Neque porro quisquam est qui dolorem ipsum
quia dolor sit amet, consectetur, adipisci velit

Learn Java for Android Development

Jeff Friesen

Apress®

Learn Java for Android Development

Jeff “JavaJeff” Friesen

Apress®

Learn Java for Android Development

ISBN-13 (pbk): 978-1-4302-3156-1

ISBN-13 (electronic): 978-1-4302-3157-8

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

President and Publisher: Paul Manning
Lead Editor: Steve Anglin
Development Editor: Tom Welsh
Technical Reviewer: Paul Connolly
Editorial Board: Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Gary Cornell, Jonathan Gennick, Jonathan Hassell, Michelle Lowman, Matthew Moodie, Duncan Parkes, Jeffrey Pepper, Frank Pohlmann, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Coordinating Editor: Debra Kelly
Copy Editor: Bill McManus
Compositor: MacPS, LLC
Indexer: John Collin
Artist: April Milne
Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer Science+Business Media, LLC., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/info/bulksales.

The source code for this book is available to readers at www.apress.com/book/view/1430231564.

To my best friend

Contents at a Glance

Contents

About the Author

Jeff "JavaJeff" Friesen has been actively involved with Java since the late 1990s. Jeff has worked with Java in various companies, including a healthcare-oriented consulting firm, where he created his own Java/C++ software for working with smart cards. Jeff has written about Java in numerous articles for JavaWorld (`www.javaworld.com`), informIT (`www.informit.com`), and java.net (`http://java.net`), and has authored several books on Java, including *Beginning Java SE 6 Platform: From Novice to Professional* (Apress, 2007; ISBN: 159059830X), which focuses exclusively on Java version 6's new and improved features. Jeff has also taught Java in university and college continuing education classes. He has a Bachelor of Science degree in mathematics and computer science from Brandon University in Brandon, Manitoba, Canada, and currently freelances in Java and other software technologies.

About the Technical Reviewer

Paul Connolly is the Director of Engineering for Atypon Systems' RightSuite product line. RightSuite is an enterprise access-control and commerce solution used by many of the world's largest publishing and media companies. Paul enjoys designing and implementing high-performance, enterprise-class software systems. He is also an active contributor in the open-source community.

Prior to joining Atypon Systems, Paul worked as a senior software engineer at Standard & Poor's where he architected and developed key communications systems. Paul is a Sun Certified Java Programmer, Sun Certified Business Component Developer, and a Sun Certified Web Component Developer. Paul lives in New York City with his wife, Marina.

Acknowledgments

I thank Steve Anglin for contacting me to write this book, Debra Kelly for guiding me through the various aspects of this project, Tom Welsh for helping me with the development of my chapters, Paul Connolly for his diligence in catching various flaws that would otherwise have made it into this book, and Bill McManus and the production team for making the book's content look good.

It has been many years since I started writing about Java, and I also thank the following editors who have helped me share my knowledge with others: Chris Adamson, Bridget Collins, Richard Dal Porto, Sean Dixon, Victoria Elzey, Kevin Farnham, Todd Green, Jennifer Orr, Athen O'Shea, Esther Schindler, Daniel Steinberg, Jill Steinberg, Dustin Sullivan, and Atlanta Wilson.

Introduction

Smartphones and other touch-based mobile devices are all the rage these days. Their popularity is largely due to their ability to run *apps*. Although the iPhone and iPad with their growing collection of Objective-C-based apps are the leaders of the pack, Android-based smartphones with their growing collection of Java-based apps are proving to be a strong competitor.

Not only are many iPhone/iPad developers making money by selling their apps, many Android developers are also making money by selling similar apps. According to tech websites such as *The Register* (`www.theregister.co.uk/`), some Android developers are making lots of money (`www.theregister.co.uk/2010/03/02/android_app_profit/`).

