Table of Contents

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Chapter 1. Introduction

Welcome to Java. This chapter begins by explaining what Java is and describing some of the features that distinguish it from other programming languages. Next, it outlines the structure of this book, with special emphasis on what is new in Java 5.0. Finally, as a quick tutorial introduction to the language, it walks you through a simple Java program you can type, compile, and run.

1.1. What Is Java?

In discussing Java, it is important to distinguish between the Java programming language, the Java Virtual Machine, and the Java platform. The Java programming language is the language in which Java applications, applets, servlets, and components are written. When a Java program is compiled, it is converted to byte codes that are the portable machine language of a CPU architecture known as the Java Virtual Machine (also called the Java VM or JVM). The JVM can be implemented directly in hardware, but it is usually implemented in the form of a software program that interprets and executes byte codes.

The Java platform is distinct from both the Java language and Java VM. The Java platform is the predefined set of Java classes that exist on every Java installation; these classes are available for use by all Java programs. The Java platform is also sometimes referred to as the Java runtime environment or the core Java APIs (application programming interfaces). The Java platform can be extended with optional packages (formerly called standard extensions). These APIs exist in some Java installations but are not guaranteed to exist in all installations.

1.1.1. The Java Programming Language

The Java programming language is a state-of-the-art, object-oriented language that has a syntax similar to that of C. The language designers strove to make the Java language powerful, but, at the same time, they tried to avoid the overly complex features that have bogged down other object-oriented languages like C++. By keeping the language simple, the designers also made it easier for programmers to write robust, bug-free code. As a result of its elegant design and next-generation features, the Java language has proved popular with programmers, who typically find it a pleasure to work with Java after struggling with more difficult, less powerful languages.

Java 5.0, the latest version of the Java language,[1] includes a number of new language features, most notably generic types, which increase both the complexity and the power

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

of the language. Most experienced Java programmers have welcomed the new features, despite the added complexity they bring.

[1] Java 5.0 represents a significant change in version numbering for Sun. The previous version of Java is Java 1.4 so you may sometimes hear Java 5.0 informally referred to as Java 1.5.

1.1.2. The Java Virtual Machine

The Java Virtual Machine, or Java interpreter, is the crucial piece of every Java installation. By design, Java programs are portable, but they are only portable to platforms to which a Java interpreter has been ported. Sun ships VM implementations for its own Solaris operating system and for Microsoft Windows and Linux platforms. Many other vendors, including Apple and various commercial Unix vendors, provide Java interpreters for their platforms. The Java VM is not only for desktop systems, however. It has been ported to set-top boxes and handheld devices that run Windows CE and PalmOS.

Although interpreters are not typically considered high-performance systems, Java VM performance has improved dramatically since the first versions of the language. The latest releases of Java run remarkably fast. Of particular note is a VM technology called *just-in-time* (JIT) compilation whereby Java byte codes are converted on the fly into native platform machine language, boosting execution speed for code that is run repeatedly.

1.1.3. The Java Platform

The Java platform is just as important as the Java programming language and the Java Virtual Machine. All programs written in the Java language rely on the set of predefined classes[2] that comprise the Java platform. Java classes are organized into related groups known as *packages*. The Java platform defines packages for functionality such as input/output, networking, graphics, user-interface creation, security, and much more.

[2] A *class* is a module of Java code that defines a data structure and a set of methods (also called procedures, functions, or subroutines) that operate on that data.

It is important to understand what is meant by the term platform. To a computer programmer, a platform is defined by the APIs he can rely on when writing programs. These APIs are usually defined by the operating system of the target computer. Thus, a programmer writing a program to run under Microsoft Windows must use a different set of APIs than a programmer writing the same program for a Unix-based system. In this respect, Windows and Unix are distinct platforms.

Java is not an operating system. Nevertheless, the Java platform provides APIs with a comparable breadth and depth to those defined by an operating system. With the Java platform, you can write applications in Java without sacrificing the advanced features available to programmers writing native applications targeted at a particular underlying operating system. An application written on the Java platform runs on any operating system that supports the Java platform. This means you do not have to create distinct

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Windows, Macintosh, and Unix versions of your programs, for example. A single Java program runs on all these operating systems, which explains why "Write once, run anywhere" is Sun's motto for Java.

The Java platform is not an operating system, but for programmers, it is an alternative development target and a very popular one at that. The Java platform reduces programmers' reliance on the underlying operating system, and, by allowing programs to run on top of any operating system, it increases end users' freedom to choose an operating system.

