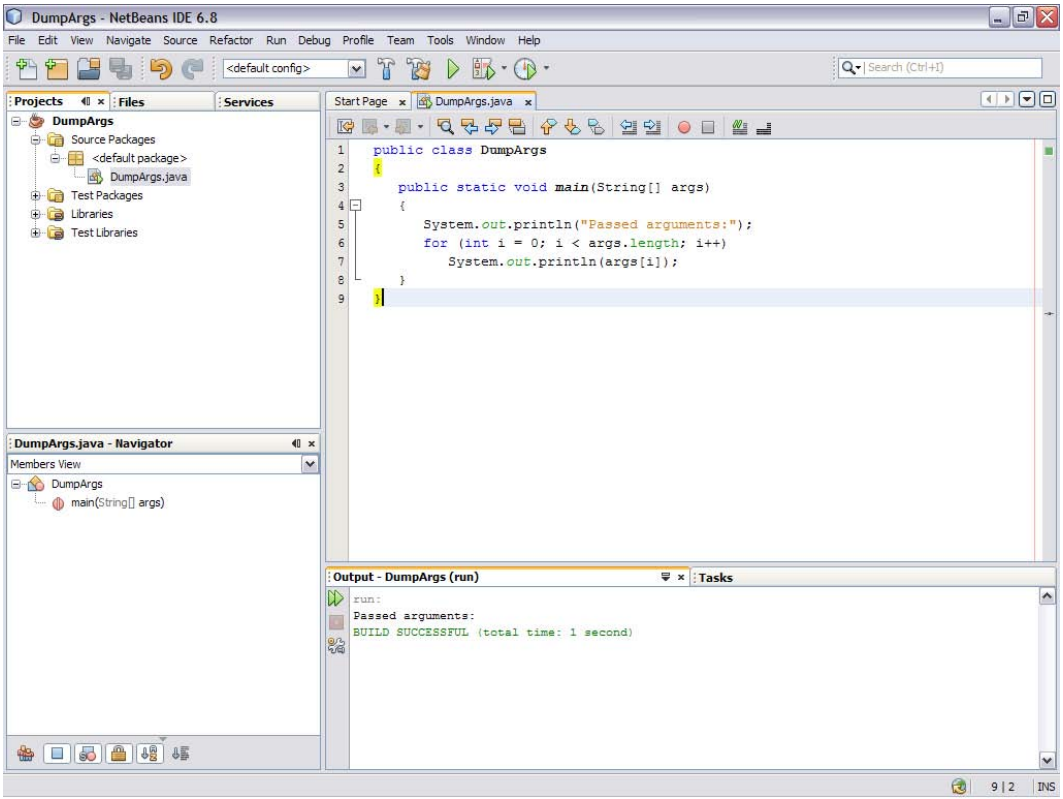

Figure 1–4. *An Output tab appears to the left of the Tasks tab and shows the* `DumpArgs` *application's output.*

Figure 1–4's Output tab reveals only the result of the `System.out.println("Passed arguments:")` method call. To see more output, you must pass command-line arguments to `DumpArgs`. Accomplish this task from within NetBeans IDE 6.8 as follows:

1. Select Project Properties (DumpArgs) from the File menu.
2. In the resulting Project Properties dialog box, select Run in the Categories tree and enter **Curly Moe Larry** in the Arguments text field on the resulting pane. Click the OK button.

Once again, select Run Main Project from the Run menu to run the DumpArgs application. This time, the Output tab should reveal Curly, Moe, and Larry on separate lines below Passed arguments:.

This is all I have to say about the NetBeans IDE. For more information, study the tutorials via the Start Page tab, access IDE help via the Help menu, and explore the NetBeans knowledge base at http://netbeans.org/kb/.

Eclipse IDE

Eclipse IDE is an open source IDE for developing programs in Java and other languages (such as C, COBOL, PHP, Perl, and Python). Eclipse Classic is one distribution of this IDE that is available for download; version 3.5.2 is the current version at time of writing.

You should download and install Eclipse Classic to follow along with this section's Eclipse-oriented example. Begin by pointing your browser to http://www.eclipse.org/downloads/ and accomplishing the following tasks:

1. Scroll down the page until you see an Eclipse Classic entry.
2. Click one of the platform links (such as Linux 32 Bit) to the right of this entry.
3. Select a download mirror from the subsequently displayed page and proceed to download the distribution's archive file.

I downloaded the approximately 163MB eclipse-SDK-3.5.2-win32.zip archive file for my Windows XP platform, unarchived this file, moved the resulting eclipse home directory to another location, and created a shortcut to that directory's eclipse.exe file.

> **NOTE:** Unlike NetBeans IDE 6.8, which requires that a suitable JDK be installed before you can run the installer, a JDK does not have to be installed before running eclipse.exe because the Eclipse IDE comes with its own Java compiler. However, you will need at least JDK 6 Update 16 to run most of this book's code (or JDK 7 to run all of the code).

Assuming that you have installed Eclipse Classic, start this application. You should discover a splash screen identifying this IDE and a dialog box that lets you choose the location of a *workspace* for storing projects, followed by a main window like that shown in Figure 1–5.

The Eclipse user interface is based on a main window that consists of a menu bar, a toolbar, a workbench area, and a status bar. The workbench area initially presents a Welcome tab with icon links for accessing tutorials and more.

To help you get comfortable with the Eclipse user interface, I will show you how to create a DumpArgs project containing a single DumpArgs.java source file with Listing 1–1's source code. You will also learn how to compile and run this application.

