CORE JAVA

Volume I—Fundamentals

ELEVENTH EDITION

CAY S. HORSTMANN

Core Java

Volume I—Fundamentals

Eleventh Edition

This page intentionally left blank

Core Java

Volume I—Fundamentals

Eleventh Edition

Cay S. Horstmann

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Library of Congress Preassigned Control Number: 2018942070

ISBN-13: 978-0-13-516630-7
ISBN-10: 0-13-516630-6

ScoutAutomatedPrintCode

Contents

This page intentionally left blank

Preface

To the Reader

In late 1995, the Java programming language burst onto the Internet scene and gained instant celebrity status. The promise of Java technology was that it would become the *universal glue* that connects users with information wherever it comes from—web servers, databases, information providers, or any other imaginable source. Indeed, Java is in a unique position to fulfill this promise. It is an extremely solidly engineered language that has gained wide acceptance. Its built-in security and safety features are reassuring both to programmers and to the users of Java programs. Java has built-in support for advanced programming tasks, such as network programming, database connectivity, and concurrency.

Since 1995, eleven major revisions of the Java Development Kit have been released. Over the course of the last 20 years, the Application Programming Interface (API) has grown from about 200 to over 4,000 classes. The API now spans such diverse areas as user interface construction, database management, internationalization, security, and XML processing.

The book that you are reading right now is the first volume of the eleventh edition of *Core Java*. Each edition closely followed a release of the Java Development Kit, and each time, we rewrote the book to take advantage of the newest Java features. This edition has been updated to reflect the features of Java Standard Edition (SE) 9, 10, and 11.

As with the previous editions of this book, *we still target serious programmers who want to put Java to work on real projects.* We think of you, our reader, as a programmer with a solid background in a programming language other than Java, and we assume that you don't like books filled with toy examples (such as toasters, zoo animals, or "nervous text"). You won't find any of these in our book. Our goal is to enable you to fully understand the Java language and library, not to give you an illusion of understanding.

In this book you will find lots of sample code demonstrating almost every language and library feature that we discuss. We keep the sample programs purposefully simple to focus on the major points, but, for the most part, they

aren't fake and they don't cut corners. They should make good starting points for your own code.

We assume you are willing, even eager, to learn about all the advanced features that Java puts at your disposal. For example, we give you a detailed treatment of

- Object-oriented programming
- Reflection and proxies
- Interfaces and inner classes
- Exception handling
- Generic programming
- The collections framework
- The event listener model
- Graphical user interface design
- Concurrency

With the explosive growth of the Java class library, a one-volume treatment of all the features of Java that serious programmers need to know is no longer possible. Hence, we decided to break up the book into two volumes. This first volume concentrates on the fundamental concepts of the Java language, along with the basics of user-interface programming. The second volume, *Core Java, Volume II—Advanced Features*, goes further into the enterprise features and advanced user-interface programming. It includes detailed discussions of

- The Stream API
- File processing and regular expressions
- Databases
- XML processing
- Annotations
- Internationalization
- Network programming
- Advanced GUI components
- Advanced graphics
- Native methods

When writing a book, errors and inaccuracies are inevitable. We'd very much like to know about them. But, of course, we'd prefer to learn about each of them only once. We have put up a list of frequently asked questions, bug fixes, and workarounds on a web page at `http://horstmann.com/corejava`. Strategically placed at the end of the errata page (to encourage you to read through

it first) is a form you can use to report bugs and suggest improvements. Please don't be disappointed if we don't answer every query or don't get back to you immediately. We do read all e-mail and appreciate your input to make future editions of this book clearer and more informative.

A Tour of This Book

Chapter 1 gives an overview of the capabilities of Java that set it apart from other programming languages. We explain what the designers of the language set out to do and to what extent they succeeded. Then, we give a short history of how Java came into being and how it has evolved.

In **Chapter 2**, we tell you how to download and install the JDK and the program examples for this book. Then we guide you through compiling and running a console application and a graphical application. You will see how to use the plain JDK, a Java IDE, and the JShell tool.

Chapter 3 starts the discussion of the Java language. In this chapter, we cover the basics: variables, loops, and simple functions. If you are a C or C++ programmer, this is smooth sailing because the syntax for these language features is essentially the same as in C. If you come from a non-C background such as Visual Basic, you will want to read this chapter carefully.

Object-oriented programming (OOP) is now in the mainstream of programming practice, and Java is an object-oriented programming language. **Chapter 4** introduces encapsulation, the first of two fundamental building blocks of object orientation, and the Java language mechanism to implement it—that is, classes and methods. In addition to the rules of the Java language, we also give advice on sound OOP design. Finally, we cover the marvelous `javadoc` tool that formats your code comments as a set of hyperlinked web pages. If you are familiar with C++, you can browse through this chapter quickly. Programmers coming from a non-object-oriented background should expect to spend some time mastering the OOP concepts before going further with Java.

Classes and encapsulation are only one part of the OOP story, and **Chapter 5** introduces the other—namely, *inheritance*. Inheritance lets you take an existing class and modify it according to your needs. This is a fundamental technique for programming in Java. The inheritance mechanism in Java is quite similar to that in C++. Once again, C++ programmers can focus on the differences between the languages.

Chapter 6 shows you how to use Java's notion of an *interface*. Interfaces let you go beyond the simple inheritance model of Chapter 5. Mastering interfaces allows you to have full access to the power of Java's completely object-oriented approach to programming. After we cover interfaces, we move on to *lambda expressions*, a concise way for expressing a block of code that can be executed at a later point in time. We then cover a useful technical feature of Java called *inner classes*.

Chapter 7 discusses *exception handling*—Java's robust mechanism to deal with the fact that bad things can happen to good programs. Exceptions give you an efficient way of separating the normal processing code from the error handling. Of course, even after hardening your program by handling all exceptional conditions, it still might fail to work as expected. In the final part of this chapter, we give you a number of useful debugging tips.

Chapter 8 gives an overview of generic programming. Generic programming makes your programs easier to read and safer. We show you how to use strong typing and remove unsightly and unsafe casts, and how to deal with the complexities that arise from the need to stay compatible with older versions of Java.

The topic of **Chapter 9** is the collections framework of the Java platform. Whenever you want to collect multiple objects and retrieve them later, you should use a collection that is best suited for your circumstances, instead of just tossing the elements into an array. This chapter shows you how to take advantage of the standard collections that are prebuilt for your use.

Chapter 10 provides an introduction into GUI programming. We show how you can make windows, how to paint on them, how to draw with geometric shapes, how to format text in multiple fonts, and how to display images. Next, you'll see how to write code that responds to events, such as mouse clicks or key presses.

Chapter 11 discusses the Swing GUI toolkit in great detail. The Swing toolkit allows you to build cross-platform graphical user interfaces. You'll learn all about the various kinds of buttons, text components, borders, sliders, list boxes, menus, and dialog boxes. However, some of the more advanced components are discussed in Volume II.

Chapter 12 finishes the book with a discussion of concurrency, which enables you to program tasks to be done in parallel. This is an important and exciting application of Java technology in an era where most processors have multiple cores that you want to keep busy.

A **bonus JavaFX chapter** contains a rapid introduction into JavaFX, a modern GUI toolkit for desktop applications. If you read the print book, download the chapter from the book companion site at `http://horstmann.com/corejava`.

The **Appendix** lists the reserved words of the Java language.

Conventions

As is common in many computer books, we use `monospace type` to represent computer code.

NOTE: Notes are tagged with "note" icons that look like this.

TIP: Tips are tagged with "tip" icons that look like this.

CAUTION: When there is danger ahead, we warn you with a "caution" icon.

C++ NOTE: There are many C++ notes that explain the differences between Java and C++. You can skip over them if you don't have a background in C++ or if you consider your experience with that language a bad dream of which you'd rather not be reminded.

Java comes with a large programming library, or Application Programming Interface (API). When using an API call for the first time, we add a short summary description at the end of the section. These descriptions are a bit more informal but, we hope, also a little more informative than those in the official online API documentation. The names of interfaces are in italics, just like in the official documentation. The number after a class, interface, or method name is the JDK version in which the feature was introduced, as shown in the following example:

```
Application Programming Interface 9
```

Programs whose source code is on the book's companion web site are presented as listings, for instance:

Listing 1.1 `InputTest/InputTest.java`

Sample Code

The web site for this book at `http://horstmann.com/corejava` contains all sample code from the book. See Chapter 2 for more information on installing the Java Development Kit and the sample code.

Register your copy of *Core Java, Volume I—Fundamentals, Eleventh Edition,* on the InformIT site for convenient access to updates and/or corrections as they become available. To start the registration process, go to `informit.com/register` and log in or create an account. Enter the product ISBN (9780135166307) and click Submit. Look on the Registered Products tab for an Access Bonus Content link next to this product, and follow that link to access any available bonus materials. If you would like to be notified of exclusive offers on new editions and updates, please check the box to receive email from us.

Acknowledgments

Writing a book is always a monumental effort, and rewriting it doesn't seem to be much easier, especially with the continuous change in Java technology. Making a book a reality takes many dedicated people, and it is my great pleasure to acknowledge the contributions of the entire *Core Java* team.

A large number of individuals at Pearson provided valuable assistance but managed to stay behind the scenes. I'd like them all to know how much I appreciate their efforts. As always, my warm thanks go to my editor, Greg Doench, for steering the book through the writing and production process, and for allowing me to be blissfully unaware of the existence of all those folks behind the scenes. I am very grateful to Julie Nahil for production support, and to Dmitry Kirsanov and Alina Kirsanova for copyediting and typesetting the manuscript. My thanks also to my coauthor of earlier editions, Gary Cornell, who has since moved on to other ventures.

Thanks to the many readers of earlier editions who reported embarrassing errors and made lots of thoughtful suggestions for improvement. I am particularly grateful to the excellent reviewing team who went over the manuscript with an amazing eye for detail and saved me from many embarrassing errors.

Reviewers of this and earlier editions include Chuck Allison (Utah Valley University), Lance Andersen (Oracle), Paul Anderson (Anderson Software Group), Alec Beaton (IBM), Cliff Berg, Andrew Binstock (Oracle), Joshua Bloch, David Brown, Corky Cartwright, Frank Cohen (PushToTest), Chris Crane (devXsolution), Dr. Nicholas J. De Lillo (Manhattan College), Rakesh Dhoopar (Oracle), David Geary (Clarity Training), Jim Gish (Oracle), Brian Goetz (Oracle), Angela Gordon, Dan Gordon (Electric Cloud), Rob Gordon, John Gray (University of Hartford), Cameron Gregory (olabs.com), Marty Hall (coreservlets.com, Inc.), Vincent Hardy (Adobe Systems), Dan Harkey (San Jose State University), William Higgins (IBM), Vladimir Ivanovic (PointBase), Jerry Jackson (CA Technologies), Tim Kimmet (Walmart), Chris Laffra, Charlie Lai (Apple), Angelika Langer, Doug Langston, Hang Lau (McGill University), Mark Lawrence, Doug Lea (SUNY Oswego), Gregory Longshore, Bob Lynch (Lynch Associates), Philip Milne (consultant), Mark Morrissey (The Oregon Graduate Institute), Mahesh Neelakanta (Florida Atlantic University), Hao Pham, Paul Philion, Blake Ragsdell, Stuart Reges (University of Arizona), Simon Ritter (Azul Systems), Rich Rosen (Interactive Data Corporation), Peter Sanders (ESSI University, Nice, France), Dr. Paul Sanghera (San

Jose State University and Brooks College), Paul Sevinc (Teamup AG), Devang Shah (Sun Microsystems), Yoshiki Shibata, Bradley A. Smith, Steven Stelting (Oracle), Christopher Taylor, Luke Taylor (Valtech), George Thiruvathukal, Kim Topley (StreamingEdge), Janet Traub, Paul Tyma (consultant), Peter van der Linden, Christian Ullenboom, Burt Walsh, Dan Xu (Oracle), and John Zavgren (Oracle).

Cay Horstmann
San Francisco, California
June 2018

CHAPTER 1

An Introduction to Java

In this chapter

The first release of Java in 1996 generated an incredible amount of excitement, not just in the computer press, but in mainstream media such as the *New York Times*, the *Washington Post*, and *BusinessWeek*. Java has the distinction of being the first and only programming language that had a ten-minute story on National Public Radio. A $100,000,000 venture capital fund was set up solely for products using a *specific* computer language. I hope you will enjoy a brief history of Java that you will find in this chapter.

1.1 Java as a Programming Platform

In the first edition of this book, my coauthor Gary Cornell and I had this to write about Java:

"As a computer language, Java's hype is overdone: Java is certainly a *good* programming language. There is no doubt that it is one of the better languages available to serious programmers. We think it could *potentially* have been a

great programming language, but it is probably too late for that. Once a language is out in the field, the ugly reality of compatibility with existing code sets in."

Our editor got a lot of flack for this paragraph from someone very high up at Sun Microsystems, the company that originally developed Java. The Java language has a lot of nice features that we will examine in detail later in this chapter. It has its share of warts, and some of the newer additions to the language are not as elegant as the original features because of compatibility requirements.

But, as we already said in the first edition, Java was never just a language. There are lots of programming languages out there, but few of them make much of a splash. Java is a whole *platform*, with a huge library, containing lots of reusable code, and an execution environment that provides services such as security, portability across operating systems, and automatic garbage collection.