1.1.4. Versions of Java

As of this writing, there have been six major versions of Java. They are:

Java 1.0

This was the first public version of Java. It contained 212 classes organized in 8 packages. It was simple and elegant but is now completely outdated.

Java 1.1

This release of Java more than doubled the size of the Java platform to 504 classes in 23 packages. It introduced nested types (or "inner classes"), an important change to the Java language itself, and included significant performance improvements in the Java VM. This version is outdated.

Java 1.2

This was a very significant release of Java; it tripled the size of the Java platform to 1,520 classes in 59 packages. Important additions included the Collections API for working with sets, lists, and maps of objects and the Swing API for creating graphical user interfaces. Because of the many new features included in the 1.2 release, the platform was rebranded as "the Java 2 Platform." The term "Java 2" was simply a trademark, however, and not an actual version number for the release.

Java 1.3

This was primarily a maintenance release, focused on bug fixes, stability, and performance improvements (including the high-performance "HotSpot" virtual machine). Additions to the platform included the Java Naming and Directory Interface (JNDI) and the Java Sound APIs, which were previously available as extensions to the platform. The most interesting classes in this release were probably

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

`java.util.Timer` and `java.lang.reflect.Proxy`. In total, Java 1.3 contains 1,842 classes in 76 packages.

Java 1.4

This was another big release, adding important new functionality and increasing the size of the platform by 62% to 2,991 classes and interfaces in 135 packages. New features included a high-performance, low-level I/O API; support for pattern matching with regular expressions; a logging API; a user preferences API; new Collections classes; an XML-based persistence mechanism for JavaBeans; support for XML parsing using both the DOM and SAX APIs; user authentication with the Java Authentication and Authorization Service (JAAS) API; support for secure network connections using the SSL protocol; support for cryptography; a new API for reading and writing image files; an API for network printing; a handful of new GUI components in the Swing API; and a simplified drag-and-drop architecture for Swing. In addition to these platform changes, the 1.4 release introduced an `assert` statement to the Java language.

Java 5.0

The most recent release of Java introduces a number of changes to the core language itself including generic types, enumerated types, annotations, varargs methods, autoboxing, and a new `for/in` statement. Because of the major language changes, the version number was incremented. This release would logically be known as "Java 2.0" if Sun had not already used the term "Java 2" for marketing Java 1.2.

In addition to the language changes, Java 5.0 includes a number of additions to the Java platform as well. This release includes 3562 classes and interfaces in 166 packages. Notable additions include utilities for concurrent programming, a remote management framework, and classes for the remote management and instrumentation of the Java VM itself.

See the Preface for a list of changes in this edition of the book, including pointers to coverage of the new language and platform features.

To write programs in Java, you must obtain the Java Development Kit (JDK). Sun releases a new version of the JDK for each new version of Java. Don't confuse the JDK with the Java Runtime Environment (JRE). The JRE contains everything you need to run Java programs, but it does not contain the tools you need to develop Java programs (primarily the compiler).

In addition to the Standard Edition of Java used by most Java developers and documented in this book, Sun has also released the Java 2 Platform, Enterprise Edition (or J2EE) for

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

enterprise developers and the Java 2 Platform, Micro Edition (J2ME) for consumer electronic systems, such as handheld PDAs and cellular telephones. See *Java Enterprise in a Nutshell* and *Java Micro Edition in a Nutshell* (both by O'Reilly) for more information on these other editions.

1.2. Key Benefits of Java

Why use Java at all? Is it worth learning a new language and a new platform? This section explores some of the key benefits of Java.

1.2.1. Write Once, Run Anywhere

Sun identifies "Write once, run anywhere" as the core value proposition of the Java platform. Translated from business jargon, this means that the most important promise of Java technology is that you have to write your application only once—for the Java platform—and then you'll be able to run it *anywhere*.

Anywhere, that is, that supports the Java platform. Fortunately, Java support is becoming ubiquitous. It is integrated into practically all major operating systems. It is built into the popular web browsers, which places it on virtually every Internet-connected PC in the world. It is even being built into consumer electronic devices such as television set-top boxes, PDAs, and cell phones.

1.2.2. Security

Another key benefit of Java is its security features. Both the language and the platform were designed from the ground up with security in mind. The Java platform allows users to download untrusted code over a network and run it in a secure environment in which it cannot do any harm: untrusted code cannot infect the host system with a virus, cannot read or write files from the hard drive, and so forth. This capability alone makes the Java platform unique.