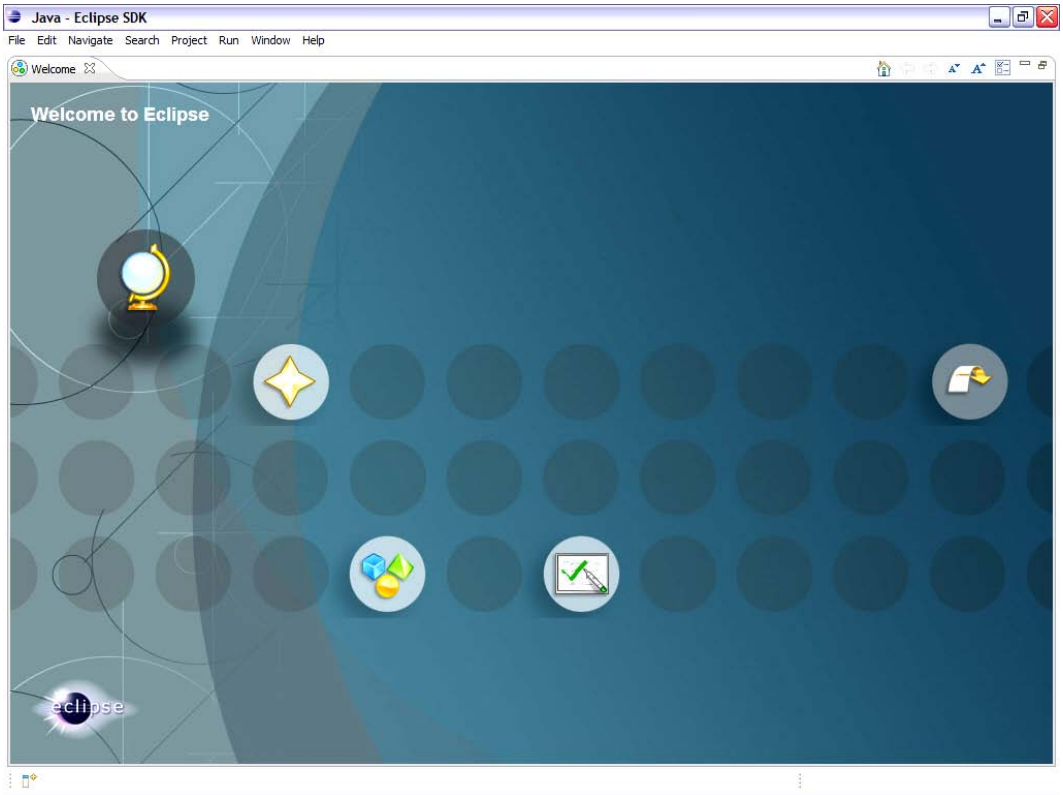

Figure 1–5. *The Eclipse IDE 3.5.2 main window*

Complete the following steps to create the `DumpArgs` project:

1. Select New from the File menu and Java Project from the resulting pop-up menu.
2. In the resulting New Java Project dialog box, enter **DumpArgs** into the Project name text field. Keep all the other defaults and click the Finish button.
3. Click the rightmost (Workbench) icon link to go to the workbench. Eclipse bypasses the Welcome tab and takes you to the workbench the next time you start this IDE.

TIP: To return to the Welcome tab, select Welcome from the Help menu.

After the final step, you will see a workbench similar to that shown in Figure 1–6.

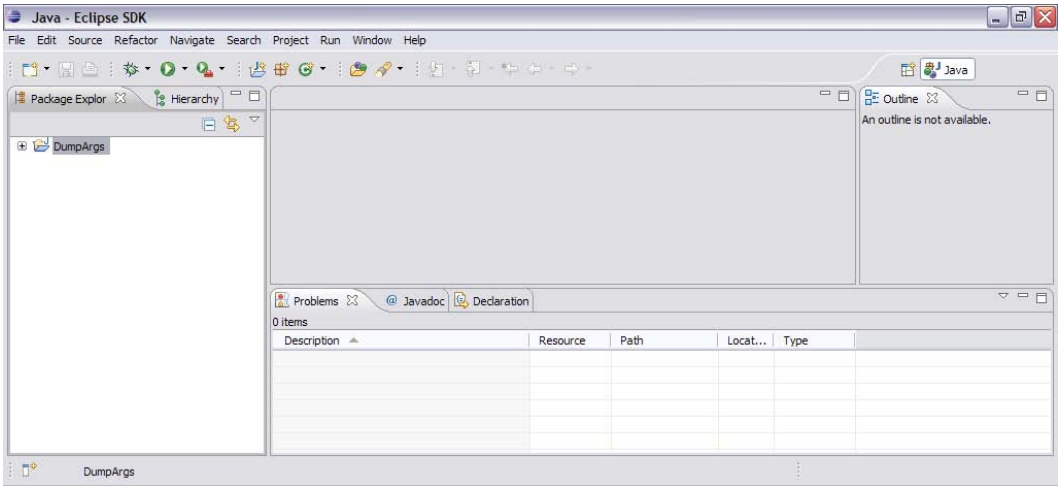

Figure 1–6. *The workbench is divided into multiple work areas.*

On the left side of the workbench, you see a tab titled Package Explorer. This tab identifies the workspace's projects in terms of packages (discussed in Chapter 4). At the moment, only a single DumpArgs entry appears on this tab.

Clicking the + icon to the left of DumpArgs expands this entry to reveal src and JRE System Library items. The src item stores the `DumpArgs` project's source files, and JRE System Library identifies various JRE files that are used to run this application.

We will now add a new file named `DumpArgs.java` to src, as follows:

1. Highlight src and select New from the File menu, and File from the resulting pop-up menu.

2. In the resulting New File dialog box, enter **DumpArgs.java** into the File name text field, and click the Finish button.

Eclipse responds by displaying an editor tab titled DumpArgs.java. Copy Listing 1–1 into this tab, and then compile and run this application by selecting Run from the Run menu. Figure 1–7 shows the results.

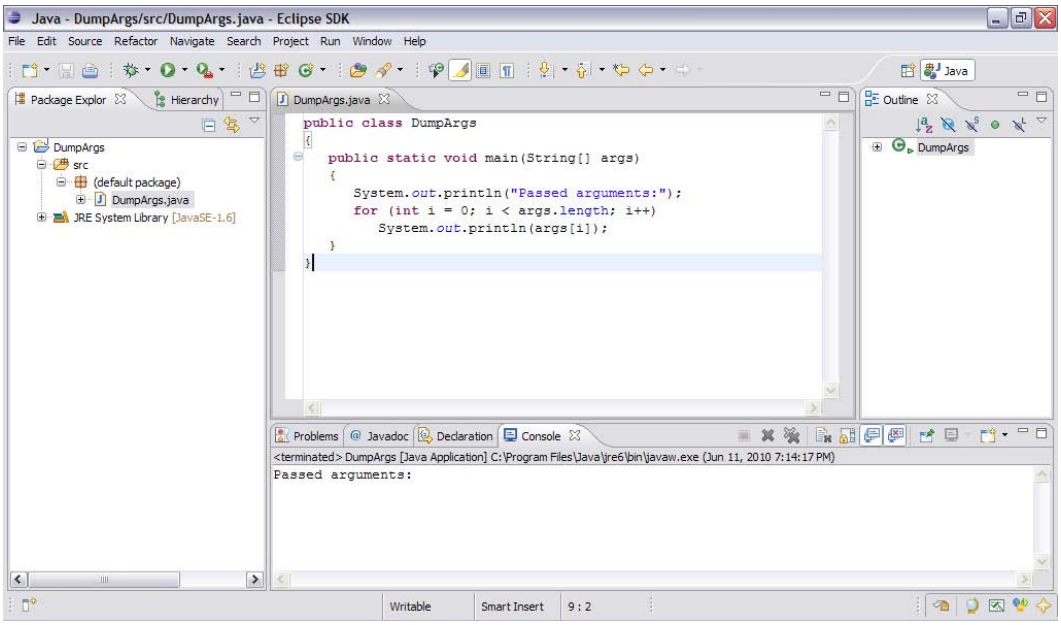

Figure 1–7. *The Console tab at the bottom of the workbench presents the* `DumpArgs` *application's output.*

As with the NetBeans IDE, you must pass command-line arguments to `DumpArgs` to see additional output from this application. Accomplish this task from within Eclipse IDE 3.5.2 as follows:

1. Select Run Configurations from the Run menu.
2. In the resulting Run Configurations dialog box, select the Arguments tab.
3. Enter **Curly Moe Larry** into the Program arguments text area and click the Close button.

Once again, select Run from the Run menu to run the `DumpArgs` application. This time, the Console tab reveals Curly, Moe, and Larry on separate lines below Passed arguments:.