As a programmer, you will want a language with a pleasant syntax and comprehensible semantics (i.e., not C++). Java fits the bill, as do dozens of other fine languages. Some languages give you portability, garbage collection, and the like, but they don't have much of a library, forcing you to roll your own if you want fancy graphics or networking or database access. Well, Java has everything—a good language, a high-quality execution environment, and a vast library. That combination is what makes Java an irresistible proposition to so many programmers.

1.2 The Java "White Paper" Buzzwords

The authors of Java wrote an influential white paper that explains their design goals and accomplishments. They also published a shorter overview that is organized along the following 11 buzzwords:

1. Simple
2. Object-Oriented
3. Distributed
4. Robust
5. Secure
6. Architecture-Neutral
7. Portable
8. Interpreted

9. High-Performance
10. Multithreaded
11. Dynamic

In the following subsections, you will find a summary, with excerpts from the white paper, of what the Java designers say about each buzzword, together with a commentary based on my experiences with the current version of Java.

NOTE: The white paper can be found at `www.oracle.com/technetwork/java/langenv-140151.html`. You can retrieve the overview with the 11 buzzwords at `http://horstmann.com/corejava/java-an-overview/7Gosling.pdf`.

1.2.1 Simple

> *We wanted to build a system that could be programmed easily without a lot of esoteric training and which leveraged today's standard practice. So even though we found that C++ was unsuitable, we designed Java as closely to C++ as possible in order to make the system more comprehensible. Java omits many rarely used, poorly understood, confusing features of C++ that, in our experience, bring more grief than benefit.*

The syntax for Java is, indeed, a cleaned-up version of C++ syntax. There is no need for header files, pointer arithmetic (or even a pointer syntax), structures, unions, operator overloading, virtual base classes, and so on. (See the C++ notes interspersed throughout the text for more on the differences between Java and C++.) The designers did not, however, attempt to fix all of the clumsy features of C++. For example, the syntax of the `switch` statement is unchanged in Java. If you know C++, you will find the transition to the Java syntax easy.

At the time Java was released, C++ was actually not the most commonly used programming language. Many developers used Visual Basic and its drag-and-drop programming environment. These developers did not find Java simple. It took several years for Java development environments to catch up. Nowadays, Java development environments are far ahead of those for most other programming languages.

> *Another aspect of being simple is being small. One of the goals of Java is to enable the construction of software that can run stand-alone on small machines. The size of the basic interpreter and class support is about 40K; the basic standard libraries and thread support (essentially a self-contained microkernel) add another 175K.*

This was a great achievement at the time. Of course, the library has since grown to huge proportions. There is now a separate Java Micro Edition with a smaller library, suitable for embedded devices.

1.2.2 Object-Oriented

> *Simply stated, object-oriented design is a programming technique that focuses on the data—objects—and on the interfaces to those objects. To make an analogy with carpentry, an "object-oriented" carpenter would be mostly concerned with the chair he is building, and secondarily with the tools used to make it; a "non-object-oriented" carpenter would think primarily of his tools. The object-oriented facilities of Java are essentially those of C++.*

Object orientation was pretty well established when Java was developed. The object-oriented features of Java are comparable to those of C++. The major difference between Java and C++ lies in multiple inheritance, which Java has replaced with a simpler concept of interfaces. Java has a richer capacity for runtime introspection (discussed in Chapter 5) than C++.

1.2.3 Distributed

> *Java has an extensive library of routines for coping with TCP/IP protocols like HTTP and FTP. Java applications can open and access objects across the Net via URLs with the same ease as when accessing a local file system.*

Nowadays, one takes this for granted—but in 1995, connecting to a web server from a C++ or Visual Basic program was a major undertaking.

1.2.4 Robust

> *Java is intended for writing programs that must be reliable in a variety of ways. Java puts a lot of emphasis on early checking for possible problems, later dynamic (runtime) checking, and eliminating situations that are error-prone. . . . The single biggest difference between Java and C/C++ is that Java has a pointer model that eliminates the possibility of overwriting memory and corrupting data.*

The Java compiler detects many problems that in other languages would show up only at runtime. As for the second point, anyone who has spent hours chasing memory corruption caused by a pointer bug will be very happy with this aspect of Java.

1.2.5 Secure

Java is intended to be used in networked/distributed environments. Toward that end, a lot of emphasis has been placed on security. Java enables the construction of virus-free, tamper-free systems.

From the beginning, Java was designed to make certain kinds of attacks impossible, among them:

- Overrunning the runtime stack—a common attack of worms and viruses
- Corrupting memory outside its own process space
- Reading or writing files without permission

Originally, the Java attitude towards downloaded code was "Bring it on!" Untrusted code was executed in a sandbox environment where it could not impact the host system. Users were assured that nothing bad could happen because Java code, no matter where it came from, could never escape from the sandbox.

However, the security model of Java is complex. Not long after the first version of the Java Development Kit was shipped, a group of security experts at Princeton University found subtle bugs that allowed untrusted code to attack the host system.

Initially, security bugs were fixed quickly. Unfortunately, over time, hackers got quite good at spotting subtle flaws in the implementation of the security architecture. Sun, and then Oracle, had a tough time keeping up with bug fixes.

After a number of high-profile attacks, browser vendors and Oracle became increasingly cautious. Java browser plug-ins no longer trust remote code unless it is digitally signed and users have agreed to its execution.

NOTE: Even though in hindsight, the Java security model was not as successful as originally envisioned, Java was well ahead of its time. A competing code delivery mechanism from Microsoft relied on digital signatures alone for security. Clearly this was not sufficient: As any user of Microsoft's own products can confirm, programs from well-known vendors do crash and create damage.

1.2.6 Architecture-Neutral

> *The compiler generates an architecture-neutral object file format. The compiled code is executable on many processors, given the presence of the Java runtime system. The Java compiler does this by generating bytecode instructions which have nothing to do with a particular computer architecture. Rather, they are designed to be both easy to interpret on any machine and easy to translate into native machine code on the fly.*

Generating code for a "virtual machine" was not a new idea at the time. Programming languages such as Lisp, Smalltalk, and Pascal had employed this technique for many years.

Of course, interpreting virtual machine instructions is slower than running machine instructions at full speed. However, virtual machines have the option of translating the most frequently executed bytecode sequences into machine code—a process called just-in-time compilation.

Java's virtual machine has another advantage. It increases security because it can check the behavior of instruction sequences.

1.2.7 Portable

> *Unlike C and C++, there are no "implementation-dependent" aspects of the specification. The sizes of the primitive data types are specified, as is the behavior of arithmetic on them.*

For example, an `int` in Java is always a 32-bit integer. In C/C++, `int` can mean a 16-bit integer, a 32-bit integer, or any other size that the compiler vendor likes. The only restriction is that the `int` type must have at least as many bytes as a `short int` and cannot have more bytes than a `long int`. Having a fixed size for number types eliminates a major porting headache. Binary data is stored and transmitted in a fixed format, eliminating confusion about byte ordering. Strings are saved in a standard Unicode format.

> *The libraries that are a part of the system define portable interfaces. For example, there is an abstract* `Window` *class and implementations of it for UNIX, Windows, and the Macintosh.*

The example of a `Window` class was perhaps poorly chosen. As anyone who has ever tried knows, it is an effort of heroic proportions to implement a user interface that looks good on Windows, the Macintosh, and ten flavors of UNIX. Java 1.0 made the heroic effort, delivering a simple toolkit that provided common user interface elements on a number of platforms. Unfortunately, the result was a library that, with a lot of work, could give barely acceptable

results on different systems. That initial user interface toolkit has since been replaced, and replaced again, and portability across platforms remains an issue.

However, for everything that isn't related to user interfaces, the Java libraries do a great job of letting you work in a platform-independent manner. You can work with files, regular expressions, XML, dates and times, databases, network connections, threads, and so on, without worrying about the underlying operating system. Not only are your programs portable, but the Java APIs are often of higher quality than the native ones.

1.2.8 Interpreted

> *The Java interpreter can execute Java bytecodes directly on any machine to which the interpreter has been ported. Since linking is a more incremental and lightweight process, the development process can be much more rapid and exploratory.*

This was a real stretch. Anyone who has used Lisp, Smalltalk, Visual Basic, Python, R, or Scala knows what a "rapid and exploratory" development process is. You try out something, and you instantly see the result. For the first 20 years of Java's existence, development environments were not focused on that experience. It wasn't until Java 9 that the `jshell` tool supported rapid and exploratory programming.

1.2.9 High-Performance

> *While the performance of interpreted bytecodes is usually more than adequate, there are situations where higher performance is required. The bytecodes can be translated on the fly (at runtime) into machine code for the particular CPU the application is running on.*

In the early years of Java, many users disagreed with the statement that the performance was "more than adequate." Today, however, the just-in-time compilers have become so good that they are competitive with traditional compilers and, in some cases, even outperform them because they have more information available. For example, a just-in-time compiler can monitor which code is executed frequently and optimize just that code for speed. A more sophisticated optimization is the elimination (or "inlining") of function calls. The just-in-time compiler knows which classes have been loaded. It can use inlining when, based upon the currently loaded collection of classes, a particular function is never overridden, and it can undo that optimization later if necessary.

1.2.10 Multithreaded

[The] benefits of multithreading are better interactive responsiveness and real-time behavior.

Nowadays, we care about concurrency because Moore's law has come to an end. Instead of faster processors, we just get more of them, and we have to keep them busy. Yet when you look at most programming languages, they show a shocking disregard for this problem.

Java was well ahead of its time. It was the first mainstream language to support concurrent programming. As you can see from the white paper, its motivation was a little different. At the time, multicore processors were exotic, but web programming had just started, and processors spent a lot of time waiting for a response from the server. Concurrent programming was needed to make sure the user interface didn't freeze.

Concurrent programming is never easy, but Java has done a very good job making it manageable.

1.2.11 Dynamic

In a number of ways, Java is a more dynamic language than C or C++. It was designed to adapt to an evolving environment. Libraries can freely add new methods and instance variables without any effect on their clients. In Java, finding out runtime type information is straightforward.

This is an important feature in situations where code needs to be added to a running program. A prime example is code that is downloaded from the Internet to run in a browser. In C or C++, this is indeed a major challenge, but the Java designers were well aware of dynamic languages that made it easy to evolve a running program. Their achievement was to bring this feature to a mainstream programming language.

NOTE: Shortly after the initial success of Java, Microsoft released a product called J++ with a programming language and virtual machine that were almost identical to Java. This effort failed to gain traction, and Microsoft followed through with another language called C# that also has many similarities to Java but runs on a different virtual machine. This book does not cover J++ or C#.

1.3 Java Applets and the Internet

The idea here is simple: Users will download Java bytecodes from the Internet and run them on their own machines. Java programs that work on web pages are called *applets*. To use an applet, you only need a Java-enabled web browser, which will execute the bytecodes for you. You need not install any software. You get the latest version of the program whenever you visit the web page containing the applet. Most importantly, thanks to the security of the virtual machine, you never need to worry about attacks from hostile code.

Inserting an applet into a web page works much like embedding an image. The applet becomes a part of the page, and the text flows around the space used for the applet. The point is, this image is *alive*. It reacts to user commands, changes its appearance, and exchanges data between the computer presenting the applet and the computer serving it.

Figure 1.1 shows the Jmol applet that displays molecular structures. By using the mouse, you can rotate and zoom each molecule to better understand its structure. At the time that applets were invented, this kind of direct manipulation was not achievable with web pages—there was only rudimentary JavaScript and no HTML canvas.

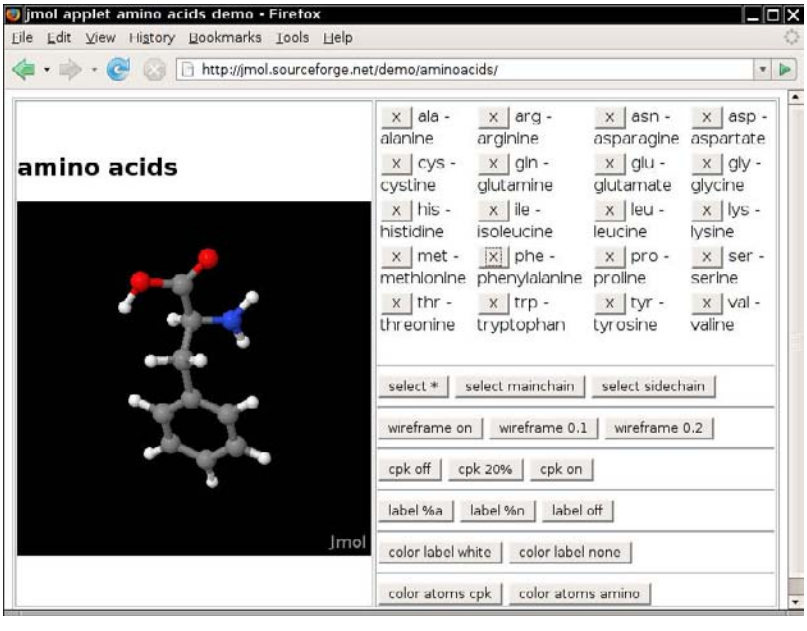

Figure 1.1 The Jmol applet

When applets first appeared, they created a huge amount of excitement. Many people believe that the lure of applets was responsible for the astonishing popularity of Java. However, the initial excitement soon turned into frustration. Various versions of the Netscape and Internet Explorer browsers ran different versions of Java, some of which were seriously outdated. This sorry situation made it increasingly difficult to develop applets that took advantage of the most current Java version. Instead, Adobe's Flash technology became popular for achieving dynamic effects in the browser. Later, when Java was dogged by serious security issues, browsers and the Java browser plug-in became increasingly restrictive. Nowadays, it requires skill and dedication to get applets to work in your browser. For example, if you visit the Jmol web site at `http://jmol.sourceforge.net/demo/aminoacids/`, you will likely encounter a message exhorting you to configure your browser for allowing applets to run.