Java 1.2 took the security model a step further. It made security levels and restrictions highly configurable and extended them beyond applets. As of Java 1.2, any Java code, whether it is an applet, a servlet, a JavaBeans component, or a complete Java application, can be run with restricted permissions that prevent it from doing harm to the host system.

The security features of the Java language and platform have been subjected to intense scrutiny by security experts around the world. In the earlier days of Java, security-related bugs, some of them potentially serious, were found and promptly fixed. Because of the strong security promises Java makes, it is big news when a new security bug is found. No other mainstream platform can make security guarantees nearly as strong as those Java makes. No one can say that Java security holes will not be found in the future, but if Java's

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

security is not yet perfect, it has been proven strong enough for practical day-to-day use and is certainly better than any of the alternatives.

1.2.3. Network-Centric Programming

Sun's corporate motto has always been "The network is the computer." The designers of the Java platform believed in the importance of networking and designed the Java platform to be network-centric. From a programmer's point of view, Java makes it easy to work with resources across a network and to create network-based applications using client/server or multitier architectures.

1.2.4. Dynamic, Extensible Programs

Java is both dynamic and extensible. Java code is organized in modular object-oriented units called *classes*. Classes are stored in separate files and are loaded into the Java interpreter only when needed. This means that an application can decide as it is running what classes it needs and can load them when it needs them. It also means that a program can dynamically extend itself by loading the classes it needs to expand its functionality.

The network-centric design of the Java platform means that a Java application can dynamically extend itself by loading new classes over a network. An application that takes advantage of these features ceases to be a monolithic block of code. Instead, it becomes an interacting collection of independent software components. Thus, Java enables a powerful new metaphor of application design and development.

1.2.5. Internationalization

The Java language and the Java platform were designed from the start with the rest of the world in mind. When it was created, Java was the only commonly used programming language that had internationalization features at its core rather than tacked on as an afterthought. While most programming languages use 8-bit characters that represent only the alphabets of English and Western European languages, Java uses 16-bit Unicode characters that represent the phonetic alphabets and ideographic character sets of the entire world. Java's internationalization features are not restricted to just low-level character representation, however. The features permeate the Java platform, making it easier to write internationalized programs with Java than it is with any other environment.

1.2.6. Performance

As described earlier, Java programs are compiled to a portable intermediate form known as byte codes, rather than to native machine-language instructions. The Java Virtual Machine runs a Java program by interpreting these portable byte-code instructions. This architecture means that Java programs are faster than programs or scripts written in purely interpreted languages, but Java programs are typically slower than C and C++

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

programs compiled to native machine language. Keep in mind, however, that although Java programs are compiled to byte code, not all of the Java platform is implemented with interpreted byte codes. For efficiency, computationally intensive portions of the Java platform—such as the string-manipulation methods—are implemented using native machine code.

Although early releases of Java suffered from performance problems, the speed of the Java VM has improved dramatically with each new release. The VM has been highly tuned and optimized in many significant ways. Furthermore, most current implementations include a just-in-time (JIT) compiler, which converts Java byte codes to native machine instructions on the fly. Using sophisticated JIT compilers, Java programs can execute at speeds comparable to the speeds of native C and C++ applications.

Java is a portable, interpreted language; Java programs run almost as fast as native, nonportable C and C++ programs. Performance used to be an issue that made some programmers avoid using Java. With the improvements made in Java 1.2, 1.3, 1.4, and 5.0, performance issues should no longer keep anyone away.

1.2.7. Programmer Efficiency and Time-to-Market

The final, and perhaps most important, reason to use Java is that programmers like it. Java is an elegant language combined with a powerful and (usually) well-designed set of APIs. Programmers enjoy programming in Java and are often amazed at how quickly they can get results with it. Because Java is a simple and elegant language with a well-designed, intuitive set of APIs, programmers write better code with fewer bugs than for other platforms, thus reducing development time.

1.3. An Example Program

Example 1-1 shows a Java program to compute factorials.[3] Note that the numbers at the beginning of each line are not part of the program; they are there for ease of reference when we dissect the program line-by-line.