This is all I have to say about the Eclipse IDE. For more information, study the tutorials via the Welcome tab, access IDE help via the Help menu, and explore the Eclipse documentation at `http://www.eclipse.org/documentation/`.

Four of a Kind

Application development is not an easy task. If you do not plan carefully before you develop an application, you will probably waste your time and money as you endeavor to create it, and waste your users' time and money if it does not meet their needs.

CAUTION: It is extremely important to carefully test your software. You could face a lawsuit if malfunctioning software causes financial harm to its users.

In this section, I present one technique for developing applications efficiently. I present this technique in the context of a Java application that lets you play a simple card game called *Four of a Kind* against the computer.

Understanding Four of a Kind

Before sitting down at the computer and writing code, we need to fully understand the *problem domain* that we are trying to model via that code. In this case, the problem domain is *Four of a Kind*, and we want to understand how this card game works.

Two to four players play *Four of a Kind* with a standard 52-card deck. The object of the game is to be the first player to put down four cards that have the same rank (four aces, for example), which wins the game.

The game begins by shuffling the deck and placing it face down. Each player takes a card from the top of the deck. The player with the highest ranked card (king is highest) deals four cards to each player, starting with the player to the dealer's left. The dealer then starts his/her turn.

The player examines his/her cards to determine which cards are optimal for achieving four of a kind. The player then throws away the least helpful card on a discard pile and picks up another card from the top of the deck. (If each card has a different rank, the player randomly selects a card to throw away.) If the player has four of a kind, the player puts down these cards (face up) and wins the game.

Modeling Four of a Kind in Pseudocode

Now that we understand how *Four of a Kind* works, we can begin to model this game. We will not model the game in Java source code because we would get bogged down in too many details. Instead, we will use pseudocode for this task.

Pseudocode is a compact and informal high-level description of the problem domain. Unlike the previous description of *Four of a Kind*, the pseudocode equivalent is a step-by-step recipe for solving the problem. Check out Listing 1–3.

Listing 1–3. *Four of a Kind pseudocode for two players (human and computer)*

```
1. Create a deck of cards and shuffle the deck.
2. Create empty discard pile.
3. Have each of the human and computer players take a card from the top of the deck.
4. Designate the player with the highest ranked card as the current player.
5. Return both cards to the bottom of the deck.
6. The current player deals four cards to each of the two players in alternating
fashion, with the first card being dealt to the other player.
```

```
 7. The current player examines its current cards to see which cards are optimal for
achieving four of a kind. The current player throws the least helpful card onto the top
of the discard pile.
 8. The current player picks up the deck's top card. If the current player has four of a
kind, it puts down its cards and wins the game.
 9. Designate the other player as the current player.
10. If the deck has no more cards, empty the discard pile to the deck and shuffle the
deck.
11. Repeat at step 7.
```

Deriving Listing 1–3's pseudocode from the previous description is the first step in achieving an application that implements *Four of a Kind*. This pseudocode performs various tasks, including decision making and repetition.

Despite being a more useful guide to understanding how *Four of a Kind* works, Listing 1–3 is too high level for translation to Java. Therefore, we must refine this pseudocode to facilitate the translation process. Listing 1–4 presents this refinement.

Listing 1–4. *Refined Four of a Kind pseudocode for two players (human and computer)*

```
 1. deck = new Deck()
 2. deck.shuffle()
 3. discardPile = new DiscardPile()
 4. hCard = deck.deal()
 5. cCard = deck.deal()
 6. if hCard.rank() == cCard.rank()
    6.1. deck.putBack(hCard)
    6.2. deck.putBack(cCard)
    6.3. deck.shuffle()
    6.4. Repeat at step 4
 7. curPlayer = HUMAN
    7.1. if cCard.rank() > hCard.rank()
         7.1.1. curPlayer = COMPUTER
 8. deck.putBack(hCard)
 9. deck.putBack(cCard)
10. if curPlayer == HUMAN
    10.1. for i = 0 to 3
          10.1.1. cCards[i] = deck.deal()
          10.1.2. hCards[i] = deck.deal()
    else
    10.2. for i = 0 to 3
          10.2.1. hCards[i] = deck.deal()
          10.2.2. cCards[i] = deck.deal()
11. if curPlayer == HUMAN
    11.01. output(hCards)
    11.02. choice = prompt("Identify card to throw away")
    11.03. discardPile.setTopCard(hCards[choice])
    11.04. hCards[choice] = deck.deal()
    11.05. if isFourOfAKind(hCards)
           11.05.1. output("Human wins!")
           11.05.2. putDown(hCards)
           11.05.3. output("Computer's cards:")
           11.05.4. putDown(cCards)
           11.05.5. End game
    11.06. curPlayer = COMPUTER
```

```
    else
    11.07. choice = leastDesirableCard(cCards)
    11.08. discardPile.setTopCard(cCards[choice])
    11.09. cCards[choice] = deck.deal()
    11.10. if isFourOfAKind(cCards)
             11.10.1. output("Computer wins!")
             11.10.2. putDown(cCards)
             11.10.3. End game
    11.11. curPlayer = HUMAN
12. if deck.isEmpty()
    12.1. if discardPile.topCard() != null
            12.1.1. deck.putBack(discardPile.getTopCard())
            12.1.2. Repeat at step 12.1.
    12.2. deck.shuffle()
13. Repeat at step 11.
```

In addition to being longer than Listing 1–3, Listing 1–4 shows the refined pseudocode becoming more like Java. For example, Listing 1–4 reveals Java expressions (such as `new Deck()`, to create a `Deck` object), operators (such as `==`, to compare two values for equality), and method calls (such as `deck.isEmpty()`, to call `deck`'s `isEmpty()` method to return a Boolean value indicating whether [true] or not [false] the deck identified by `deck` is empty of cards).

Converting Pseudocode to Java Code

Now that you have had a chance to absorb Listing 1–4's Java-like pseudocode, you are ready to examine the process of converting that pseudocode to Java source code. This process consists of a couple of steps.

The first step in converting Listing 1–4's pseudocode to Java involves identifying important components of the game's structure and implementing these components as classes. I will formally introduce classes in Chapter 2.

Apart from the computer player (which is implemented via game logic), the important components are card, deck, and discard pile. I represent these components via `Card`, `Deck`, and `DiscardPile` classes. Listing 1–5 presents `Card`.

> **NOTE:** Do not be concerned if you find this section's Java source code somewhat hard to follow. After you have read the next few chapters, you should find this code easier to understand.