1.4 A Short History of Java

This section gives a short history of Java's evolution. It is based on various published sources (most importantly an interview with Java's creators in the July 1995 issue of *SunWorld*'s online magazine).

Java goes back to 1991, when a group of Sun engineers, led by Patrick Naughton and James Gosling (a Sun Fellow and an all-around computer wizard), wanted to design a small computer language that could be used for consumer devices like cable TV switchboxes. Since these devices do not have a lot of power or memory, the language had to be small and generate very tight code. Also, as different manufacturers may choose different central processing units (CPUs), it was important that the language not be tied to any single architecture. The project was code-named "Green."

The requirements for small, tight, and platform-neutral code led the team to design a portable language that generated intermediate code for a virtual machine.

The Sun people came from a UNIX background, so they based their language on C++ rather than Lisp, Smalltalk, or Pascal. But, as Gosling says in the interview, "All along, the language was a tool, not the end." Gosling decided to call his language "Oak" (presumably because he liked the look of an oak tree that was right outside his window at Sun). The people at Sun later realized that Oak was the name of an existing computer language, so they changed the name to Java. This turned out to be an inspired choice.

In 1992, the Green project delivered its first product, called "*7." It was an extremely intelligent remote control. Unfortunately, no one was interested in producing this at Sun, and the Green people had to find other ways to market

their technology. However, none of the standard consumer electronics companies were interested either. The group then bid on a project to design a cable TV box that could deal with emerging cable services such as video-on-demand. They did not get the contract. (Amusingly, the company that did was led by the same Jim Clark who started Netscape—a company that did much to make Java successful.)

The Green project (with a new name of "First Person, Inc.") spent all of 1993 and half of 1994 looking for people to buy its technology. No one was found. (Patrick Naughton, one of the founders of the group and the person who ended up doing most of the marketing, claims to have accumulated 300,000 air miles in trying to sell the technology.) First Person was dissolved in 1994.

While all of this was going on at Sun, the World Wide Web part of the Internet was growing bigger and bigger. The key to the World Wide Web was the browser translating hypertext pages to the screen. In 1994, most people were using Mosaic, a noncommercial web browser that came out of the supercomputing center at the University of Illinois in 1993. (Mosaic was partially written by Marc Andreessen as an undergraduate student on a work-study project, for $6.85 an hour. He moved on to fame and fortune as one of the cofounders and the chief of technology at Netscape.)

In the *SunWorld* interview, Gosling says that in mid-1994, the language developers realized that "We could build a real cool browser. It was one of the few things in the client/server mainstream that needed some of the weird things we'd done: architecture-neutral, real-time, reliable, secure—issues that weren't terribly important in the workstation world. So we built a browser."

The actual browser was built by Patrick Naughton and Jonathan Payne and evolved into the HotJava browser, which was designed to show off the power of Java. The browser was capable of executing Java code inside web pages. This "proof of technology" was shown at SunWorld '95 on May 23, 1995, and inspired the Java craze that continues today.

Sun released the first version of Java in early 1996. People quickly realized that Java 1.0 was not going to cut it for serious application development. Sure, you could use Java 1.0 to make a nervous text applet that moved text randomly around in a canvas. But you couldn't even *print* in Java 1.0. To be blunt, Java 1.0 was not ready for prime time. Its successor, version 1.1, filled in the most obvious gaps, greatly improved the reflection capability, and added a new event model for GUI programming. It was still rather limited, though.

The big news of the 1998 JavaOne conference was the upcoming release of Java 1.2, which replaced the early toylike GUI and graphics toolkits with sophisticated scalable versions. Three days (!) after its release in December

1998, Sun's marketing department changed the name to the catchy *Java 2 Standard Edition Software Development Kit Version 1.2*.

Besides the Standard Edition, two other editions were introduced: the Micro Edition for embedded devices such as cell phones, and the Enterprise Edition for server-side processing. This book focuses on the Standard Edition.

Versions 1.3 and 1.4 of the Standard Edition were incremental improvements over the initial Java 2 release, with an ever-growing standard library, increased performance, and, of course, quite a few bug fixes. During this time, much of the initial hype about Java applets and client-side applications abated, but Java became the platform of choice for server-side applications.

Version 5.0 was the first release since version 1.1 that updated the Java *language* in significant ways. (This version was originally numbered 1.5, but the version number jumped to 5.0 at the 2004 JavaOne conference.) After many years of research, generic types (roughly comparable to C++ templates) have been added—the challenge was to add this feature without requiring changes in the virtual machine. Several other useful language features were inspired by C#: a "for each" loop, autoboxing, and annotations.

Version 6 (without the .0 suffix) was released at the end of 2006. Again, there were no language changes but additional performance improvements and library enhancements.

As datacenters increasingly relied on commodity hardware instead of specialized servers, Sun Microsystems fell on hard times and was purchased by Oracle in 2009. Development of Java stalled for a long time. In 2011, Oracle released a new version, with simple enhancements, as Java 7.

In 2014, the release of Java 8 followed, with the most significant changes to the Java language in almost two decades. Java 8 embraces a "functional" style of programming that makes it easy to express computations that can be executed concurrently. All programming languages must evolve to stay relevant, and Java has shown a remarkable capacity to do so.

The main feature of Java 9 goes all the way back to 2008. At that time, Mark Reinhold, the chief engineer of the Java platform, started an effort to break up the huge, monolithic Java platform. This was to be achieved by introducing *modules*, self-contained units of code that provide a specific functionality. It took eleven years to design and implement a module system that is a good fit for the Java platform, and it remains to be seen whether it is also a good fit for Java applications and libraries. Java 9, released in 2017, has other appealing features that we cover in this book.

Starting in 2018, Java versions are released every six months, to enable faster introduction of features. Certain versions, such as Java 11, are designated as long-term support versions.

Table 1.1 shows the evolution of the Java language and library. As you can see, the size of the application programming interface (API) has grown tremendously.

Table 1.1 Evolution of the Java Language

Version	Year	New Language Features	Number of Classes and Interfaces
1.0	1996	The language itself	211
1.1	1997	Inner classes	477
1.2	1998	The `strictfp` modifier	1,524
1.3	2000	None	1,840
1.4	2002	Assertions	2,723
5.0	2004	Generic classes, "for each" loop, varargs, autoboxing, metadata, enumerations, static import	3,279
6	2006	None	3,793
7	2011	Switch with strings, diamond operator, binary literals, exception handling enhancements	4,024
8	2014	Lambda expressions, interfaces with default methods, stream and date/time libraries	4,240
9	2017	Modules, miscellaneous language and library enhancements	6,005

1.5 Common Misconceptions about Java

This chapter closes with a commented list of some common misconceptions about Java.

Java is an extension of HTML.

Java is a programming language; HTML is a way to describe the structure of a web page. They have nothing in common except that there are HTML extensions for placing Java applets on a web page.

I use XML, so I don't need Java.

Java is a programming language; XML is a way to describe data. You can process XML data with any programming language, but the Java API contains excellent support for XML processing. In addition, many important XML tools are implemented in Java. See Volume II for more information.

Java is an easy programming language to learn.

No programming language as powerful as Java is easy. You always have to distinguish between how easy it is to write toy programs and how hard it is to do serious work. Also, consider that only seven chapters in this book discuss the Java language. The remaining chapters of both volumes show how to put the language to work, using the Java *libraries*. The Java libraries contain thousands of classes and interfaces and tens of thousands of functions. Luckily, you do not need to know every one of them, but you do need to know surprisingly many to use Java for anything realistic.

Java will become a universal programming language for all platforms.

This is possible in theory. But in practice, there are domains where other languages are entrenched. Objective C and its successor, Swift, are not going to be replaced on iOS devices. Anything that happens in a browser is controlled by JavaScript. Windows programs are written in C++ or C#. Java has the edge in server-side programming and in cross-platform client applications.

Java is just another programming language.

Java is a nice programming language; most programmers prefer it to C, C++, or C#. But there have been hundreds of nice programming languages that never gained widespread popularity, whereas languages with obvious flaws, such as C++ and Visual Basic, have been wildly successful.

Why? The success of a programming language is determined far more by the utility of the *support system* surrounding it than by the elegance of its syntax. Are there useful, convenient, and standard libraries for the features that you need to implement? Are there tool vendors that build great programming and debugging environments? Do the language and the toolset integrate with the rest of the computing infrastructure? Java is successful because its libraries let you easily do things such as networking, web applications, and concurrency. The fact that Java reduces pointer errors is a bonus, so programmers seem to be more productive with Java—but these factors are not the source of its success.

Java is proprietary, and should therefore be avoided.

When Java was first created, Sun gave free licenses to distributors and end users. Although Sun had ultimate control over Java, they involved many

other companies in the development of language revisions and the design of new libraries. Source code for the virtual machine and the libraries has always been freely available, but only for inspection, not for modification and redistribution. Java was "closed source, but playing nice."

This situation changed dramatically in 2007, when Sun announced that future versions of Java would be available under the General Public License (GPL), the same open source license that is used by Linux. Oracle has committed to keeping Java open source. There is only one fly in the ointment—patents. Everyone is given a patent grant to use and modify Java, subject to the GPL, but only on desktop and server platforms. If you want to use Java in embedded systems, you need a different license and will likely need to pay royalties. However, these patents will expire within the next decade, and at that point Java will be entirely free.

Java is interpreted, so it is too slow for serious applications.

In the early days of Java, the language was interpreted. Nowadays, the Java virtual machine uses a just-in-time compiler. The "hot spots" of your code will run just as fast in Java as they would in C++, and in some cases even faster.

All Java programs run inside a web page.

All Java *applets* run inside a web browser. That is the definition of an applet—a Java program running inside a browser. But most Java programs are stand-alone applications that run outside of a web browser. In fact, many Java programs run on web servers and produce the code for web pages.

Java programs are a major security risk.

In the early days of Java, there were some well-publicized reports of failures in the Java security system. Researchers viewed it as a challenge to find chinks in the Java armor and to defy the strength and sophistication of the applet security model. The technical failures that they found have all been quickly corrected. Later, there were more serious exploits, to which Sun, and later Oracle, responded too slowly. Browser manufacturers reacted, and perhaps overreacted, by deactivating Java by default. To keep this in perspective, consider the far greater number of virus attacks in Windows executable files that cause real grief but surprisingly little criticism of the weaknesses of the attacked platform. Even 20 years after its creation, Java is far safer than any other commonly available execution platform.

JavaScript is a simpler version of Java.

JavaScript, a scripting language that can be used inside web pages, was invented by Netscape and originally called LiveScript. JavaScript has a syntax

that is reminiscent of Java, and the languages' names sound similar, but otherwise they are unrelated. In particularly, Java is *strongly typed*—the compiler catches many errors that arise from type misuse. In JavaScript, such errors are only found when the program runs, which makes their elimination far more laborious.

With Java, I can replace my desktop computer with a cheap "Internet appliance."

When Java was first released, some people bet big that this was going to happen. Companies produced prototypes of Java-powered network computers, but users were not ready to give up a powerful and convenient desktop for a limited machine with no local storage. Nowadays, of course, the world has changed, and for a large majority of end users, the platform that matters is a mobile phone or tablet. The majority of these devices are controlled by the Android platform, which is a derivative of Java. Learning Java programming will help you with Android programming as well.

CHAPTER 2

The Java Programming Environment

In this chapter

In this chapter, you will learn how to install the Java Development Kit (JDK) and how to compile and run various types of programs: console programs, graphical applications, and applets. You can run the JDK tools by typing commands in a terminal window. However, many programmers prefer the comfort of an integrated development environment. You will learn how to use a freely available development environment to compile and run Java programs. Once you have mastered the techniques in this chapter and picked your development tools, you are ready to move on to Chapter 3, where you will begin exploring the Java programming language.

2.1 Installing the Java Development Kit

The most complete and up-to-date versions of the Java Development Kit (JDK) are available from Oracle for Linux, Mac OS, Solaris, and Windows. Versions in various states of development exist for many other platforms, but those versions are licensed and distributed by the vendors of those platforms.

2.1.1 Downloading the JDK

To download the Java Development Kit, visit the web site at `www.oracle.com/technetwork/java/javase/downloads` and be prepared to decipher an amazing amount of jargon before you can get the software you need. See Table 2.1 for a summary.

Table 2.1 Java Jargon

Name	Acronym	Explanation
Java Development Kit	JDK	The software for programmers who want to write Java programs
Java Runtime Environment	JRE	The software for consumers who want to run Java programs
Server JRE	—	The software for running Java programs on servers
Standard Edition	SE	The Java platform for use on desktops and simple server applications
Enterprise Edition	EE	The Java platform for complex server applications
Micro Edition	ME	The Java platform for use on small devices
JavaFX	—	An alternate toolkit for graphical user interfaces that is included with certain Java SE distributions prior to Java 11
OpenJDK	—	A free and open source implementation of Java SE
Java 2	J2	An outdated term that described Java versions from 1998 until 2006
Software Development Kit	SDK	An outdated term that described the JDK from 1998 until 2006
Update	u	Oracle's term for a bug fix release up to Java 8
NetBeans	—	Oracle's integrated development environment

You already saw the abbreviation JDK for Java Development Kit. Somewhat confusingly, versions 1.2 through 1.4 of the kit were known as the Java SDK (Software Development Kit). You will still find occasional references to the old term. Up to Java 10, there is also a Java Runtime Environment (JRE) that contains only the virtual machine. That is not what you want as a developer. It is intended for end users who have no need for the compiler.