In today's tough economic climate, perhaps you would like to try your hand at becoming an Android developer and make some money. If you have good ideas, perseverance, and some artistic talent (or perhaps know some talented individuals), you are already part of the way toward achieving this goal.

Tip: A good reason to consider Android app development over iPhone/iPad app development is the lower startup costs that you will incur with Android. For example, you do not need to purchase a Mac on which to develop Android apps (a Mac is required for developing iPhone/iPad apps); your existing Windows, Linux, or Unix machine will do nicely.

Most importantly, you will need to possess a solid understanding of the Java language and foundational application programming interfaces (APIs) before jumping into Android. After all, Android apps are written in Java and interact with many of the standard Java APIs (such as threading and input/output APIs).

I wrote *Learn Java for Android Development* to give you a solid Java foundation that you can later extend with knowledge of Android architecture, API, and tool specifics. This book will give you a strong grasp of the Java language and many important APIs that are fundamental to Android apps and other Java applications. It will also introduce you to key development tools.

Learn Java for Android Development is organized into ten chapters and one appendix. Each chapter focuses on a collection of related topics and presents a set of exercises that you should complete to get the most benefit from the chapter's content. The appendix provides the solutions to each chapter's exercises.

Note: You can download this book's source code by pointing your web browser to `www.apress.com/book/view/1430231564` and clicking the Source Code link under Book Resources. Although most of this code is compilable with Java version 6, you will need Java version 7 to compile one of the applications.

Chapter 1 introduces you to Java by first focusing on Java's dual nature (language and platform). It then briefly introduces you to Sun's/Oracle's Java SE, Java EE, and Java ME editions of the Java development software, as well as Google's Android edition. You next learn how to download and install the Java SE Development Kit (JDK), and learn some Java basics by developing and playing with a pair of simple Java applications. After receiving a brief introduction to the NetBeans and Eclipse IDEs, you learn about application development in the context of *Four of a Kind*, a console-based card game.

Chapter 2 starts you on an in-depth journey of the Java language by focusing on language fundamentals (such as types, expressions, variables, and statements) in the contexts of classes and objects. Because applications are largely based on classes, it is important to learn how to architect classes correctly, and this chapter shows you how to do so.

Chapter 3 adds to Chapter 2's pool of object-based knowledge by introducing you to those language features that take you from object-based applications to object-oriented applications. Specifically, you learn about features related to inheritance, polymorphism, and interfaces. While exploring inheritance, you learn about Java's ultimate superclass. Also, while exploring interfaces, you discover the real reason for their inclusion in the Java language; interfaces are not merely a workaround for Java's lack of support for multiple implementation inheritance, but serve a higher purpose.

Chapter 4 introduces you to four categories of advanced language features: nested types, packages, static imports, and exceptions. While discussing nested types, I briefly introduce the concept of a closure and state that closures will appear in Java version 7 (which many expect to arrive later this year).

Note: I wrote this book several months before Java version 7's expected arrival in the fall of 2010. Although I have tried to present an accurate portrayal of version 7–specific language features, it is possible that feature syntax may differ somewhat from what is presented in this book. Also, I only discuss closures briefly because this feature was still in a state of flux while writing this book. For more information about closures and other functional programming concepts (such as lambdas) being considered for Java version 7, I recommend that you check out articles such as "Functional Programming Concepts in JDK 7" by Alex Collins (`http://java.dzone.com/articles/lambdas-closures-jdk-7`).

Chapter 5 continues to explore advanced language features by focusing on assertions, annotations, generics, and enums. Although the topic of generics has brought confusion to many developers, I believe that my discussion of this topic will clear up much of the murkiness. Among other items, you learn how to interpret type declarations such as `Enum<E extends Enum<E>>`.

Chapter 6 begins a trend that focuses more on APIs than language features. This chapter first introduces you to many of Java's math-oriented types (such as `Math`, `StrictMath`, `BigDecimal`, and `BigInteger`), and also introduces you to Java's `strictfp` reserved word. It then looks at the `Package` class, primitive wrapper classes, and the References API.