Listing 1–5. *Merging suits and ranks into cards*

```
/**
 *  Simulating a playing card.
 *
 *  @author Jeff Friesen
 */
public enum Card
{
   ACE_OF_CLUBS(Suit.CLUBS, Rank.ACE),
   TWO_OF_CLUBS(Suit.CLUBS, Rank.TWO),
```

```
THREE_OF_CLUBS(Suit.CLUBS, Rank.THREE),
FOUR_OF_CLUBS(Suit.CLUBS, Rank.FOUR),
FIVE_OF_CLUBS(Suit.CLUBS, Rank.FIVE),
SIX_OF_CLUBS(Suit.CLUBS, Rank.SIX),
SEVEN_OF_CLUBS(Suit.CLUBS, Rank.SEVEN),
EIGHT_OF_CLUBS(Suit.CLUBS, Rank.EIGHT),
NINE_OF_CLUBS(Suit.CLUBS, Rank.NINE),
TEN_OF_CLUBS(Suit.CLUBS, Rank.TEN),
JACK_OF_CLUBS(Suit.CLUBS, Rank.JACK),
QUEEN_OF_CLUBS(Suit.CLUBS, Rank.QUEEN),
KING_OF_CLUBS(Suit.CLUBS, Rank.KING),
ACE_OF_DIAMONDS(Suit.DIAMONDS, Rank.ACE),
TWO_OF_DIAMONDS(Suit.DIAMONDS, Rank.TWO),
THREE_OF_DIAMONDS(Suit.DIAMONDS, Rank.THREE),
FOUR_OF_DIAMONDS(Suit.DIAMONDS, Rank.FOUR),
FIVE_OF_DIAMONDS(Suit.DIAMONDS, Rank.FIVE),
SIX_OF_DIAMONDS(Suit.DIAMONDS, Rank.SIX),
SEVEN_OF_DIAMONDS(Suit.DIAMONDS, Rank.SEVEN),
EIGHT_OF_DIAMONDS(Suit.DIAMONDS, Rank.EIGHT),
NINE_OF_DIAMONDS(Suit.DIAMONDS, Rank.NINE),
TEN_OF_DIAMONDS(Suit.DIAMONDS, Rank.TEN),
JACK_OF_DIAMONDS(Suit.DIAMONDS, Rank.JACK),
QUEEN_OF_DIAMONDS(Suit.DIAMONDS, Rank.QUEEN),
KING_OF_DIAMONDS(Suit.DIAMONDS, Rank.KING),
ACE_OF_HEARTS(Suit.HEARTS, Rank.ACE),
TWO_OF_HEARTS(Suit.HEARTS, Rank.TWO),
THREE_OF_HEARTS(Suit.HEARTS, Rank.THREE),
FOUR_OF_HEARTS(Suit.HEARTS, Rank.FOUR),
FIVE_OF_HEARTS(Suit.HEARTS, Rank.FIVE),
SIX_OF_HEARTS(Suit.HEARTS, Rank.SIX),
SEVEN_OF_HEARTS(Suit.HEARTS, Rank.SEVEN),
EIGHT_OF_HEARTS(Suit.HEARTS, Rank.EIGHT),
NINE_OF_HEARTS(Suit.HEARTS, Rank.NINE),
TEN_OF_HEARTS(Suit.HEARTS, Rank.TEN),
JACK_OF_HEARTS(Suit.HEARTS, Rank.JACK),
QUEEN_OF_HEARTS(Suit.HEARTS, Rank.QUEEN),
KING_OF_HEARTS(Suit.HEARTS, Rank.KING),
ACE_OF_SPADES(Suit.SPADES, Rank.ACE),
TWO_OF_SPADES(Suit.SPADES, Rank.TWO),
THREE_OF_SPADES(Suit.SPADES, Rank.THREE),
FOUR_OF_SPADES(Suit.SPADES, Rank.FOUR),
FIVE_OF_SPADES(Suit.SPADES, Rank.FIVE),
SIX_OF_SPADES(Suit.SPADES, Rank.SIX),
SEVEN_OF_SPADES(Suit.SPADES, Rank.SEVEN),
EIGHT_OF_SPADES(Suit.SPADES, Rank.EIGHT),
NINE_OF_SPADES(Suit.SPADES, Rank.NINE),
TEN_OF_SPADES(Suit.SPADES, Rank.TEN),
JACK_OF_SPADES(Suit.SPADES, Rank.JACK),
QUEEN_OF_SPADES(Suit.SPADES, Rank.QUEEN),
KING_OF_SPADES(Suit.SPADES, Rank.KING);

private Suit suit;
/**
 *  Return <code>Card</code>'s suit.
 *
 *  @return <code>CLUBS</code>, <code>DIAMONDS</code>, <code>HEARTS</code>,
 *  or <code>SPADES</code>
```

```
    */
   public Suit suit() { return suit; }
   private Rank rank;
   /**
    *  Return <code>Card</code>'s rank.
    *
    *  @return <code>ACE</code>, <code>TWO</code>, <code>THREE</code>,
    *  <code>FOUR</code>, <code>FIVE</code>, <code>SIX</code>,
    *  <code>SEVEN</code>, <code>EIGHT</code>, <code>NINE</code>,
    *  <code>TEN</code>, <code>JACK</code>, <code>QUEEN</code>,
    *  <code>KING</code>.
    */
   public Rank rank() { return rank; }
   Card(Suit suit, Rank rank)
   {
      this.suit = suit;
      this.rank = rank;
   }
   /**
    *  A card's suit is its membership.
    *
    *  @author Jeff Friesen
    */
   public enum Suit
   {
      CLUBS, DIAMONDS, HEARTS, SPADES
   }
   /**
    *  A card's rank is its integer value.
    *
    *  @author Jeff Friesen
    */
   public enum Rank
   {
      ACE, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN, JACK, QUEEN,
      KING
   }
}
```

Listing 1–5 begins with a Javadoc comment that is used to briefly describe the `Card` class and identify this class's author. I will introduce you to Javadoc comments at the end of this section.

> **NOTE:** One feature of Javadoc comments is the ability to embed HTML tags. These tags specify different kinds of formatting for sections of text within these comments. For example, `<code>` and `</code>` specify that their enclosed text is to be formatted as a code listing. Later in this chapter, you will learn how to convert these comments into HTML documentation.

`Card` is an example of an enum, which is a special kind of class that I discuss in Chapter 5. For now, think of `Card` as a place to create and store `Card` objects that identify all 52 cards that make up a standard deck.

Card declares a nested Suit enum. (I discuss nesting in Chapter 4.) A card's *suit* denotes its membership. The only legal Suit values are CLUBS, DIAMONDS, HEARTS, and SPADES.

Card also declares a nested Rank enum. A card's *rank* denotes its value: ACE, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN, JACK, QUEEN, and KING are the only legal Rank values.

A Card object is created when Suit and Rank objects are passed to its constructor. (I discuss constructors in Chapter 2.) For example, KING_OF_HEARTS(Suit.HEARTS, Rank.KING) combines Suit.HEARTS and Rank.KING into KING_OF_HEARTS.

Card provides a rank() method for returning a Card's Rank object. Similarly, Card provides a suit() method for returning a Card's Suit object. For example, KING_OF_HEARTS.rank() returns Rank.KING, and KING_OF_HEARTS.suit() returns Suit.HEARTS.

Listing 1–6 presents the Java source code to the Deck class, which implements a deck of 52 cards.