Next, you'll see the term Java SE everywhere. That is the Java Standard Edition, in contrast to Java EE (Enterprise Edition) and Java ME (Micro Edition).

You might run into the term Java 2 that was coined in 1998 when the marketing folks at Sun felt that a fractional version number increment did not properly communicate the momentous advances of JDK 1.2. However, since they had that insight only after the release, they decided to keep the version number 1.2 for the *development kit*. Subsequent releases were numbered 1.3, 1.4, and 5.0. The *platform*, however, was renamed from Java to Java 2. Thus, we had Java 2 Standard Edition Software Development Kit Version 5.0, or J2SE SDK 5.0.

Fortunately, in 2006, the numbering was simplified. The next version of the Java Standard Edition was called Java SE 6, followed by Java SE 7 and Java SE 8.

However, the "internal" version numbers are 1.6.0, 1.7.0, and 1.8.0. This minor madness finally ran its course with Java SE 9, when the version number became 9, and then 9.0.1. (Why not 9.0.0 for the initial version? To keep a modicum of excitement, the version number specification requires that trailing zeroes are dropped for the fleeting interval between a major release and its first security update.)

NOTE: For the remainder of the book, we will drop the "SE" acronym. When you see "Java 9", that means "Java SE 9".

Prior to Java 9, there were 32-bit and 64-bit versions of the Java Development Kit. The 32-bit versions are no longer developed by Oracle. You need to have a 64-bit operating system to use the Oracle JDK.

With Linux, you have a choice between an RPM file and a `.tar.gz` file. We recommend the latter—you can simply uncompress it anywhere you like.

Now you know how to pick the right JDK. To summarize:

- You want the JDK (Java SE Development Kit), not the JRE.
- Linux: Pick the `.tar.gz` version.

Accept the license agreement and download the file.

NOTE: Depending on the constellation of the planets, Oracle may offer you a bundle that contains both the Java Development Kit and the NetBeans integrated development environment. I suggest that you stay away from all bundles and install only the Java Development Kit at this time. If you later decide to use NetBeans, simply download it from `http://netbeans.org`.

2.1.2 Setting up the JDK

After downloading the JDK, you need to install it and figure out where it was installed—you'll need that information later.

- Under Windows, launch the setup program. You will be asked where to install the JDK. It is best not to accept a default location with spaces in the path name, such as `c:\Program Files\Java\jdk-11.0.`*x*. Just take out the `Program Files` part of the path name.
- On the Mac, run the installer. It installs the software into `/Library/Java/JavaVirtualMachines/jdk-11.0.`*x*`.jdk/Contents/Home`. Locate it with the Finder.
- On Linux, simply uncompress the `.tar.gz` file to a location of your choice, such as your home directory or `/opt`. Or, if you installed from the RPM file, double-check that it is installed in `/usr/java/jdk-11.0.`*x*.

In this book, the installation directory is denoted as *jdk*. For example, when referring to the *jdk*`/bin` directory, I mean the directory with a name such as `/opt/jdk-11.0.4/bin` or `c:\Java\jdk-11.0.4\bin`.

When you install the JDK on Windows or Linux, you need to carry out one additional step: Add the *jdk*`/bin` directory to the executable path—the list of directories that the operating system traverses to locate executable files.

- On Linux, add a line such as the following to the end of your `~/.bashrc` or `~/.bash_profile` file:

  ```
  export PATH=jdk/bin:$PATH
  ```

  Be sure to use the correct path to the JDK, such as `/opt/jdk-11.0.4`.

- Under Windows 10, type "environment" into the search bar of the Windows Settings, and select "Edit environment variables for your account" (see Figure 2.1). An Environment Variables dialog should appear. (It may hide

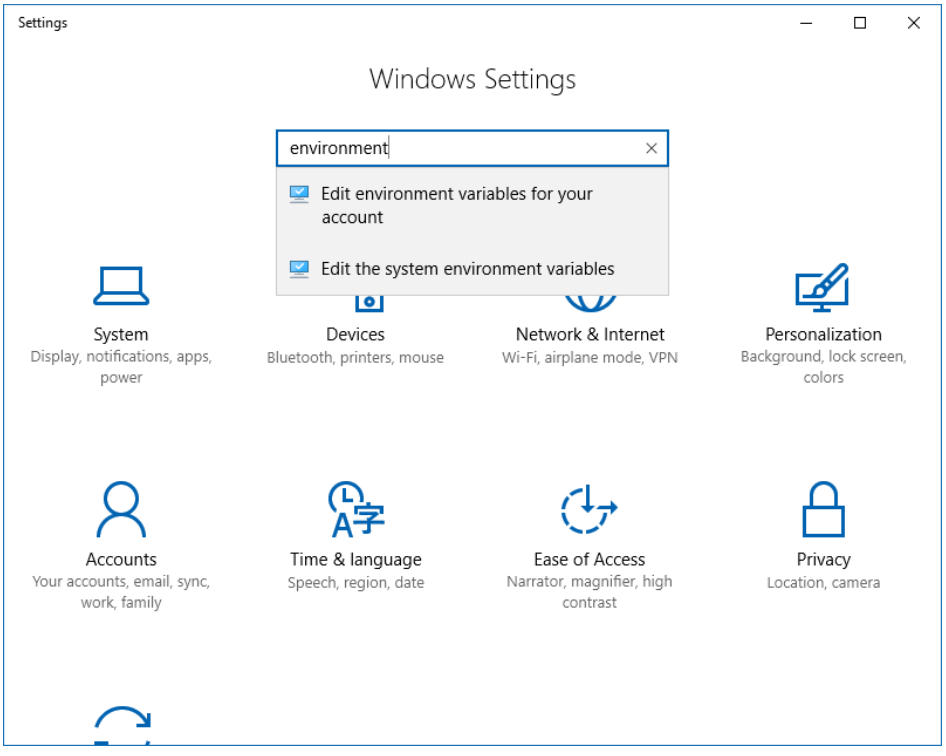

Figure 2.1 Setting system properties in Windows 10

behind the Windows Settings dialog. If you can't find it anywhere, try running `sysdm.cpl` from the Run dialog that you get by holding down the Windows and R key at the same time, and then select the Advanced tab and click the Environment Variables button.) Locate and select a variable named `Path` in the User Variables list. Click the Edit button, then the New button, and add an entry with the *jdk*`\bin` directory (see Figure 2.2).

Save your settings. Any new "Command Prompt" windows that you start will have the correct path.

Here is how you test whether you did it right: Start a terminal window. Type the line

```
javac --version
```

and press the Enter key. You should get a display such as this one:

```
javac 11.0.1
```

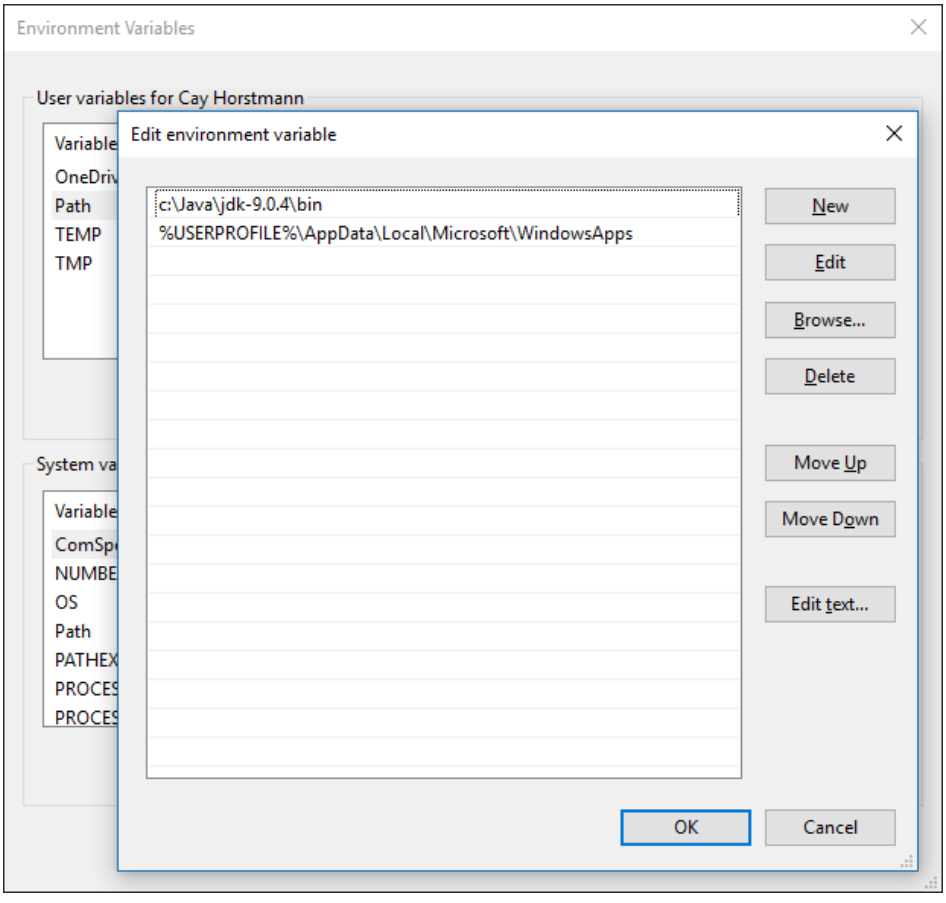

Figure 2.2 Setting the `Path` environment variable in Windows 10

If instead you get a message such as "javac: command not found" or "The name specified is not recognized as an internal or external command, operable program or batch file," then you need to go back and double-check your installation.

2.1.3 Installing Source Files and Documentation

The library source files are delivered in the JDK as a compressed file `lib/src.zip`. Unpack that file to get access to the source code. Simply do the following:

1. Make sure the JDK is installed and the *jdk*`/bin` directory is on the executable path.

2. Make a directory javasrc in your home directory. If you like, you can do this from a terminal window.

```
mkdir javasrc
```

3. Inside the *jdk*/lib directory, locate the file src.zip.
4. Unzip the src.zip file into the javasrc directory. In a terminal window, you can execute the commands

```
cd javasrc
jar xvf jdk/lib/src.zip
cd ..
```

TIP: The src.zip file contains the source code for all public libraries. To obtain even more source (for the compiler, the virtual machine, the native methods, and the private helper classes), go to http://openjdk.java.net.

The documentation is contained in a compressed file that is separate from the JDK. You can download the documentation from www.oracle.com/technetwork/java/javase/downloads. Follow these steps:

1. Download the documentation zip file. It is called jdk-11.0.*x*_doc-all.zip.
2. Unzip the file and rename the doc directory into something more descriptive, like javadoc. If you like, you can do this from the command line:

```
jar xvf Downloads/jdk-11.0.x_doc-all.zip
mv docs jdk-11-docs
```

3. In your browser, navigate to jdk-11-docs/index.html and add this page to your bookmarks.

You should also install the *Core Java* program examples. You can download them from http://horstmann.com/corejava. The programs are packaged into a zip file corejava.zip. Just unzip them into your home directory. They will be located in a directory corejava. If you like, you can do this from the command line:

```
jar xvf Downloads/corejava.zip
```

2.2 Using the Command-Line Tools

If your programming experience comes from a development environment such as Microsoft Visual Studio, you are accustomed to a system with a built-in text editor, menus to compile and launch a program, and a debugger. The JDK contains nothing even remotely similar. You do *everything* by typing in commands in a terminal window. This sounds cumbersome, but it is

nevertheless an essential skill. When you first install Java, you will want to troubleshoot your installation before you install a development environment. Moreover, by executing the basic steps yourself, you gain a better understanding of what a development environment does behind your back.

However, after you have mastered the basic steps of compiling and running Java programs, you will want to use a professional development environment. You will see how to do that in the following section.

Let's get started the hard way: compiling and launching a Java program from the command line.

1. Open a terminal window.
2. Go to the `corejava/v1ch02/Welcome` directory. (The `corejava` directory is where you installed the source code for the book examples, as explained in Section 2.1.3, "Installing Source Files and Documentation," on p. 22.)
3. Enter the following commands:

```
javac Welcome.java
java Welcome
```

   You should see the output shown in Figure 2.3 in the terminal window.

Figure 2.3 Compiling and running `Welcome.java`

Congratulations! You have just compiled and run your first Java program.

What happened? The `javac` program is the Java compiler. It compiles the file `Welcome.java` into the file `Welcome.class`. The `java` program launches the Java virtual machine. It executes the bytecodes that the compiler placed in the class file.

The `Welcome` program is extremely simple. It merely prints a message to the terminal. You may enjoy looking inside the program, shown in Listing 2.1. You will see how it works in the next chapter.

Listing 2.1 `Welcome/Welcome.java`

```
1  /**
2   * This program displays a greeting for the reader.
3   * @version 1.30 2014-02-27
4   * @author Cay Horstmann
5   */
6  public class Welcome
7  {
8     public static void main(String[] args)
9     {
10       String greeting = "Welcome to Core Java!";
11       System.out.println(greeting);
12       for (int i = 0; i < greeting.length(); i++)
13          System.out.print("=");
14       System.out.println();
15    }
16 }
```

In the age of integrated development environments, many programmers are unfamiliar with running programs in a terminal window. Any number of things can go wrong, leading to frustrating results.