[3] The factorial of an integer is the product of the number and all positive integers less than the number. So, for example, the factorial of 4, which is also written 4!, is 4 times 3 times 2 times 1, or 24. By definition, 0! is 1.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Example 1-1. Factorial.java: a program to compute factorials

```
 1 /**
 2  * This program computes the factorial of a number
 3  */
 4 public class Factorial {                    // Define a class
 5   public static void main(String[] args) { // The program starts here
 6     int input = Integer.parseInt(args[0]); // Get the user's input
 7     double result = factorial(input);      // Compute the factorial
 8     System.out.println(result);            // Print out the result
 9   }                                        // The main() method ends here
10
11   public static double factorial(int x) {  // This method computes x!
12     if (x < 0)                             // Check for bad input
13       return 0.0;                          // If bad, return 0
14     double fact = 1.0;                     // Begin with an initial value
15     while(x > 1) {                         // Loop until x equals 1
16       fact = fact * x;                     // Multiply by x each time
17       x = x - 1;                           // And then decrement x
18     }                                      // Jump back to start of loop
19     return fact;                           // Return the result
20   }                                        // factorial() ends here
21 }                                          // The class ends here
```

1.3.1. Compiling and Running the Program

Before we look at how the program works, we must first discuss how to run it. In order to compile and run the program, you need a Java development kit (JDK) of some sort. Sun Microsystems created the Java language and ships a free JDK for its Solaris operating system and also for Linux and Microsoft Windows platforms.[4] At the time of this writing, the current version of Sun's JDK is available for download from http://java.sun.com. Be sure to get the JDK and not the Java Runtime Environment. The JRE enables you to run existing Java programs, but not to write and compile your own.

[4] Other companies, such as Apple, have licensed and ported the JDK to their operating systems. In Apple's case, this arrangement leads to a delay in the latest JDK being available on that platform.

The Sun JDK is not the only Java programming environment you can use. *gcj*, for example, is a Java compiler released under the GNU general public license. A number of companies sell Java IDEs (integrated development environments), and high-quality open-source IDEs are also available. This book assumes that you are using Sun's JDK and its accompanying command-line tools. If you are using a product from some other vendor, be sure to read that vendor's documentation to learn how to compile and run a simple program, like that shown in Example 1-1.

Once you have a Java programming environment installed, the first step towards running our program is to type it in. Using your favorite text editor, enter the program as it is shown in Example 1-1.[5] Omit the line numbers, which are just for reference. Note that Java is a case-sensitive language, so you must type lowercase letters in lowercase and uppercase letters in uppercase. You'll notice that many of the lines of this program end with semicolons. It is a common mistake to forget these characters, but the program won't work

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

without them, so be careful! You can omit everything from // to the end of a line: those are *comments* that are there for your benefit and are ignored by Java.

[5] I recommend that you type this example in by hand, to get a feel for the language. If you *really* don't want to, however, you can download this, and all examples in the book, from http://www.oreilly.com/catalog/javanut5/.

When writing Java programs, you should use a text editor that saves files in plain-text format, not a word processor that supports fonts and formatting and saves files in a proprietary format. My favorite text editor on Unix systems is *Emacs*. If you use a Windows system, you might use *Notepad* or *WordPad*, if you don't have a more specialized programmer's editor (versions of GNU Emacs, for example, are available for Windows). If you are using an IDE, it should include an appropriate text editor; read the documentation that came with the product. When you are done entering the program, save it in a file named *Factorial.java*. This is important; the program will not work if you save it by any other name.

After writing a program like this one, the next step is to compile it. With Sun's JDK, the Java compiler is known as *javac*. *javac* is a command-line tool, so you can only use it from a terminal window, such as an MS-DOS window on a Windows system or an *xterm* window on a Unix system. Compile the program by typing the following command:

```
C:\> javac Factorial.java
```

If this command prints any error messages, you probably got something wrong when you typed in the program. If it does not print any error messages, however, the compilation has succeeded, and *javac* creates a file called *Factorial.class*. This is the compiled version of the program.

Once you have compiled a Java program, you must still run it. Java programs are not compiled into native machine language, so they cannot be executed directly by the system. Instead, they are run by another program known as the Java interpreter. In Sun's JDK, the interpreter is a command-line program named, appropriately enough, *java*. To run the factorial program, type:

```
C:\> java Factorial 4
```

java is the command to run the Java interpreter, *Factorial* is the name of the Java program we want the interpreter to run, and *4* is the input data—the number we want the interpreter to compute the factorial of. The program prints a single line of output, telling us that the factorial of 4 is 24:

```
C:\> java Factorial 4
24.0
```

Congratulations! You've just written, compiled, and run your first Java program. Try running it again to compute the factorials of some other numbers.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

1.3.2. Analyzing the Program

Now that you have run the factorial program, let's analyze it line by line to see what makes a Java program tick.