Chapter 7 continues to explore Java's basic APIs by focusing on reflection, string management, the `System` class, and threading.

Chapter 8 focuses exclusively on Java's collections framework, which provides you with a solution for organizing objects in lists, sets, queues, and maps.

Chapter 9 continues to explore Java's utility APIs by introducing you to the concurrency utilities, internationalization, preferences, random number generation, and regular expressions.

Chapter 10 is all about input/output (I/O). In this chapter, you explore Java's classic I/O support in terms of its `File` class, `RandomAccessFile` class, various stream classes, and various writer/reader classes. My discussion of stream I/O includes coverage of Java's object serialization and deserialization mechanisms.

Note: This book largely discusses APIs that are common to Java SE and Android. However, it diverges from this practice in Chapter 9, where I use the Swing toolkit to provide a graphical user interface for one of this chapter's internationalization examples. (Android does not support Swing.)

After you complete this book, I recommend that you obtain a copy of *Beginning Android 2* by Mark L Murphy (Apress, 2010; ISBN: 1430226293) and start learning how to develop Android apps. In that book, "you'll learn how to develop applications for Android 2.x mobile devices, using simple examples that are ready to run with your copy of the JDK."

Note: Over the next few months, I will make available at my `javajeff.mb.ca` website six additional PDF-based chapters. These chapters will introduce you to more Java APIs (such as networking and database APIs) that I could not discuss in this book because the book has greatly exceeded its initial 400-page estimate (and the good folks at Apress have been gracious enough to let me do so, but there are limits). I present more information about these PDF files at the end of Chapter 10's "Summary" section.

Thanks for purchasing my book. I hope you find it a helpful preparation for, and I wish you lots of success in achieving, a satisfying and lucrative career as an Android app developer.

Jeff "JavaJeff" Friesen, August 2010

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.