Pay attention to the following points:

- If you type in the program by hand, make sure you correctly enter the uppercase and lowercase letters. In particular, the class name is `Welcome` and not `welcome` or `WELCOME`.
- The compiler requires a *file name* (`Welcome.java`). When you run the program, you specify a *class name* (`Welcome`) without a `.java` or `.class` extension.
- If you get a message such as "Bad command or file name" or "javac: command not found", go back and double-check your installation, in particular the executable path setting.
- If `javac` reports that it cannot find the file `Welcome.java`, you should check whether that file is present in the directory.

Under Linux, check that you used the correct capitalization for `Welcome.java`.

Under Windows, use the `dir` command, *not* the graphical Explorer tool. Some text editors (in particular Notepad) insist on adding an extension `.txt` to every file's name. If you use Notepad to edit `Welcome.java`, it will actually save it as `Welcome.java.txt`. Under the default Windows settings, Explorer conspires with Notepad and hides the `.txt` extension because it belongs to a "known file type." In that case, you need to rename the file, using the `ren` command, or save it again, placing quotes around the file name: `"Welcome.java"`.

- If you launch your program and get an error message complaining about a `java.lang.NoClassDefFoundError`, then carefully check the name of the offending class.

  If you get a complaint about `welcome` (with a lowercase `w`), then you should reissue the `java Welcome` command with an uppercase `W`. As always, case matters in Java.

  If you get a complaint about `Welcome/java`, it means you accidentally typed `java Welcome.java`. Reissue the command as `java Welcome`.

- If you typed `java Welcome` and the virtual machine can't find the `Welcome` class, check if someone has set the `CLASSPATH` environment variable on your system. It is not a good idea to set this variable globally, but some poorly written software installers in Windows do just that. Follow the same procedure as for setting the `PATH` environment variable, but this time, remove the setting.

TIP: The excellent tutorial at `http://docs.oracle.com/javase/tutorial/getStarted/cupojava` goes into much greater detail about the "gotchas" that beginners can run into.

NOTE: In JDK 11, the `javac` command is not required with a single source file. This feature is intended to support shell scripts starting with a "shebang" line `#!/path/to/java`.

The `Welcome` program was not terribly exciting. Next, try out a graphical application. This program is a simple image file viewer that loads and displays an image. As before, compile and run the program from the command line.

1. Open a terminal window.
2. Change to the directory `corejava/v1ch02/ImageViewer`.

3. Enter the following:

```
javac ImageViewer.java
java ImageViewer
```

A new program window pops up with the ImageViewer application. Now, select File → Open and look for an image file to open. (There are a couple of sample files in the same directory.) The image is displayed (see Figure 2.4). To close the program, click on the Close box in the title bar or select File → Exit from the menu.

Figure 2.4 Running the ImageViewer application

Have a quick look at the source code (Listing 2.2). The program is substantially longer than the first program, but it is not too complex if you consider how much code it would take in C or C++ to write a similar application. You'll learn how to write graphical user interfaces like this in Chapter 10.

Listing 2.2 ImageViewer/ImageViewer.java

```
1 import java.awt.*;
2 import java.io.*;
3 import javax.swing.*;
4
5 /**
6  * A program for viewing images.
7  * @version 1.31 2018-04-10
```

(Continues)

Listing 2.2 *(Continued)*

```
 * @author Cay Horstmann
 */
public class ImageViewer
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(() -> {
         var frame = new ImageViewerFrame();
         frame.setTitle("ImageViewer");
         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
         frame.setVisible(true);
      });
   }
}

/**
 * A frame with a label to show an image.
 */
class ImageViewerFrame extends JFrame
{
   private static final int DEFAULT_WIDTH = 300;
   private static final int DEFAULT_HEIGHT = 400;

   public ImageViewerFrame()
   {
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // use a label to display the images
      var label = new JLabel();
      add(label);

      // set up the file chooser
      var chooser = new JFileChooser();
      chooser.setCurrentDirectory(new File("."));

      // set up the menu bar
      var menuBar = new JMenuBar();
      setJMenuBar(menuBar);

      var menu = new JMenu("File");
      menuBar.add(menu);

      var openItem = new JMenuItem("Open");
```

```
51        menu.add(openItem);
52        openItem.addActionListener(event -> {
53           // show file chooser dialog
54           int result = chooser.showOpenDialog(null);
55
56           // if file selected, set it as icon of the label
57           if (result == JFileChooser.APPROVE_OPTION)
58           {
59              String name = chooser.getSelectedFile().getPath();
60              label.setIcon(new ImageIcon(name));
61           }
62        });
63
64        var exitItem = new JMenuItem("Exit");
65        menu.add(exitItem);
66        exitItem.addActionListener(event -> System.exit(0));
67     }
68  }
```

2.3 Using an Integrated Development Environment

In the preceding section, you saw how to compile and run a Java program from the command line. That is a useful skill for troubleshooting, but for most day-to-day work, you should use an integrated development environment. These environments are so powerful and convenient that it simply doesn't make much sense to labor on without them. Excellent choices are the freely available Eclipse, IntelliJ IDEA, and NetBeans. In this chapter, you will learn how to get started with Eclipse. Of course, if you prefer a different development environment, you can certainly use it with this book.

Get started by downloading Eclipse from `http://eclipse.org/downloads`. Versions exist for Linux, Mac OS X, and Windows. Run the installation program and pick the installation set called "Eclipse IDE for Java Developers".

Here are the steps to write a program with Eclipse.

1. After starting Eclipse, select File → New → Project from the menu.
2. Select "Java Project" from the wizard dialog (see Figure 2.5).
3. Click the Next button. *Uncheck* the "Use default location" checkbox. Click on Browse and navigate to the `corejava/v1ch02/Welcome` directory (Figure 2.6).

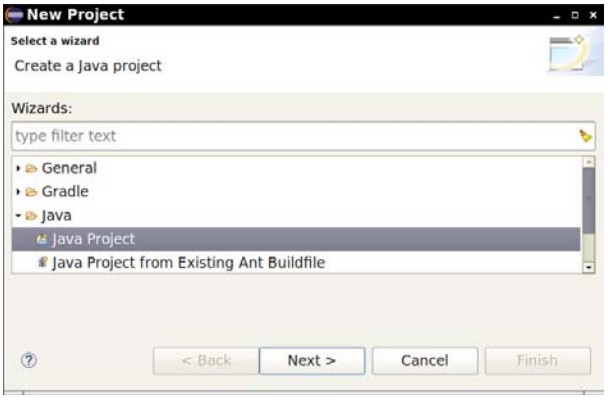

Figure 2.5 The New Project dialog in Eclipse

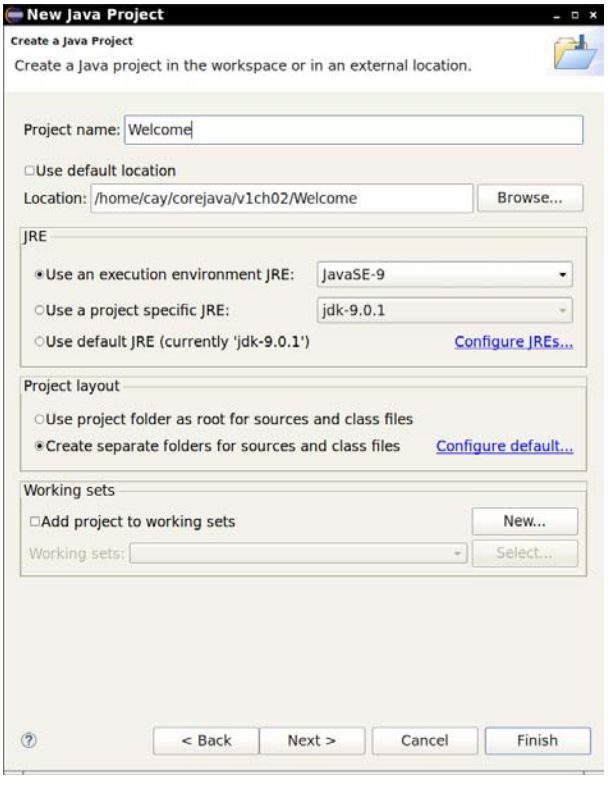

Figure 2.6 Configuring a project in Eclipse

4. Click the Finish button. The project is now created.
5. Click on the triangles in the left pane next to the project until you locate the file `Welcome.java`, and double-click on it. You should now see a pane with the program code (see Figure 2.7).

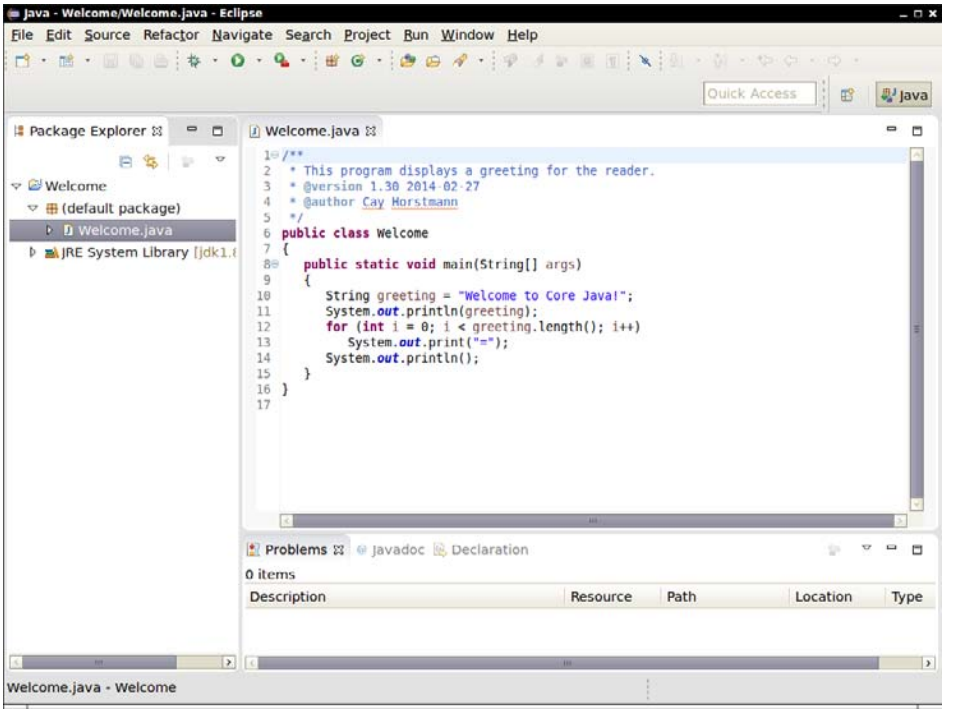

Figure 2.7 Editing a source file with Eclipse

6. With the right mouse button, click on the project name (Welcome) in the left pane. Select Run → Run As → Java Application. The program output is displayed in the console pane.

Presumably, this program does not have typos or bugs. (It was only a few lines of code, after all.) Let us suppose, for the sake of argument, that your code occasionally contains a typo (perhaps even a syntax error). Try it out—ruin your file, for example, by changing the capitalization of `String` as follows:

```
string greeting = "Welcome to Core Java!";
```

Note the wiggly line under `string`. In the tabs below the source code, click on Problems and expand the triangles until you see an error message that complains about an unknown `string` type (see Figure 2.8). Click on the error message. The cursor moves to the matching line in the edit pane, where you can correct your error. This allows you to fix your errors quickly.

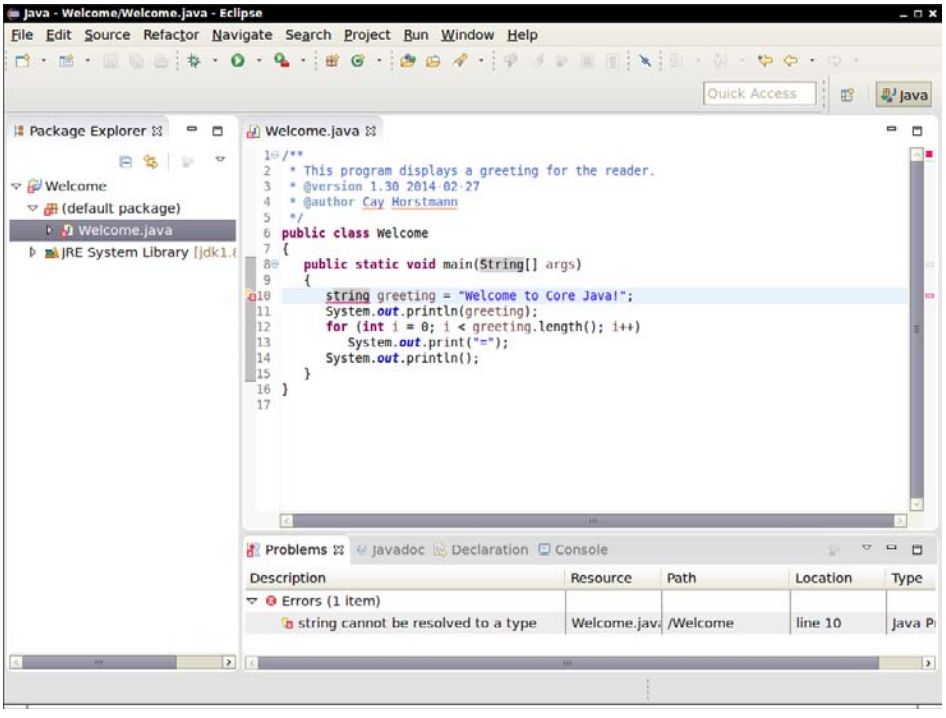

Figure 2.8 Error messages in Eclipse

TIP: Often, an Eclipse error report is accompanied by a lightbulb icon. Click on the lightbulb to get a list of suggested fixes.