1.3.2.1. Comments

The first three lines of the program are a comment. Java ignores them, but they tell a human programmer what the program does. A comment begins with the characters `/*` and ends with the characters `*/`. Any amount of text, including multiple lines of text, may appear between these characters. Java also supports another type of comment, which you can see in lines 4 through 21. If the characters `//` appear in a Java program, Java ignores those characters and any other text that appears between those characters and the end of the line.

1.3.2.2. Defining a class

Line 4 is the first line of Java code. It says that we are defining a class named `Factorial`. This explains why the program had to be stored in a file named *Factorial.java*. That filename indicates that the file contains Java source code for a class named `Factorial`. The word `public` is a *modifier*; it says that the class is publicly available and that anyone may use it. The open curly-brace character (`{`) marks the beginning of the body of the class, which extends all the way to line 21, where we find the matching close curly-brace character (`}`). The program contains a number of pairs of curly braces; the lines are indented to show the nesting within these braces.

A class is the fundamental unit of program structure in Java, so it is not surprising that the first line of our program declares a class. All Java programs are classes, although some programs use many classes instead of just one. Java is an object-oriented programming language, and classes are a fundamental part of the object-oriented paradigm. Each class defines a unique kind of object. Example 1-1 is not really an object-oriented program, however, so I'm not going to go into detail about classes and objects here. That is the topic of Chapter 3. For now, all you need to understand is that a class defines a set of interacting *members*. Those members may be fields, methods, or other classes. The `Factorial` class contains two members, both of which are methods. They are described in upcoming sections.

1.3.2.3. Defining a method

Line 5 begins the definition of a *method* of our `Factorial` class. A method is a named chunk of Java code. A Java program can call, or *invoke*, a method to execute the code in it. If you have programmed in other languages, you have probably seen methods before, but they may have been called functions, procedures, or subroutines. The interesting thing about methods is that they have *parameters* and *return values*. When you call a method,

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

you pass it some data you want it to operate on, and it returns a result to you. A method is like an algebraic function:

```
y = f(x)
```

Here, the mathematical function `f` performs some computation on the value represented by `x` and returns a value, which we represent by `y`.

To return to line 5, the `public` and `static` keywords are modifiers. `public` means the method is publicly accessible; anyone can use it. The meaning of the `static` modifier is not important here; it is explained in Chapter 3. The `void` keyword specifies the return value of the method. In this case, it specifies that this method does not have a return value.

The word `main` is the name of the method. `main` is a special name.[6] When you run the Java interpreter, it reads in the class you specify, then looks for a method named `main()`.[7] When the interpreter finds this method, it starts running the program at that method. When the `main()` method finishes, the program is done, and the Java interpreter exits. In other words, the `main()` method is the main entry point into a Java program. It is not actually sufficient for a method to be named `main()`, however. The method must be declared `public static void` exactly as shown in line 5. In fact, the only part of line 5 you can change is the word `args`, which you can replace with any word you want. You'll be using this line in all of your Java programs, so go ahead and commit it to memory now!

[6] All Java programs that are run directly by the Java interpreter must have a `main()` method. Programs of this sort are often called *applications*. It is possible to write programs that are not run directly by the interpreter, but are dynamically loaded into some other already running Java program. Examples are *applets*, which are programs run by a web browser, and *servlets*, which are programs run by a web server. Applets are discussed in *Java Foundation Classes in a Nutshell* (O'Reilly) while servlets are discussed in *Java Enterprise in a Nutshell* (O'Reilly). In this book, we consider only applications.

[7] By convention, when this book refers to a method, it follows the name of the method by a pair of parentheses. As you'll see, parentheses are an important part of method syntax, and they serve here to keep method names distinct from the names of classes, fields, variables, and so on.

Following the name of the `main()` method is a list of method parameters in parentheses. This `main()` method has only a single parameter. `String[]` specifies the type of the parameter, which is an array of strings (i.e., a numbered list of strings of text). `args` specifies the name of the parameter. In the algebraic equation `f(x)`, `x` is simply a way of referring to an unknown value. `args` serves the same purpose for the `main()` method. As we'll see, the name `args` is used in the body of the method to refer to the unknown value that is passed to the method.