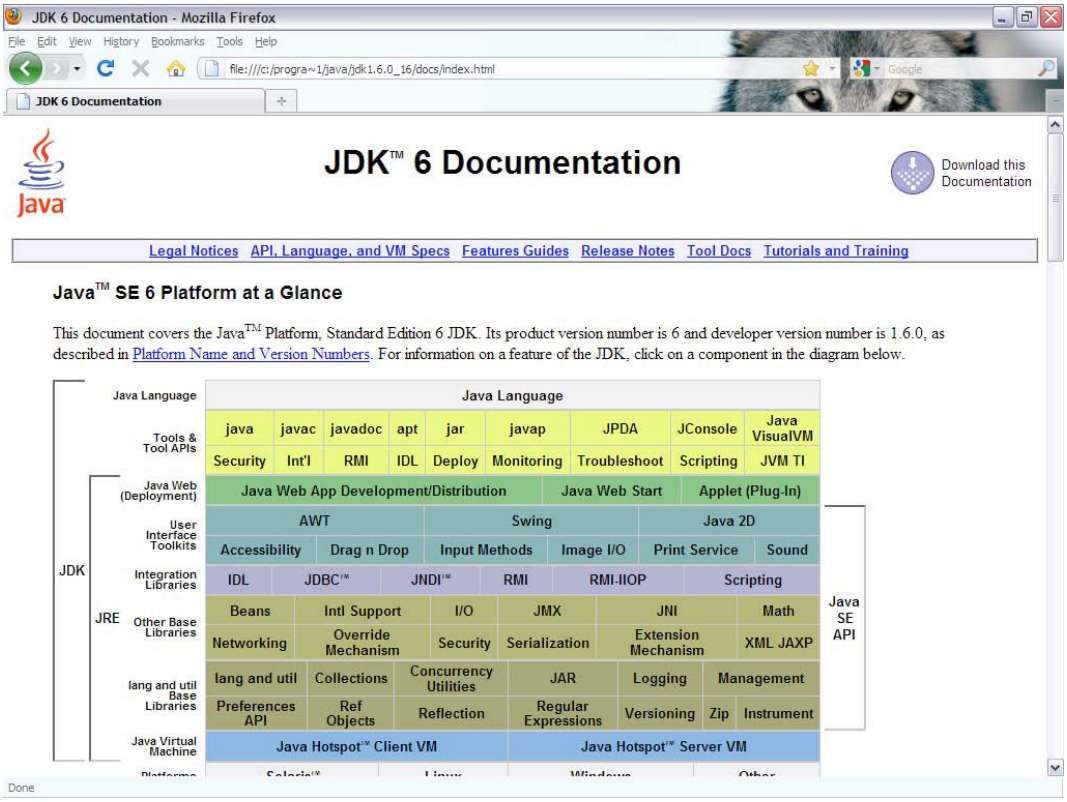

Figure 1–1. *The first part of the Java documentation's start page*

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.

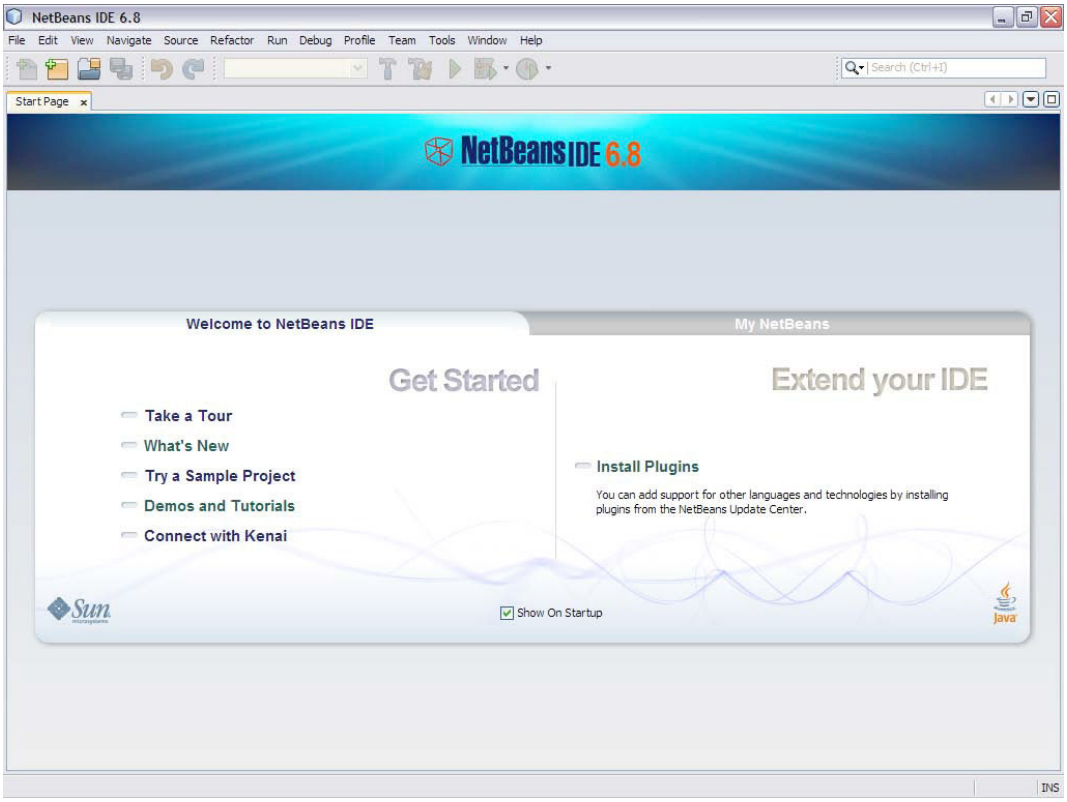

Figure 1–2. *The NetBeans IDE 6.8 main window*

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.