2.4 JShell

In the preceding section, you saw how to compile and run a Java program. Java 9 introduces another way of working with Java. The JShell program provides a "read-evaluate-print loop," or REPL. You type a Java expression; JShell evaluates your input, prints the result, and waits for your next input. To start JShell, simply type `jshell` in a terminal window (see Figure 2.9).

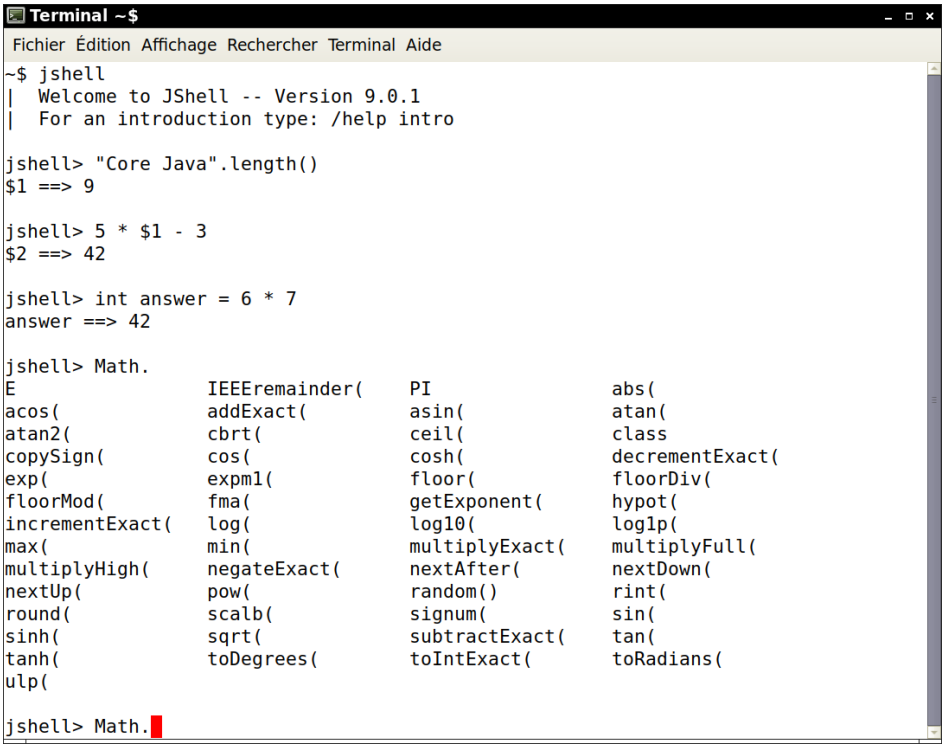

Figure 2.9 Running JShell

JShell starts with a greeting, followed by a prompt:

```
|  Welcome to JShell -- Version 11.0.1
|  For an introduction type: /help intro

jshell>
```

Now type an expression, such as

```
"Core Java".length()
```

JShell responds with the result—in this case, the number of characters in the string "Core Java".

```
$1 ==> 9
```

Note that you do *not* type `System.out.println`. JShell automatically prints the value of every expression that you enter.

The $1 in the output indicates that the result is available in further calculations. For example, if you type

```
5 * $1 - 3
```

the response is

```
$2 ==> 42
```

If you need a variable many times, you can give it a more memorable name. However, you have to follow the Java syntax and specify both the type and the name. (We will cover the syntax in Chapter 3.) For example,

```
jshell> int answer = 6 * 7
answer ==> 42
```

Another useful feature is tab completion. Type

```
Math.
```

followed by the Tab key. You get a list of all methods that you can invoke with the Math class:

```
jshell> Math.
E                IEEEremainder(   PI               abs(
acos(            addExact(        asin(            atan(
atan2(           cbrt(            ceil(            class
copySign(        cos(             cosh(            decrementExact(
exp(             expm1(           floor(           floorDiv(
floorMod(        fma(             getExponent(     hypot(
incrementExact(  log(             log10(           log1p(
max(             min(             multiplyExact(   multiplyFull(
multiplyHigh(    negateExact(     nextAfter(       nextDown(
nextUp(          pow(             random()         rint(
round(           scalb(           signum(          sin(
sinh(            sqrt(            subtractExact(   tan(
tanh(            toDegrees(       toIntExact(      toRadians(
ulp(
```

Now type l and hit the Tab key again. The method name is completed to log, and you get a shorter list:

```
jshell> Math.log
log(     log10(   log1p(
```

Now you can fill in the rest by hand:

```
jshell> Math.log10(0.001)
$3 ==> -3.0
```

To repeat a command, hit the ↑ key until you see the line that you want to reissue or edit. You can move the cursor in the line with the ← and → keys,

and add or delete characters. Hit Enter when you are done. For example, hit and replace `0.001` with `1000`, then hit Enter:

```
jshell> Math.log10(1000)
$4 ==> 3.0
```

JShell makes it easy and fun to learn about the Java language and library without having to launch a heavy-duty development environment and without fussing with `public static void main`.

In this chapter, you learned about the mechanics of compiling and running Java programs. You are now ready to move on to Chapter 3 where you will start learning the Java language.

This page intentionally left blank

CHAPTER 3

Fundamental Programming Structures in Java

In this chapter

At this point, we are assuming that you successfully installed the JDK and were able to run the sample programs that we showed you in Chapter 2.

It's time to start programming. This chapter shows you how the basic programming concepts such as data types, branches, and loops are implemented in Java.

3.1 A Simple Java Program

Let's look more closely at one of the simplest Java programs you can have—one that merely prints a message to console:

```
public class FirstSample
{
   public static void main(String[] args)
   {
      System.out.println("We will not use 'Hello, World!'");
   }
}
```

It is worth spending all the time you need to become comfortable with the framework of this sample; the pieces will recur in all applications. First and foremost, *Java is case sensitive*. If you made any mistakes in capitalization (such as typing `Main` instead of `main`), the program will not run.

Now let's look at this source code line by line. The keyword `public` is called an *access modifier*; these modifiers control the level of access other parts of a program have to this code. We'll have more to say about access modifiers in Chapter 5. The keyword `class` reminds you that everything in a Java program lives inside a class. Although we will spend a lot more time on classes in the next chapter, for now think of a class as a container for the program logic that defines the behavior of an application. As mentioned in Chapter 1, classes are the building blocks with which all Java applications and applets are built. *Everything* in a Java program must be inside a class.

Following the keyword `class` is the name of the class. The rules for class names in Java are quite generous. Names must begin with a letter, and after that, they can have any combination of letters and digits. The length is essentially unlimited. You cannot use a Java reserved word (such as `public` or `class`) for a class name. (See the appendix for a list of reserved words.)

The standard naming convention (which we follow in the name `FirstSample`) is that class names are nouns that start with an uppercase letter. If a name consists of multiple words, use an initial uppercase letter in each of the words. (This use of uppercase letters in the middle of a word is sometimes called "camel case" or, self-referentially, "CamelCase".

You need to make the file name for the source code the same as the name of the public class, with the extension .java appended. Thus, you must store this code in a file called FirstSample.java. (Again, case is important—don't use firstsample.java.)

If you have named the file correctly and not made any typos in the source code, then when you compile this source code, you end up with a file containing the bytecodes for this class. The Java compiler automatically names the bytecode file FirstSample.class and stores it in the same directory as the source file. Finally, launch the program by issuing the following command:

```
java FirstSample
```

(Remember to leave off the .class extension.) When the program executes, it simply displays the string We will not use 'Hello, World!' on the console.

When you use

```
java ClassName
```

to run a compiled program, the Java virtual machine always starts execution with the code in the main method in the class you indicate. (The term "method" is Java-speak for a function.) Thus, you *must* have a main method in the source of your class for your code to execute. You can, of course, add your own methods to a class and call them from the main method. (We cover writing your own methods in the next chapter.)

NOTE: According to the Java Language Specification, the main method must be declared public. (The Java Language Specification is the official document that describes the Java language. You can view or download it from http://docs.oracle.com/javase/specs.

However, several versions of the Java launcher were willing to execute Java programs even when the main method was not public. A programmer filed a bug report. To see it, visit http://bugs.java.com/bugdatabase/index.jsp and enter the Bug ID 4252539. In 1999, that bug was marked as "closed, will not be fixed." A Sun engineer added an explanation that the Java Virtual Machine Specification (at http://docs.oracle.com/javase/specs/jvms/se8/html) does not mandate that main is public and that "fixing it will cause potential troubles." Fortunately, sanity finally prevailed. The Java launcher in Java 1.4 and beyond enforces that the main method is public.

There are a couple of interesting aspects about this story. On the one hand, it is frustrating to have quality assurance engineers, who are often overworked and not always experts in the fine points of Java, make questionable decisions about bug reports. On the other hand, it is remarkable that Sun made the bug

reports and their resolutions available for anyone to scrutinize, long before Java was open source. At one point, Sun even let programmers vote for their most despised bugs and used the vote counts to decide which of them would get fixed in the next JDK release.

Notice the braces { } in the source code. In Java, as in C/C++, braces delineate the parts (usually called *blocks*) in your program. In Java, the code for any method must be started by an opening brace { and ended by a closing brace }.

Brace styles have inspired an inordinate amount of useless controversy. We follow a style that lines up matching braces. As whitespace is irrelevant to the Java compiler, you can use whatever brace style you like. We will have more to say about the use of braces when we talk about the various kinds of loops.

For now, don't worry about the keywords `static void`—just think of them as part of what you need to get a Java program to compile. By the end of Chapter 4, you will understand this incantation completely. The point to remember for now is that every Java application must have a `main` method that is declared in the following way:

```
public class ClassName
{
   public static void main(String[] args)
   {
      program statements
   }
}
```

C++ NOTE: As a C++ programmer, you know what a class is. Java classes are similar to C++ classes, but there are a few differences that can trap you. For example, in Java *all* functions are methods of some class. (The standard terminology refers to them as methods, not member functions.) Thus, in Java you must have a shell class for the `main` method. You may also be familiar with the idea of *static member functions* in C++. These are member functions defined inside a class that do not operate on objects. The `main` method in Java is always static. Finally, as in C/C++, the `void` keyword indicates that this method does not return a value. Unlike C/C++, the `main` method does not return an "exit code" to the operating system. If the `main` method exits normally, the Java program has the exit code `0`, indicating successful completion. To terminate the program with a different exit code, use the `System.exit` method.

Next, turn your attention to this fragment:

```
{
   System.out.println("We will not use 'Hello, World!'");
}
```

Braces mark the beginning and end of the *body* of the method. This method has only one statement in it. As with most programming languages, you can think of Java statements as sentences of the language. In Java, every statement must end with a semicolon. In particular, carriage returns do not mark the end of a statement, so statements can span multiple lines if need be.

The body of the `main` method contains a statement that outputs a single line of text to the console.

Here, we are using the `System.out` object and calling its `println` method. Notice the periods used to invoke a method. Java uses the general syntax

object.*method*(*parameters*)

as its equivalent of a function call.

In this case, we are calling the `println` method and passing it a string parameter. The method displays the string parameter on the console. It then terminates the output line, so that each call to `println` displays its output on a new line. Notice that Java, like C/C++, uses double quotes to delimit strings. (You can find more information about strings later in this chapter.)

Methods in Java, like functions in any programming language, can use zero, one, or more *parameters* (some programmers call them *arguments*). Even if a method takes no parameters, you must still use empty parentheses. For example, a variant of the `println` method with no parameters just prints a blank line. You invoke it with the call

```
System.out.println();
```

NOTE: `System.out` also has a `print` method that doesn't add a newline character to the output. For example, `System.out.print("Hello")` prints `Hello` without a newline. The next output appears immediately after the letter `o`.

3.2 Comments

Comments in Java, as in most programming languages, do not show up in the executable program. Thus, you can add as many comments as needed without fear of bloating the code. Java has three ways of marking comments.

The most common form is a //. Use this for a comment that runs from the // to the end of the line.

```
System.out.println("We will not use 'Hello, World!'"); // is this too cute?
```

When longer comments are needed, you can mark each line with a //, or you can use the /* and */ comment delimiters that let you block off a longer comment.

Finally, a third kind of comment is used to generate documentation automatically. This comment uses a /** to start and a */ to end. You can see this type of comment in Listing 3.1. For more on this type of comment and on automatic documentation generation, see Chapter 4.

Listing 3.1 FirstSample/FirstSample.java

```
1  /**
2   * This is the first sample program in Core Java Chapter 3
3   * @version 1.01 1997-03-22
4   * @author Gary Cornell
5   */
6  public class FirstSample
7  {
8     public static void main(String[] args)
9     {
10       System.out.println("We will not use 'Hello, World!'");
11    }
12 }
```

CAUTION: /* */ comments do not nest in Java. That is, you might not be able to deactivate code simply by surrounding it with /* and */ because the code you want to deactivate might itself contain a */ delimiter.