As I've just explained, the `main()` method is a special one that is called by the Java interpreter when it starts running a Java class (program). When you invoke the Java interpreter like this:

```
C:\> java Factorial 4
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

the string "4" is passed to the `main( )` method as the value of the parameter named `args`. More precisely, an array of strings containing only one entry, `4`, is passed to `main()`. If we invoke the program like this:

```
C:\> java Factorial 4 3 2 1
```

then an array of four strings, `4`, `3`, `2`, and `1`, is passed to the `main( )` method as the value of the parameter named `args`. Our program looks only at the first string in the array, so the other strings are ignored.

Finally, the last thing on line 5 is an open curly brace. This marks the beginning of the body of the `main()` method, which continues until the matching close curly brace on line 9. Methods are composed of *statements*, which the Java interpreter executes in sequential order. In this case, lines 6, 7, and 8 are three statements that compose the body of the `main()` method. Each statement ends with a semicolon to separate it from the next. This is an important part of Java syntax; beginning programmers often forget the semicolons.

1.3.2.4. Declaring a variable and parsing input

The first statement of the `main()` method, line 6, declares a variable and assigns a value to it. In any programming language, a *variable* is simply a symbolic name for a value. We've already seen that, in this program, the name `args` refers to the parameter value passed to the `main()` method. Method parameters are one type of variable. It is also possible for methods to declare additional "local" variables. Methods can use local variables to store and reference the intermediate values they use while performing their computations.

This is exactly what we are doing on line 6. That line begins with the words `int input`, which declare a variable named `input` and specify that the variable has the type `int`; that is, it is an integer. Java can work with several different types of values, including integers, real or floating-point numbers, characters (e.g., letters and digits), and strings of text. Java is a *strongly typed* language, which means that all variables must have a type specified and can refer only to values of that type. Our `input` variable always refers to an integer, so it cannot refer to a floating-point number or a string. Method parameters are also typed. Recall that the `args` parameter had a type of `String[ ]`.

Continuing with line 6, the variable declaration `int input` is followed by the = character. This is the assignment operator in Java; it sets the value of a variable. When reading Java code, don't read = as "equals," but instead read it as "is assigned the value." As we'll see in Chapter 2, there is a different operator for "equals."

The value assigned to our `input` variable is `Integer.parseInt(args[0])`. This is a method invocation. This first statement of the `main( )` method invokes another method whose name is `Integer.parseInt( )`. As you might guess, this method "parses" an

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

integer; that is, it converts a string representation of an integer, such as `4`, to the integer itself. The `Integer.parseInt()` method is not part of the Java language, but it is a core part of the Java API or Application Programming Interface. Every Java program can use the powerful set of classes and methods defined by this core API. The second half of this book is a quick reference that documents that core API.

When you call a method, you pass values (called *arguments*) that are assigned to the corresponding parameters defined by the method, and the method returns a value. The argument passed to `Integer.parseInt()` is `args[0]`. Recall that `args` is the name of the parameter for `main()`; it specifies an array (or list) of strings. The elements of an array are numbered sequentially, and the first one is always numbered `0`. We care about only the first string in the `args` array, so we use the expression `args[0]` to refer to that string. When we invoke the program as shown earlier, line 6 takes the first string specified after the name of the class, `4`, and passes it to the method named `Integer.parseInt()`. This method converts the string to the corresponding integer and returns the integer as its return value. Finally, this returned integer is assigned to the variable named `input`.

1.3.2.5. Computing the result

The statement on line 7 is a lot like the statement on line 6. It declares a variable and assigns a value to it. The value assigned to the variable is computed by invoking a method. The variable is named `result`, and it has a type of `double`. `double` means a double-precision floating-point number. The variable is assigned a value that is computed by the `factorial( )` method. The `factorial()` method, however, is not part of the standard Java API. Instead, it is defined as part of our program by lines 11 through 19. The argument passed to `factorial( )` is the value referred to by the `input` variable that was computed on line 6. We'll consider the body of the `factorial()` method shortly, but you can surmise from its name that this method takes an input value, computes the factorial of that value, and returns the result.

1.3.2.6. Displaying output