Listing 1–6. *Pick a card, any card*

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

/**
 *  Simulate a deck of cards.
 *
 *  @author Jeff Friesen
 */
public class Deck
{
   private Card[] cards = new Card[]
   {
      Card.ACE_OF_CLUBS,
      Card.TWO_OF_CLUBS,
      Card.THREE_OF_CLUBS,
      Card.FOUR_OF_CLUBS,
      Card.FIVE_OF_CLUBS,
      Card.SIX_OF_CLUBS,
      Card.SEVEN_OF_CLUBS,
      Card.EIGHT_OF_CLUBS,
      Card.NINE_OF_CLUBS,
      Card.TEN_OF_CLUBS,
      Card.JACK_OF_CLUBS,
      Card.QUEEN_OF_CLUBS,
      Card.KING_OF_CLUBS,
      Card.ACE_OF_DIAMONDS,
      Card.TWO_OF_DIAMONDS,
      Card.THREE_OF_DIAMONDS,
      Card.FOUR_OF_DIAMONDS,
      Card.FIVE_OF_DIAMONDS,
      Card.SIX_OF_DIAMONDS,
      Card.SEVEN_OF_DIAMONDS,
      Card.EIGHT_OF_DIAMONDS,
      Card.NINE_OF_DIAMONDS,
```

```
      Card.TEN_OF_DIAMONDS,
      Card.JACK_OF_DIAMONDS,
      Card.QUEEN_OF_DIAMONDS,
      Card.KING_OF_DIAMONDS,
      Card.ACE_OF_HEARTS,
      Card.TWO_OF_HEARTS,
      Card.THREE_OF_HEARTS,
      Card.FOUR_OF_HEARTS,
      Card.FIVE_OF_HEARTS,
      Card.SIX_OF_HEARTS,
      Card.SEVEN_OF_HEARTS,
      Card.EIGHT_OF_HEARTS,
      Card.NINE_OF_HEARTS,
      Card.TEN_OF_HEARTS,
      Card.JACK_OF_HEARTS,
      Card.QUEEN_OF_HEARTS,
      Card.KING_OF_HEARTS,
      Card.ACE_OF_SPADES,
      Card.TWO_OF_SPADES,
      Card.THREE_OF_SPADES,
      Card.FOUR_OF_SPADES,
      Card.FIVE_OF_SPADES,
      Card.SIX_OF_SPADES,
      Card.SEVEN_OF_SPADES,
      Card.EIGHT_OF_SPADES,
      Card.NINE_OF_SPADES,
      Card.TEN_OF_SPADES,
      Card.JACK_OF_SPADES,
      Card.QUEEN_OF_SPADES,
      Card.KING_OF_SPADES
   };
   private List<Card> deck;
   /**
    *  Create a <code>Deck</code> of 52 <code>Card</code> objects. Shuffle
    *  these objects.
    */
   public Deck()
   {
      deck = new ArrayList<Card>();
      for (int i = 0; i < cards.length; i++)
      {
         deck.add(cards[i]);
         cards[i] = null;
      }
      Collections.shuffle(deck);
   }
   /**
    *  Deal the <code>Deck</code>'s top <code>Card</code> object.
    *
    *  @return the <code>Card</code> object at the top of the
    *  <code>Deck</code>
    */
   public Card deal()
   {
      return deck.remove(0);
   }
   /**
```

```
    *  Return an indicator of whether or not the <code>Deck</code> is empty.
    *
    *  @return true if the <code>Deck</code> contains no <code>Card</code>
    *  objects; otherwise, false
    */
   public boolean isEmpty()
   {
      return deck.isEmpty();
   }
   /**
    *  Put back a <code>Card</code> at the bottom of the <code>Deck</code>.
    *
    *  @param card <code>Card</code> object being put back
    */
   public void putBack(Card card)
   {
      deck.add(card);
   }
   /**
    *  Shuffle the <code>Deck</code>.
    */
   public void shuffle()
   {
      Collections.shuffle(deck);
   }
}
```

`Deck` initializes a private `cards` array to all 52 `Card` objects. Because it is easier to implement `Deck` via a list that stores these objects, `Deck`'s constructor creates this list and adds each `Card` object to the list. (I discuss `List` and `ArrayList` in Chapter 8.)

`Deck` also provides `deal()`, `isEmpty()`, `putBack()`, and `shuffle()` methods to deal a single `Card` from the `Deck` (the `Card` is physically removed from the `Deck`), determine whether or not the `Deck` is empty, put a `Card` back into the `Deck`, and shuffle the `Deck`'s `Card`s.

Listing 1–7 presents the source code to the `DiscardPile` class, which implements a discard pile on which players can throw away a maximum of 52 cards.

Listing 1–7. *A garbage dump for cards*

```
/**
 *  Simulate a pile of discarded cards.
 *
 *  @author Jeff Friesen
 */
public class DiscardPile
{
   private Card[] cards;
   private int top;
   /**
    *  Create a <code>DiscardPile</code> that can accommodate a maximum of 52
    *  <code>Card</code>s. The <code>DiscardPile</code> is initially empty.
    */
   public DiscardPile()
   {
      cards = new Card[52]; // Room for entire deck on discard pile (should
```

```
                              // never happen).
      top = -1;
   }
   /**
    *  Return the <code>Card</code> at the top of the <code>DiscardPile</code>.
    *
    *  @return <code>Card</code> object at top of <code>DiscardPile</code> or
    *  null if <code>DiscardPile</code> is empty
    */
   public Card getTopCard()
   {
      if (top == -1)
         return null;
      Card card = cards[top];
      cards[top--] = null;
      return card;
   }
   /**
    *  Set the <code>DiscardPile</code>'s top card to the specified
    *  <code>Card</code> object.
    *
    *  @param card <code>Card</code> object being thrown on top of the
    *  <code>DiscardPile</code>
    */
   public void setTopCard(Card card)
   {
      cards[++top] = card;
   }
   /**
    *  Identify the top <code>Card</code> on the <code>DiscardPile</code>
    *  without removing this <code>Card</code>.
    *
    *  @return top <code>Card</code>, or null if <code>DiscardPile</code> is
    *  empty
    */
   public Card topCard()
   {
      return (top == -1) ? null : cards[top];
   }
}
```

`DiscardPile` implements a discard pile on which to throw `Card` objects. It implements the discard pile via a stack metaphor: the last `Card` object thrown on the pile sits at the top of the pile and is the first `Card` object to be removed from the pile.

This class stores its stack of `Card` objects in a private `cards` array. I found it convenient to specify 52 as this array's storage limit because the maximum number of `Card`s is 52. (Game play will never result in all `Card`s being stored in the array.)

Along with its constructor, `DiscardPile` provides `getTopCard()`, `setTopCard()`, and `topCard()` methods to remove and return the stack's top `Card`, store a new `Card` object on the stack as its top `Card`, and return the top `Card` without removing it from the stack.

The constructor demonstrates a single-line comment, which starts with the `//` character sequence. This comment documents that the `cards` array has room to store the entire `Deck` of `Card`s. I will formally introduce single-line comments in Chapter 2.

The second step in converting Listing 1–4's pseudocode to Java involves introducing a `FourOfAKind` class whose `main()` method contains the Java code equivalent of this pseudocode. Listing 1–8 presents `FourOfAKind`.

Listing 1–8. *`FourOfAKind` application source code*