3.3 Data Types

Java is a *strongly typed language*. This means that every variable must have a declared type. There are eight *primitive types* in Java. Four of them are integer types; two are floating-point number types; one is the character type `char`, used for code units in the Unicode encoding scheme (see Section 3.3.3, "The `char` Type," on p. 46); and one is a `boolean` type for truth values.

NOTE: Java has an arbitrary-precision arithmetic package. However, "big numbers," as they are called, are Java *objects* and not a primitive Java type. You will see how to use them later in this chapter.

3.3.1 Integer Types

The integer types are for numbers without fractional parts. Negative values are allowed. Java provides the four integer types shown in Table 3.1.

Table 3.1 Java Integer Types

Type	Storage Requirement	Range (Inclusive)
`int`	4 bytes	–2,147,483,648 to 2,147,483,647 (just over 2 billion)
`short`	2 bytes	–32,768 to 32,767
`long`	8 bytes	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`byte`	1 byte	–128 to 127

In most situations, the `int` type is the most practical. If you want to represent the number of inhabitants of our planet, you'll need to resort to a `long`. The `byte` and `short` types are mainly intended for specialized applications, such as low-level file handling, or for large arrays when storage space is at a premium.

Under Java, the ranges of the integer types do not depend on the machine on which you will be running the Java code. This alleviates a major pain for the programmer who wants to move software from one platform to another, or even between operating systems on the same platform. In contrast, C and C++ programs use the most efficient integer type for each processor. As a result, a C program that runs well on a 32-bit processor may exhibit integer overflow on a 16-bit system. Since Java programs must run with the same results on all machines, the ranges for the various types are fixed.

Long integer numbers have a suffix `L` or `l` (for example, `4000000000L`). Hexadecimal numbers have a prefix `0x` or `0X` (for example, `0xCAFE`). Octal numbers have a prefix `0` (for example, `010` is 8)—naturally, this can be confusing, so we recommend against the use of octal constants.

Starting with Java 7, you can write numbers in binary, with a prefix `0b` or `0B`. For example, `0b1001` is 9. Also starting with Java 7, you can add underscores to number literals, such as `1_000_000` (or `0b1111_0100_0010_0100_0000`) to denote one million. The underscores are for human eyes only. The Java compiler simply removes them.

C++ NOTE: In C and C++, the sizes of types such as `int` and `long` depend on the target platform. On a 16-bit processor such as the 8086, integers are 2 bytes, but on a 32-bit processor like a Pentium or SPARC they are 4-byte quantities. Similarly, `long` values are 4-byte on 32-bit processors and 8-byte on 64-bit processors. These differences make it challenging to write cross-platform programs. In Java, the sizes of all numeric types are platform-independent.

Note that Java does not have any `unsigned` versions of the `int`, `long`, `short`, or `byte` types.

NOTE: If you work with integer values that can never be negative and you really need an additional bit, you can, with some care, interpret signed integer values as unsigned. For example, instead of having a `byte` value `b` represent the range from –128 to 127, you may want a range from 0 to 255. You can store it in a `byte`. Due to the nature of binary arithmetic, addition, subtraction, and multiplication will work provided they don't overflow. For other operations, call `Byte.toUnsignedInt(b)` to get an `int` value between 0 and 255, then process the integer value and cast back to `byte`. The `Integer` and `Long` classes have methods for unsigned division and remainder.

3.3.2 Floating-Point Types

The floating-point types denote numbers with fractional parts. The two floating-point types are shown in Table 3.2.

Table 3.2 Floating-Point Types

Type	Storage Requirement	Range
`float`	4 bytes	Approximately ±3.40282347E+38F (6–7 significant decimal digits)
`double`	8 bytes	Approximately ±1.79769313486231570E+308 (15 significant decimal digits)

The name `double` refers to the fact that these numbers have twice the precision of the `float` type. (Some people call these *double-precision* numbers.) The limited precision of `float` (6–7 significant digits) is simply not sufficient for many situations. Use `float` values only when you work with a library that requires them, or when you need to store a very large number of them.

Numbers of type `float` have a suffix `F` or `f` (for example, `3.14F`). Floating-point numbers without an `F` suffix (such as `3.14`) are always considered to be of type `double`. You can optionally supply the `D` or `d` suffix (for example, `3.14D`).

NOTE: You can specify floating-point literals in hexadecimal. For example, $0.125 = 2^{-3}$ can be written as `0x1.0p-3`. In hexadecimal notation, you use a `p`, not an `e`, to denote the exponent. (An `e` is a hexadecimal digit.) Note that the mantissa is written in hexadecimal and the exponent in decimal. The base of the exponent is 2, not 10.

All floating-point computations follow the IEEE 754 specification. In particular, there are three special floating-point values to denote overflows and errors:

- Positive infinity
- Negative infinity
- NaN (not a number)

For example, the result of dividing a positive number by 0 is positive infinity. Computing 0/0 or the square root of a negative number yields NaN.

NOTE: The constants `Double.POSITIVE_INFINITY`, `Double.NEGATIVE_INFINITY`, and `Double.NaN` (as well as corresponding `Float` constants) represent these special values, but they are rarely used in practice. In particular, you cannot test

```
if (x == Double.NaN) // is never true
```

to check whether a particular result equals `Double.NaN`. All "not a number" values are considered distinct. However, you can use the `Double.isNaN` method:

```
if (Double.isNaN(x)) // check whether x is "not a number"
```

CAUTION: Floating-point numbers are *not* suitable for financial calculations in which roundoff errors cannot be tolerated. For example, the command `System.out.println(2.0 - 1.1)` prints `0.8999999999999999`, not `0.9` as you would expect. Such roundoff errors are caused by the fact that floating-point numbers are represented in the binary number system. There is no precise binary representation of the fraction 1/10, just as there is no accurate representation of the fraction 1/3 in the decimal system. If you need precise numerical computations without roundoff errors, use the `BigDecimal` class, which is introduced later in this chapter.

3.3.3 The char Type

The `char` type was originally intended to describe individual characters. However, this is no longer the case. Nowadays, some Unicode characters can be described with one `char` value, and other Unicode characters require two `char` values. Read the next section for the gory details.

Literal values of type `char` are enclosed in single quotes. For example, `'A'` is a character constant with value 65. It is different from `"A"`, a string containing a single character. Values of type `char` can be expressed as hexadecimal values that run from `\u0000` to `\uFFFF`. For example, `\u2122` is the trademark symbol (™) and `\u03C0` is the Greek letter pi (π).

Besides the `\u` escape sequences, there are several escape sequences for special characters, as shown in Table 3.3. You can use these escape sequences inside quoted character literals and strings, such as `'\u2122'` or `"Hello\n"`. The `\u` escape sequence (but none of the other escape sequences) can even be used *outside* quoted character constants and strings. For example,

```
public static void main(String\u005B\u005D args)
```

is perfectly legal—`\u005B` and `\u005D` are the encodings for `[` and `]`.

Table 3.3 Escape Sequences for Special Characters

Escape Sequence	Name	Unicode Value
`\b`	Backspace	`\u0008`
`\t`	Tab	`\u0009`
`\n`	Linefeed	`\u000a`
`\r`	Carriage return	`\u000d`
`\"`	Double quote	`\u0022`
`\'`	Single quote	`\u0027`
`\\`	Backslash	`\u005c`

CAUTION: Unicode escape sequences are processed before the code is parsed. For example, `"\u0022+\u0022"` is *not* a string consisting of a plus sign surrounded by quotation marks (U+0022). Instead, the `\u0022` are converted into `"` before parsing, yielding `""+""`, or an empty string.

Even more insidiously, you must beware of `\u` inside comments. The comment

```
// \u000A is a newline
```

yields a syntax error since `\u000A` is replaced with a newline when the program is read. Similarly, a comment

```
// look inside c:\users
```

yields a syntax error because the `\u` is not followed by four hex digits.

3.3.4 Unicode and the char Type

To fully understand the `char` type, you have to know about the Unicode encoding scheme. Unicode was invented to overcome the limitations of traditional character encoding schemes. Before Unicode, there were many different standards: ASCII in the United States, ISO 8859-1 for Western European languages, KOI-8 for Russian, GB18030 and BIG-5 for Chinese, and so on. This caused two problems. First, a particular code value corresponds to different letters in the different encoding schemes. Second, the encodings for languages with large character sets have variable length: Some common characters are encoded as single bytes, others require two or more bytes.

Unicode was designed to solve these problems. When the unification effort started in the 1980s, a fixed 2-byte code was more than sufficient to encode all characters used in all languages in the world, with room to spare for future expansion—or so everyone thought at the time. In 1991, Unicode 1.0 was released, using slightly less than half of the available 65,536 code values. Java was designed from the ground up to use 16-bit Unicode characters, which was a major advance over other programming languages that used 8-bit characters.

Unfortunately, over time, the inevitable happened. Unicode grew beyond 65,536 characters, primarily due to the addition of a very large set of ideographs used for Chinese, Japanese, and Korean. Now, the 16-bit `char` type is insufficient to describe all Unicode characters.

We need a bit of terminology to explain how this problem is resolved in Java, beginning with Java 5. A *code point* is a code value that is associated with a character in an encoding scheme. In the Unicode standard, code points are written in hexadecimal and prefixed with `U+`, such as `U+0041` for the code point of the Latin letter A. Unicode has code points that are grouped into 17 *code planes*. The first code plane, called the *basic multilingual plane*, consists of the "classic" Unicode characters with code points `U+0000` to `U+FFFF`. Sixteen additional planes, with code points `U+10000` to `U+10FFFF`, hold the *supplementary characters*.

The UTF-16 encoding represents all Unicode code points in a variable-length code. The characters in the basic multilingual plane are represented as 16-bit values, called *code units*. The supplementary characters are encoded as consecutive pairs of code units. Each of the values in such an encoding pair falls

into a range of 2048 unused values of the basic multilingual plane, called the *surrogates area* (U+D800 to U+DBFF for the first code unit, U+DC00 to U+DFFF for the second code unit). This is rather clever, because you can immediately tell whether a code unit encodes a single character or it is the first or second part of a supplementary character. For example, 𝕆 (the mathematical symbol for the set of octonions, http://math.ucr.edu/home/baez/octonions) has code point U+1D546 and is encoded by the two code units U+D835 and U+DD46. (See https://tools.ietf.org/html/rfc2781 for a description of the encoding algorithm.)

In Java, the char type describes a *code unit* in the UTF-16 encoding.

Our strong recommendation is not to use the char type in your programs unless you are actually manipulating UTF-16 code units. You are almost always better off treating strings (which we will discuss in Section 3.6, "Strings," on p. 62) as abstract data types.

3.3.5 The boolean Type

The boolean type has two values, false and true. It is used for evaluating logical conditions. You cannot convert between integers and boolean values.

C++ NOTE: In C++, numbers and even pointers can be used in place of boolean values. The value 0 is equivalent to the bool value false, and a nonzero value is equivalent to true. This is *not* the case in Java. Thus, Java programmers are shielded from accidents such as

```
if (x = 0) // oops... meant x == 0
```

In C++, this test compiles and runs, always evaluating to false. In Java, the test does not compile because the integer expression x = 0 cannot be converted to a boolean value.

3.4 Variables and Constants

As in every programming language, variables are used to store values. Constants are variables whose values don't change. In the following sections, you will learn how to declare variables and constants.

3.4.1 Declaring Variables

In Java, every variable has a *type*. You declare a variable by placing the type first, followed by the name of the variable. Here are some examples:

```
double salary;
int vacationDays;
long earthPopulation;
boolean done;
```

Notice the semicolon at the end of each declaration. The semicolon is necessary because a declaration is a complete Java statement, and all Java statements end in semicolons.

A variable name must begin with a letter and must be a sequence of letters or digits. Note that the terms "letter" and "digit" are much broader in Java than in most languages. A letter is defined as `'A'–'Z'`, `'a'–'z'`, `'_'`, `'$'`, or *any* Unicode character that denotes a letter in a language. For example, German users can use umlauts such as `'ä'` in variable names; Greek speakers could use a π. Similarly, digits are `'0'–'9'` and *any* Unicode characters that denote a digit in a language. Symbols like `'+'` or `'©'` cannot be used inside variable names, nor can spaces. *All* characters in the name of a variable are significant and *case is also significant.* The length of a variable name is essentially unlimited.

TIP: If you are really curious as to what Unicode characters are "letters" as far as Java is concerned, you can use the `isJavaIdentifierStart` and `isJavaIdentifierPart` methods in the `Character` class to check.

TIP: Even though `$` is a valid Java letter, you should not use it in your own code. It is intended for names that are generated by the Java compiler and other tools.

You also cannot use a Java reserved word as a variable name.

As of Java 9, a single underscore _ cannot be used as a variable name. A future version of Java may use _ as a wildcard symbol.

You can declare multiple variables on a single line:

```
int i, j; // both are integers
```

However, we don't recommend this style. If you declare each variable separately, your programs are easier to read.