Line 8 simply calls a method named `System.out.println( )`. This commonly used method is part of the core Java API; it causes the Java interpreter to print out a value. In this case, the value that it prints is the value referred to by the variable named `result`. This is the result of our factorial computation. If the `input` variable holds the value `4`, the `result` variable holds the value `24`, and this line prints out that value.

The `System.out.println( )` method does not have a return value. There is no variable declaration or = assignment operator in this statement since there is no value to assign to anything. Another way to say this is that, like the `main( )` method of line 5, `System.out.println( )` is declared `void`.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

1.3.2.7. The end of a method

Line 9 contains only a single character, `}`. This marks the end of the method. When the Java interpreter gets here, it is through executing the `main( )` method, so it stops running. The end of the `main( )` method is also the end of the *variable scope* for the `input` and `result` variables declared within `main( )` and for the `args` parameter of `main( )`. These variable and parameter names have meaning only within the `main( )` method and cannot be used elsewhere in the program unless other parts of the program declare different variables or parameters that happen to have the same name.

1.3.2.8. Blank lines

Line 10 is a blank line. You can insert blank lines and spaces anywhere in a program, and you should use them liberally to make the program readable. A blank line appears here to separate the `main( )` method from the `factorial()` method that begins on line 11. You'll notice that the program also uses whitespace to indent the various lines of code. This kind of indentation is optional; it emphasizes the structure of the program and greatly enhances the readability of the code.

1.3.2.9. Another method

Line 11 begins the definition of the `factorial()` method that was used by the `main ( )` method. Compare this line to line 5 to note its similarities and differences. The `factorial( )` method has the same `public` and `static` modifiers. It takes a single integer parameter, which we call `x`. Unlike the `main( )` method, which had no return value (`void`), `factorial( )` returns a value of type `double`. The open curly brace marks the beginning of the method body, which continues past the nested braces on lines 15 and 18 to line 20, where the matching close curly brace is found. The body of the `factorial ( )` method, like the body of the `main()` method, is composed of statements, which are found on lines 12 through 19.

1.3.2.10. Checking for valid input

In the `main()` method, we saw variable declarations, assignments, and method invocations. The statement on line 12 is different. It is an `if` statement, which executes another statement conditionally. We saw earlier that the Java interpreter executes the three statements of the `main()` method one after another. It always executes them in exactly that way, in exactly that order. An `if` statement is a flow-control statement; it can affect the way the interpreter runs a program.

The `if` keyword is followed by a parenthesized expression and a statement. The Java interpreter first evaluates the expression. If it is `true`, the interpreter executes the statement. If the expression is `false`, however, the interpreter skips the statement and goes to the next one. The condition for the `if` statement on line 12 is `x < 0`. It checks

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

whether the value passed to the `factorial()` method is less than zero. If it is, this expression is `true`, and the statement on line 13 is executed. Line 12 does not end with a semicolon because the statement on line 13 is part of the `if` statement. Semicolons are required only at the end of a statement.

Line 13 is a `return` statement. It says that the return value of the `factorial( )` method is `0.0`. `return` is also a flow-control statement. When the Java interpreter sees a `return`, it stops executing the current method and returns the specified value immediately. A `return` statement can stand alone, but in this case, the `return` statement is part of the `if` statement on line 12. The indentation of line 13 helps emphasize this fact. (Java ignores this indentation, but it is very helpful for humans who read Java code!) Line 13 is executed only if the expression on line 12 is `true`.

Before we move on, we should pull back a bit and talk about why lines 12 and 13 are necessary in the first place. It is an error to try to compute a factorial for a negative number, so these lines make sure that the input value `x` is valid. If it is not valid, they cause `factorial( )` to return a consistent invalid result, `0.0`.

1.3.2.11. An important variable

Line 14 is another variable declaration; it declares a variable named `fact` of type `double` and assigns it an initial value of `1.0`. This variable holds the value of the factorial as we compute it in the statements that follow. In Java, variables can be declared anywhere; they are not restricted to the beginning of a method or block of code.

1.3.2.12. Looping and computing the factorial

Line 15 introduces another type of statement: the `while` loop. Like an `if` statement, a `while` statement consists of a parenthesized expression and a statement. When the Java interpreter sees a `while` statement, it evaluates the associated expression. If that expression is `true`, the interpreter executes the statement. The interpreter repeats this process, evaluating the expression and executing the statement if the expression is `true`, until the expression evaluates to `false`. The expression on line 15 is `x > 1`, so the `while` statement loops *while* the parameter `x` holds a value that is greater than 1. Another way to say this is that the loop continues *until* `x` holds a value less than or equal to 1. We can assume from this expression that if the loop is ever going to terminate, the value of `x` must somehow be modified by the statement that the loop executes.