```
/**
 *  <code>FourOfAKind</code> implements a card game that is played between two
 *  players: one human player and the computer. You play this game with a
 *  standard 52-card deck and attempt to beat the computer by being the first
 *  player to put down four cards that have the same rank (four aces, for
 *  example), and win.
 *
 *  <p>
 *  The game begins by shuffling the deck and placing it face down. Each
 *  player takes a card from the top of the deck. The player with the highest
 *  ranked card (king is highest) deals four cards to each player starting
 *  with the other player. The dealer then starts its turn.
 *
 *  <p>
 *  The player examines its cards to determine which cards are optimal for
 *  achieving four of a kind. The player then throws away one card on a
 *  discard pile and picks up another card from the top of the deck. If the
 *  player has four of a kind, the player puts down these cards (face up) and
 *  wins the game.
 *
 *  @author Jeff Friesen
 *  @version 1.0
 */
public class FourOfAKind
{
   /**
    *  Human player
    */
   final static int HUMAN = 0;
   /**
    *  Computer player
    */
   final static int COMPUTER = 1;
   /**
    *  Application entry point.
    *
    *  @param args array of command-line arguments passed to this method
    */
   public static void main(String[] args)
   {
      System.out.println("Welcome to Four of a Kind!");
      Deck deck = new Deck(); // Deck automatically shuffled
      DiscardPile discardPile = new DiscardPile();
      Card hCard;
      Card cCard;
      while (true)
      {
         hCard = deck.deal();
         cCard = deck.deal();
         if (hCard.rank() != cCard.rank())
            break;
```

```
      deck.putBack(hCard);
      deck.putBack(cCard);
      deck.shuffle(); // prevent pathological case where every successive
   }                  // pair of cards have the same rank
   int curPlayer = HUMAN;
   if (cCard.rank().ordinal() > hCard.rank().ordinal())
      curPlayer = COMPUTER;
   deck.putBack(hCard);
   hCard = null;
   deck.putBack(cCard);
   cCard = null;
   Card[] hCards = new Card[4];
   Card[] cCards = new Card[4];
   if (curPlayer == HUMAN)
      for (int i = 0; i < 4; i++)
      {
         cCards[i] = deck.deal();
         hCards[i] = deck.deal();
      }
   else
      for (int i = 0; i < 4; i++)
      {
         hCards[i] = deck.deal();
         cCards[i] = deck.deal();
      }
   while (true)
   {
      if (curPlayer == HUMAN)
      {
         showHeldCards(hCards);
         int choice = 0;
         while (choice < 'A' || choice > 'D')
         {
            choice = prompt("Which card do you want to throw away (A, B, " +
                            "C, D)? ");
            switch (choice)
            {
               case 'a': choice = 'A'; break;
               case 'b': choice = 'B'; break;
               case 'c': choice = 'C'; break;
               case 'd': choice = 'D';
            }
         }
         discardPile.setTopCard(hCards[choice-'A']);
         hCards[choice-'A'] = deck.deal();
         if (isFourOfAKind(hCards))
         {
            System.out.println();
            System.out.println("Human wins!");
            System.out.println();
            putDown("Human's cards:", hCards);
            System.out.println();
            putDown("Computer's cards:", cCards);
            return; // Exit application by returning from main()
         }
         curPlayer = COMPUTER;
      }
```

```
      else
      {
         int choice = leastDesirableCard(cCards);
         discardPile.setTopCard(cCards[choice]);
         cCards[choice] = deck.deal();
         if (isFourOfAKind(cCards))
         {
            System.out.println();
            System.out.println("Computer wins!");
            System.out.println();
            putDown("Computer's cards:", cCards);
            return; // Exit application by returning from main()
         }
         curPlayer = HUMAN;
      }
      if (deck.isEmpty())
      {
         while (discardPile.topCard() != null)
            deck.putBack(discardPile.getTopCard());
         deck.shuffle();
      }
   }
}
/**
 *  Determine if the <code>Card</code> objects passed to this method all
 *  have the same rank.
 *
 *  @param cards array of <code>Card</code> objects passed to this method
 *
 *  @return true if all <code>Card</code> objects have the same rank;
 *  otherwise, false
 */
static boolean isFourOfAKind(Card[] cards)
{
   for (int i = 1; i < cards.length; i++)
      if (cards[i].rank() != cards[0].rank())
         return false;
   return true;
}
/**
 *  Identify one of the <code>Card</code> objects that is passed to this
 *  method as the least desirable <code>Card</code> object to hold onto.
 *
 *  @param cards array of <code>Card</code> objects passed to this method
 *
 *  @return 0-based rank (ace is 0, king is 13) of least desirable card
 */
static int leastDesirableCard(Card[] cards)
{
   int[] rankCounts = new int[13];
   for (int i = 0; i < cards.length; i++)
      rankCounts[cards[i].rank().ordinal()]++;
   int minCount = Integer.MAX_VALUE;
   int minIndex = -1;
   for (int i = 0; i < rankCounts.length; i++)
      if (rankCounts[i] < minCount && rankCounts[i] != 0)
      {
```

```
            minCount = rankCounts[i];
            minIndex = i;
         }
      for (int i = 0; i < cards.length; i++)
         if (cards[i].rank().ordinal() == minIndex)
            return i;
      return 0; // Needed to satisfy compiler (should never be executed)
   }
   /**
    *  Prompt the human player to enter a character.
    *
    *  @param msg message to be displayed to human player
    *
    *  @return integer value of character entered by user.
    */
   static int prompt(String msg)
   {
      System.out.print(msg);
      try
      {
         int ch = System.in.read();
         // Erase all subsequent characters including terminating \n newline
         // so that they do not affect a subsequent call to prompt().
         while (System.in.read() != '\n');
         return ch;
      }
      catch (java.io.IOException ioe)
      {
      }
      return 0;
   }
   /**
    *  Display a message followed by all cards held by player. This output
    *  simulates putting down held cards.
    *
    *  @param msg message to be displayed to human player
    *  @param cards array of <code>Card</code> objects to be identified
    */
   static void putDown(String msg, Card[] cards)
   {
      System.out.println(msg);
      for (int i = 0; i < cards.length; i++)
         System.out.println(cards[i]);
   }
   /**
    *  Identify the cards being held via their <code>Card</code> objects on
    *  separate lines. Prefix each line with an uppercase letter starting with
    *  <code>A</code>.
    *
    *  @param cards array of <code>Card</code> objects to be identified
    */
   static void showHeldCards(Card[] cards)
   {
      System.out.println();
      System.out.println("Held cards:");
      for (int i = 0; i < cards.length; i++)
         if (cards[i] != null)
```