The major difference between the `if` statement on lines 12-13 and the `while` loop on lines 15-18 is that the statement associated with the `while` loop is a *compound statement*. A compound statement is zero or more statements grouped between curly braces. The `while` keyword on line 15 is followed by an expression in parentheses and then by an open

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

curly brace. This means that the body of the loop consists of all statements between that opening brace and the closing brace on line 18. Earlier in the chapter, I said that all Java statements end with semicolons. This rule does not apply to compound statements, however, as you can see by the lack of a semicolon at the end of line 18. The statements inside the compound statement (lines 16 and 17) do end with semicolons, of course.

The body of the `while` loop consists of the statements on line 16 and 17. Line 16 multiplies the value of `fact` by the value of `x` and stores the result back into `fact`. Line 17 is similar. It subtracts 1 from the value of `x` and stores the result back into `x`. The `*` character on line 16 is important: it is the multiplication *operator*. And, as you can probably guess, the `-` on line 17 is the subtraction operator. An operator is a key part of Java syntax: it performs a computation on one or two *operands* to produce a new value. Operands and operators combine to form *expressions*, such as `fact * x` or `x - 1`. We've seen other operators in the program. Line 15, for example, uses the greater-than operator (`>`) in the expression `x > 1`, which compares the value of the variable `x` to 1. The value of this expression is a boolean truth value—either `true` or `false`, depending on the result of the comparison.

To understand this `while` loop, it is helpful to think like the Java interpreter. Suppose we are trying to compute the factorial of 4. Before the loop starts, `fact` is `1.0`, and `x` is `4`. After the body of the loop has been executed once—after the first *iteration*—`fact` is `4.0`, and `x` is `3`. After the second iteration, `fact` is `12.0`, and `x` is `2`. After the third iteration, `fact` is `24.0`, and `x` is `1`. When the interpreter tests the loop condition after the third iteration, it finds that `x > 1` is no longer true, so it stops running the loop, and the program resumes at line 19.

1.3.2.13. Returning the result

Line 19 is another `return` statement, like the one we saw on line 13. This one does not return a constant value like `0.0`, but instead returns the value of the `fact` variable. If the value of `x` passed into the `factorial()` function is `4`, then, as we saw earlier, the value of `fact` is `24.0`, so this is the value returned. Recall that the `factorial()` method was invoked on line 7 of the program. When this `return` statement is executed, control returns to line 7, where the return value is assigned to the variable named `result`.

1.3.3. Exceptions

If you've made it all the way through the line-by-line analysis of Example 1-1, you are well on your way to understanding the basics of the Java language.[8] It is a simple but nontrivial program that illustrates many of the features of Java. There is one more important feature of Java programming I want to introduce, but it is one that does not appear in the program listing itself. Recall that the program computes the factorial of the number you specify on the command line. What happens if you run the program without specifying a number?

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

[8] If you didn't understand all the details of this factorial program, don't worry. We'll cover the details of the Java language a lot more thoroughly in subsequent chapters. However, if you feel like you didn't understand any of the line-by-line analysis, you may also find that the upcoming chapters are over your head. In that case, you should probably go elsewhere to learn the basics of the Java language and return to this book to solidify your understanding, and, of course, to use as a reference. One resource you may find useful in learning the language is Sun's online Java tutorial, available at http://java.sun.com/docs/books/tutorial.

```
C:\> java Factorial
java.lang.ArrayIndexOutOfBoundsException: 0
        at Factorial.main(Factorial.java:6)
C:\>
```

And what happens if you specify a value that is not a number?

```
C:\> java Factorial ten
java.lang.NumberFormatException: ten
        at java.lang.Integer.parseInt(Integer.java)
        at java.lang.Integer.parseInt(Integer.java)
        at Factorial.main(Factorial.java:6)
C:\>
```

In both cases, an error occurs or, in Java terminology, an *exception* is thrown. When an exception is thrown, the Java interpreter prints a message that explains what type of exception it was and where it occurred (both exceptions above occurred on line 6). In the first case, the exception is thrown because there are no strings in the `args` list, meaning we asked for a nonexistent string with `args[0]`. In the second case, the exception is thrown because `Integer.parseInt( )` cannot convert the string "ten" to a number. We'll see more about exceptions in Chapter 2 and learn how to handle them gracefully as they occur.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01