```
            System.out.println((char) ('A'+i) + ". " + cards[i]);
        System.out.println();
    }
}
```

Listing 1–8 follows the steps outlined by and expands on Listing 1–4's pseudocode. Because of the various comments, I do not have much to say about this listing. However, there are a couple of items that deserve mention:

- `Card`'s nested `Rank` enum stores a sequence of 13 `Rank` objects beginning with `ACE` and ending with `KING`. These objects cannot be compared directly via `>` to determine which object has the greater rank. However, their integer-based *ordinal* (positional) values can be compared by calling the `Rank` object's `ordinal()` method. For example, `Card.ACE_OF_SPADES.rank().ordinal()` returns 0 because `ACE` is located at position 0 within `Rank`'s list of `Rank` objects, and `Card.KING_OF_CLUBS.rank().ordinal()` returns 12 because `KING` is located at the last position in this list.
- The `leastDesirableCard()` method counts the ranks of the `Card`s in the array of `Card` objects passed to this method, and stores these counts in a `rankCounts` array. For example, given two of clubs, ace of spades, three of clubs, and ace of diamonds, this array identifies one two, two aces, and one three. This method then searches `rankCounts` from smallest index (representing ace) to largest index (representing king) for the first smallest nonzero count (there might be a tie, as in one two and one three)—a zero count represents no `Card`s having that rank in the array of `Card` objects. Finally, the method searches the array of `Card` objects to identify the object whose rank ordinal matches the index of the smallest nonzero count, and returns the index of this `Card` object.

  This behavior implies that the least desirable card is always the smallest ranked card. Furthermore, if there are multiple cards of the same rank, and if this rank is smaller than the rank of another card in the array, this method will choose the first (in a left-to-right manner) of the multiple cards having the same rank as the least desirable card. However, if the rank of the multiple cards is greater than the rank of another card, this other card will be chosen as least desirable.

I previously stated that Listing 1–5 begins with a Javadoc comment that describes the `Card` class and identifies this class's author. A *Javadoc comment* is a documentation item that documents a class, a method, or other program entity.

A Javadoc comment begins with `/**` and ends with `*/`. Sandwiched between these *delimiters* (a pair of characters that mark the start and stop of some section) are text, HTML tags (such as `<p>` and `<code>`), and *Javadoc tags*, which are `@`-prefixed instructions. The following list identifies three common tags:

- `@author` identifies the source code's author.
- `@param` identifies one of a method's parameters (discussed in Chapter 2).
- `@return` identifies the kind of value that a method returns.

The JDK provides a `javadoc` tool that extracts all Javadoc comments from one or more source files and generates a set of HTML files containing this documentation in an easy-to-read format. These files serve as the program's documentation.

For example, suppose that the current directory contains `Card.java`, `Deck.java`, `DiscardPile.java`, and `FourOfAKind.java`. To extract all of the Javadoc comments that appear in these files, specify the following command:

```
javadoc *.java
```

The `javadoc` tool responds by outputting the following messages:

```
Loading source file Card.java...
Loading source file Deck.java...
Loading source file DiscardPile.java...
Loading source file FourOfAKind.java...
Constructing Javadoc information...
Standard Doclet version 1.6.0_16
Building tree for all the packages and classes...
Generating Card.html...
Generating Card.Rank.html...
Generating Card.Suit.html...
Generating Deck.html...
Generating DiscardPile.html...
Generating FourOfAKind.html...
Generating package-frame.html...
Generating package-summary.html...
Generating package-tree.html...
Generating constant-values.html...
Building index for all the packages and classes...
Generating overview-tree.html...
Generating index-all.html...
Generating deprecated-list.html...
Building index for all classes...
Generating allclasses-frame.html...
Generating allclasses-noframe.html...
Generating index.html...
Generating help-doc.html...
Generating stylesheet.css...
```

Furthermore, it generates a series of files, including the `index.html` entry-point file. If you point your web browser to this file, you should see a page that is similar to the page shown in Figure 1–8.

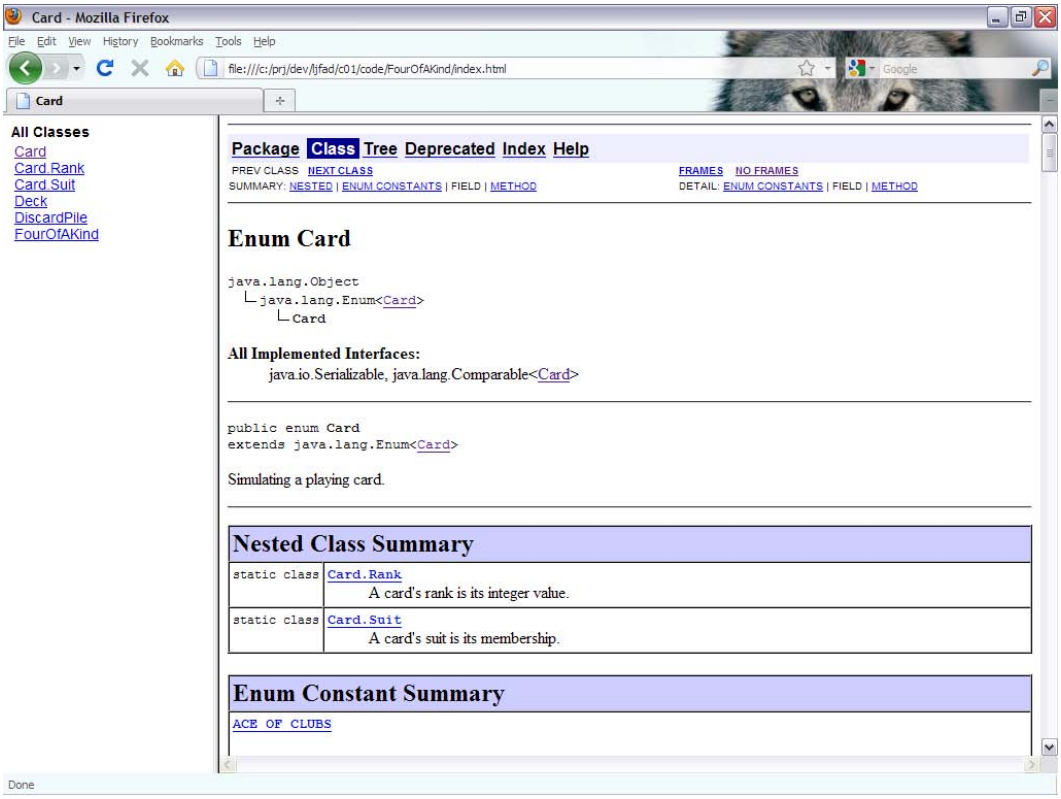

Figure 1–8. *The entry-point page into the generated Javadoc for* `FourOfAKind` *and supporting classes*

`javadoc` defaults to generating HTML-based documentation for public classes and public/protected members of classes. You will learn about public classes and public/protected members of classes in Chapter 2.

For this reason, `FourOfAKind`'s documentation reveals only the public `main()` method. It does not reveal `isFourOfAKind()` and the other package-private methods. If you want to include these methods in the documentation, you must specify `-package` with `javadoc`:

```
javadoc -package *.java
```

> **NOTE:** The standard class library's documentation was also generated by `javadoc` and adheres to the same format.

Compiling, Running, and Distributing FourOfAKind

Unlike the previous `DumpArgs` and `EchoText` applications, which each consist of one source file, `FourOfAKind` consists of `Card.java`, `Deck.java`, `DiscardPile.java`, and `FourOfAKind.java`. You can compile all four source files via the following command line:

```
javac FourOfAKind.java
```

The `javac` tool launches the Java compiler, which recursively compiles the source files of the various classes it encounters during compilation. Assuming successful compilation, you should end up with six classfiles in the current directory.

> **TIP:** You can compile all Java source files in the current directory by specifying `javac *.java`.

After successfully compiling `FourOfAKind.java` and the other three source files, specify the following command line to run this application:

```
java FourOfAKind
```

In response, you see an introductory message and the four cards that you are holding. The following output reveals a single session:

```
Welcome to Four of a Kind!

Held cards:
A. SIX_OF_CLUBS
B. QUEEN_OF_DIAMONDS
C. SIX_OF_HEARTS
D. SIX_OF_SPADES

Which card do you want to throw away (A, B, C, D)? B

Held cards:
A. SIX_OF_CLUBS
B. NINE_OF_HEARTS
C. SIX_OF_HEARTS
D. SIX_OF_SPADES

Which card do you want to throw away (A, B, C, D)? B

Held cards:
A. SIX_OF_CLUBS
B. FOUR_OF_DIAMONDS
C. SIX_OF_HEARTS
D. SIX_OF_SPADES

Which card do you want to throw away (A, B, C, D)? B

Held cards:
A. SIX_OF_CLUBS
B. KING_OF_HEARTS
C. SIX_OF_HEARTS
D. SIX_OF_SPADES
```

```
Which card do you want to throw away (A, B, C, D)? B

Held cards:
A. SIX_OF_CLUBS
B. QUEEN_OF_CLUBS
C. SIX_OF_HEARTS
D. SIX_OF_SPADES

Which card do you want to throw away (A, B, C, D)? B

Held cards:
A. SIX_OF_CLUBS
B. KING_OF_DIAMONDS
C. SIX_OF_HEARTS
D. SIX_OF_SPADES

Which card do you want to throw away (A, B, C, D)? B

Held cards:
A. SIX_OF_CLUBS
B. TWO_OF_HEARTS
C. SIX_OF_HEARTS
D. SIX_OF_SPADES

Which card do you want to throw away (A, B, C, D)? B

Held cards:
A. SIX_OF_CLUBS
B. FIVE_OF_DIAMONDS
C. SIX_OF_HEARTS
D. SIX_OF_SPADES

Which card do you want to throw away (A, B, C, D)? B

Held cards:
A. SIX_OF_CLUBS
B. JACK_OF_CLUBS
C. SIX_OF_HEARTS
D. SIX_OF_SPADES

Which card do you want to throw away (A, B, C, D)? B

Held cards:
A. SIX_OF_CLUBS
B. TWO_OF_SPADES
C. SIX_OF_HEARTS
D. SIX_OF_SPADES

Which card do you want to throw away (A, B, C, D)? B

Human wins!

Human's cards:
SIX_OF_CLUBS
SIX_OF_DIAMONDS
SIX_OF_HEARTS
```

```
SIX_OF_SPADES

Computer's cards:
SEVEN_OF_HEARTS
TEN_OF_HEARTS
SEVEN_OF_CLUBS
SEVEN_OF_DIAMONDS
```

Although *Four of a Kind* is not much of a card game, you might decide to share the `FourOfAKind` application with a friend. However, if you forget to include even one of the application's five supporting classfiles, your friend will not be able to run the application.

You can overcome this problem by bundling `FourOfAKind`'s six classfiles into a single *JAR (Java ARchive) file*, which is a ZIP file that contains a special directory and the `.jar` file extension. You can then distribute this single JAR file to your friend.

The JDK provides the `jar` tool for working with JAR files. To bundle all six classfiles into a JAR file named `FourOfAKind.jar`, you could specify the following command line, where `c` tells `jar` to create a JAR file and `f` identifies the JAR file's name:

```
jar cf FourOfAKind.jar *.class
```

After creating the JAR file, try to run the application via the following command line:

```
java -jar FourOfAKind.jar
```

Instead of the application running, you will receive an error message having to do with `java` not knowing which of the JAR file's six classfiles is the *main classfile* (the file whose class's `main()` method executes first).

You can provide this knowledge via a text file that is merged into the JAR file's *manifest*, a special file named `MANIFEST.MF` that stores information about the contents of a JAR file, and which is stored in the JAR file's `META-INF` directory. Consider Listing 1–9.

Listing 1–9. *Identifying the application's main class*

```
Main-Class: FourOfAKind
```

Listing 1–9 tells `java` which of the JAR's classfiles is the main class file. (You must leave a blank line after `Main-Class: FourOfAKind`.)

The following command line, which creates `FourOfAKind.jar`, includes `m` and the name of the text file providing manifest content:

```
jar cfm FourOfAKind.jar manifest *.class
```

This time, `java -jar FourOfAKind.jar` succeeds and the application runs because `java` is able to identify `FourOfAKind` as the main classfile.

EXERCISES

The following exercises are designed to test your understanding of what Java means, the JDK, NetBeans, Eclipse, and Java application development:

1. What is Java?
2. What is a virtual machine?
3. What is the purpose of the Java compiler?
4. True or false: A classfile's instructions are commonly referred to as bytecode.
5. What does the virtual machine's interpreter do when it learns that a sequence of bytecode instructions is being executed repeatedly?
6. How does the Java platform promote portability?
7. How does the Java platform promote security?
8. True or false: Java SE is the Java platform for developing servlets.
9. What is the JRE?
10. What is the difference between the public and private JREs?
11. What is the JDK?
12. Which JDK tool is used to compile Java source code?
13. Which JDK tool is used to run Java applications?
14. What is the purpose of the JDK's `jar` tool?
15. What is Standard I/O?
16. What is an IDE?
17. Identify two popular IDEs.
18. What is pseudocode?
19. How would you list `FourOfAKind.jar`'s *table of contents* (list of directories and files contained in the JAR file)?
20. Modify `FourOfAKind` to give each player the option of picking up the top card from the deck or discard pile (assuming that the discard pile is not empty; it is empty when the human player goes first and starts his/her first turn). Display the discard pile's top card for the human player's benefit. (Do not display the deck's top card.)

Summary

Java is a language and a platform. The language is partly patterned after the C and C++ languages to shorten the learning curve for C/C++ developers. The platform consists of a virtual machine and associated execution environment.

Developers use different editions of the Java platform to create Java programs that run on desktop computers, web browsers, web servers, mobile information devices, and embedded devices. These editions are known as Java SE, Java EE, and Java ME.

Developers also use a special Google-created edition of the Java platform to create Android apps that run on Android-enabled devices. This edition, known as the Android platform, largely consists of Java core libraries and a virtual machine known as Dalvik.

The public JRE implements the Java SE platform and makes it possible to run Java programs. The JDK provides tools (including the Java compiler) for developing Java programs and also includes a private copy of the JRE.

Working with the JDK's tools at the command line is not recommended for large projects, which are hard to manage without the help of an integrated development environment. Two popular IDEs are NetBeans and Eclipse.

Application development is not an easy task. All applications except for the most trivial require careful planning or you will probably waste your (and your users') time and money. One way to develop applications efficiently involves using pseudocode.

`FourOfAKind` gave you a significant taste of the Java language. Although much of its source code is probably hard to understand right now, you will find it much easier to grasp after reading Chapter 2, which introduces you to Java's language fundamentals.