NOTE: As you saw, names are case sensitive, for example, `hireday` and `hireDay` are two separate names. In general, you should not have two names that only differ in their letter case. However, sometimes it is difficult to come up with a

good name for a variable. Many programmers then give the variable the same name as the type, for example

```
Box box; // "Box" is the type and "box" is the variable name
```

Other programmers prefer to use an "a" prefix for the variable:

```
Box aBox;
```

3.4.2 Initializing Variables

After you declare a variable, you must explicitly initialize it by means of an assignment statement—you can never use the value of an uninitialized variable. For example, the Java compiler flags the following sequence of statements as an error:

```
int vacationDays;
System.out.println(vacationDays); // ERROR--variable not initialized
```

You assign to a previously declared variable by using the variable name on the left, an equal sign (=), and then some Java expression with an appropriate value on the right.

```
int vacationDays;
vacationDays = 12;
```

You can both declare and initialize a variable on the same line. For example:

```
int vacationDays = 12;
```

Finally, in Java you can put declarations anywhere in your code. For example, the following is valid code in Java:

```
double salary = 65000.0;
System.out.println(salary);
int vacationDays = 12; // OK to declare a variable here
```

In Java, it is considered good style to declare variables as closely as possible to the point where they are first used.

NOTE: Starting with Java 10, you do not need to declare the types of local variables if they can be inferred from the initial value. Simply use the keyword `var` instead of the type:

```
var vacationDays = 12; // vacationDays is an int
var greeting = "Hello"; // greeting is a String
```

We will start using this feature in the next chapter.

C++ NOTE: C and C++ distinguish between the *declaration* and *definition* of a variable. For example,

```
int i = 10;
```

is a definition, whereas

```
extern int i;
```

is a declaration. In Java, no declarations are separate from definitions.

3.4.3 Constants

In Java, you use the keyword `final` to denote a constant. For example:

```
public class Constants
{
   public static void main(String[] args)
   {
      final double CM_PER_INCH = 2.54;
      double paperWidth = 8.5;
      double paperHeight = 11;
      System.out.println("Paper size in centimeters: "
         + paperWidth * CM_PER_INCH + " by " + paperHeight * CM_PER_INCH);
   }
}
```

The keyword `final` indicates that you can assign to the variable once, and then its value is set once and for all. It is customary to name constants in all uppercase.

It is probably more common in Java to create a constant so it's available to multiple methods inside a single class. These are usually called *class constants*. Set up a class constant with the keywords `static final`. Here is an example of using a class constant:

```
public class Constants2
{
    public static final double CM_PER_INCH = 2.54;

    public static void main(String[] args)
    {
       double paperWidth = 8.5;
       double paperHeight = 11;
       System.out.println("Paper size in centimeters: "
          + paperWidth * CM_PER_INCH + " by " + paperHeight * CM_PER_INCH);
    }
}
```

Note that the definition of the class constant appears *outside* the `main` method. Thus, the constant can also be used in other methods of the same class. Furthermore, if the constant is declared, as in our example, `public`, methods of other classes can also use it—in our example, as `Constants2.CM_PER_INCH`.

C++ NOTE: `const` is a reserved Java keyword, but it is not currently used for anything. You must use `final` for a constant.

3.4.4 Enumerated Types

Sometimes, a variable should only hold a restricted set of values. For example, you may sell clothes or pizza in four sizes: small, medium, large, and extra large. Of course, you could encode these sizes as integers `1`, `2`, `3`, `4` or characters `S`, `M`, `L`, and `X`. But that is an error-prone setup. It is too easy for a variable to hold a wrong value (such as `0` or `m`).

You can define your own *enumerated type* whenever such a situation arises. An enumerated type has a finite number of named values. For example,

```
enum Size { SMALL, MEDIUM, LARGE, EXTRA_LARGE };
```

Now you can declare variables of this type:

```
Size s = Size.MEDIUM;
```

A variable of type `Size` can hold only one of the values listed in the type declaration, or the special value `null` that indicates that the variable is not set to any value at all.

We discuss enumerated types in greater detail in Chapter 5.

3.5 Operators

Operators are used to combine values. As you will see in the following sections, Java has a rich set of arithmetic and logical operators and mathematical functions.

3.5.1 Arithmetic Operators

The usual arithmetic operators `+`, `-`, `*`, `/` are used in Java for addition, subtraction, multiplication, and division. The `/` operator denotes integer division if both arguments are integers, and floating-point division otherwise. Integer remainder (sometimes called *modulus*) is denoted by `%`. For example, `15 / 2` is `7`, `15 % 2` is `1`, and `15.0 / 2` is `7.5`.

Note that integer division by 0 raises an exception, whereas floating-point division by 0 yields an infinite or NaN result.

NOTE: One of the stated goals of the Java programming language is portability. A computation should yield the same results no matter which virtual machine executes it. For arithmetic computations with floating-point numbers, it is surprisingly difficult to achieve this portability. The `double` type uses 64 bits to store a numeric value, but some processors use 80-bit floating-point registers. These registers yield added precision in intermediate steps of a computation. For example, consider the following computation:

```
double w = x * y / z;
```

Many Intel processors compute `x * y`, leave the result in an 80-bit register, then divide by `z`, and finally truncate the result back to 64 bits. That can yield a more accurate result, and it can avoid exponent overflow. But the result may be *different* from a computation that uses 64 bits throughout. For that reason, the initial specification of the Java virtual machine mandated that all intermediate computations must be truncated. The numeric community hated it. Not only can the truncated computations cause overflow, they are actually *slower* than the more precise computations because the truncation operations take time. For that reason, the Java programming language was updated to recognize the conflicting demands for optimum performance and perfect reproducibility. By default, virtual machine designers are now permitted to use extended precision for intermediate computations. However, methods tagged with the `strictfp` keyword must use strict floating-point operations that yield reproducible results.

For example, you can tag `main` as

```
public static strictfp void main(String[] args)
```

Then all instructions inside the `main` method will use strict floating-point computations. If you tag a class as `strictfp`, then all of its methods must use strict floating-point computations.

The gory details are very much tied to the behavior of the Intel processors. In the default mode, intermediate results are allowed to use an extended exponent, but not an extended mantissa. (The Intel chips support truncation of the mantissa without loss of performance.) Therefore, the only difference between the default and strict modes is that strict computations may overflow when default computations don't.

If your eyes glazed over when reading this note, don't worry. Floating-point overflow isn't a problem that one encounters for most common programs. We don't use the `strictfp` keyword in this book.

3.5.2 Mathematical Functions and Constants

The `Math` class contains an assortment of mathematical functions that you may occasionally need, depending on the kind of programming that you do.

To take the square root of a number, use the `sqrt` method:

```
double x = 4;
double y = Math.sqrt(x);
System.out.println(y); // prints 2.0
```

NOTE: There is a subtle difference between the `println` method and the `sqrt` method. The `println` method operates on the `System.out` object. But the `sqrt` method in the `Math` class does not operate on any object. Such a method is called a *static* method. You can learn more about static methods in Chapter 4.

The Java programming language has no operator for raising a quantity to a power: You must use the `pow` method in the `Math` class. The statement

```
double y = Math.pow(x, a);
```

sets `y` to be `x` raised to the power `a` (x^a). The `pow` method's parameters are both of type `double`, and it returns a `double` as well.

The `floorMod` method aims to solve a long-standing problem with integer remainders. Consider the expression `n % 2`. Everyone knows that this is `0` if `n` is even and `1` if `n` is odd. Except, of course, when `n` is odd and negative. Then it is `-1`. Why? When the first computers were built, someone had to make rules for how integer division and remainder should work for negative operands. Mathematicians had known the optimal (or "Euclidean") rule for a few hundred years: always leave the remainder ≥ 0. But, rather than open a math textbook, those pioneers came up with rules that seemed reasonable but are actually inconvenient.

Consider this problem. You compute the position of the hour hand of a clock. An adjustment is applied, and you want to normalize to a number between `0` and `11`. That is easy: `(position + adjustment) % 12`. But what if the adjustment is negative? Then you might get a negative number. So you have to introduce a branch, or use `((position + adjustment) % 12 + 12) % 12`. Either way, it is a hassle.

The `floorMod` method makes it easier: `floorMod(position + adjustment, 12)` always yields a value between `0` and `11`. (Unfortunately, `floorMod` gives negative results for negative divisors, but that situation doesn't often occur in practice.)

The `Math` class supplies the usual trigonometric functions:

```
Math.sin
Math.cos
Math.tan
Math.atan
Math.atan2
```

and the exponential function with its inverse, the natural logarithm, as well as the decimal logarithm:

```
Math.exp
Math.log
Math.log10
```

Finally, two constants denote the closest possible approximations to the mathematical constants π and e:

```
Math.PI
Math.E
```

TIP: You can avoid the `Math` prefix for the mathematical methods and constants by adding the following line to the top of your source file:

```
import static java.lang.Math.*;
```

For example:

```
System.out.println("The square root of \u03C0 is " + sqrt(PI));
```

We discuss static imports in Chapter 4.

NOTE: The methods in the `Math` class use the routines in the computer's floating-point unit for fastest performance. If completely predictable results are more important than performance, use the `StrictMath` class instead. It implements the algorithms from the "Freely Distributable Math Library" (`www.netlib.org/fdlibm`), guaranteeing identical results on all platforms.

NOTE: The `Math` class provides several methods to make integer arithmetic safer. The mathematical operators quietly return wrong results when a computation overflows. For example, one billion times three (`1000000000 * 3`) evaluates to `-1294967296` because the largest `int` value is just over two billion. If you call `Math.multiplyExact(1000000000, 3)` instead, an exception is generated. You can catch that exception or let the program terminate rather than quietly continue with a wrong result. There are also methods `addExact`, `subtractExact`, `incrementExact`, `decrementExact`, `negateExact`, all with `int` and `long` parameters.

3.5.3 Conversions between Numeric Types

It is often necessary to convert from one numeric type to another. Figure 3.1 shows the legal conversions.

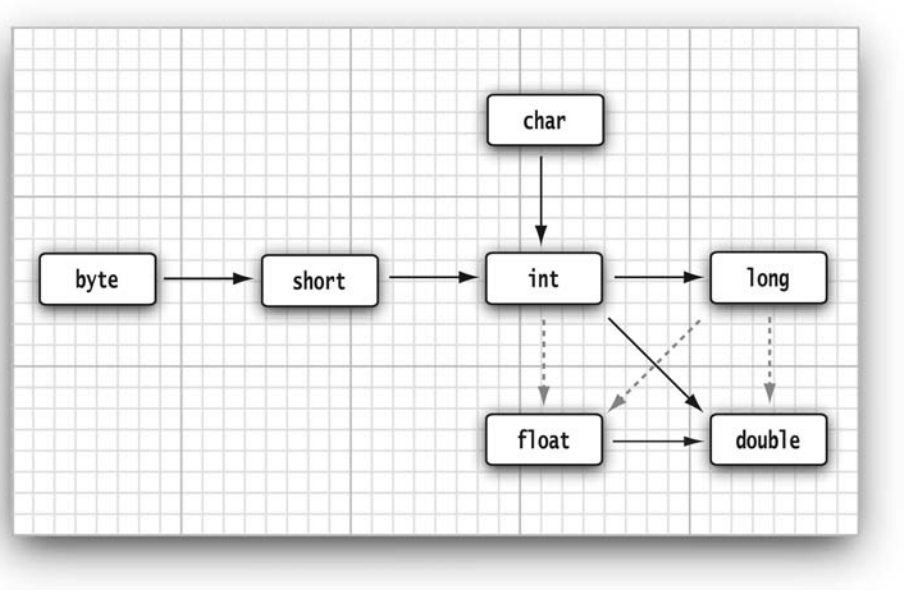

Figure 3.1 Legal conversions between numeric types

The six solid arrows in Figure 3.1 denote conversions without information loss. The three dotted arrows denote conversions that may lose precision. For example, a large integer such as `123456789` has more digits than the `float` type can represent. When the integer is converted to a `float`, the resulting value has the correct magnitude but loses some precision.

```
int n = 123456789;
float f = n; // f is 1.23456792E8
```

When two values are combined with a binary operator (such as `n + f` where `n` is an integer and `f` is a floating-point value), both operands are converted to a common type before the operation is carried out.

- If either of the operands is of type `double`, the other one will be converted to a `double`.
- Otherwise, if either of the operands is of type `float`, the other one will be converted to a `float`.

- Otherwise, if either of the operands is of type `long`, the other one will be converted to a `long`.
- Otherwise, both operands will be converted to an `int`.

3.5.4 Casts

In the preceding section, you saw that `int` values are automatically converted to `double` values when necessary. On the other hand, there are obviously times when you want to consider a `double` as an integer. Numeric conversions are possible in Java, but of course information may be lost. Conversions in which loss of information is possible are done by means of *casts*. The syntax for casting is to give the target type in parentheses, followed by the variable name. For example:

```
double x = 9.997;
int nx = (int) x;
```

Now, the variable `nx` has the value `9` because casting a floating-point value to an integer discards the fractional part.

If you want to *round* a floating-point number to the *nearest* integer (which in most cases is a more useful operation), use the `Math.round` method:

```
double x = 9.997;
int nx = (int) Math.round(x);
```

Now the variable `nx` has the value `10`. You still need to use the cast `(int)` when you call `round`. The reason is that the return value of the `round` method is a `long`, and a `long` can only be assigned to an `int` with an explicit cast because there is the possibility of information loss.

CAUTION: If you try to cast a number of one type to another that is out of range for the target type, the result will be a truncated number that has a different value. For example, `(byte) 300` is actually `44`.

C++ NOTE: You cannot cast between `boolean` values and any numeric type. This convention prevents common errors. In the rare case when you want to convert a `boolean` value to a number, you can use a conditional expression such as `b ? 1 : 0`.

3.5.5 Combining Assignment with Operators

There is a convenient shortcut for using binary operators in an assignment. For example,

```
x += 4;
```

is equivalent to

```
x = x + 4;
```

(In general, place the operator to the left of the = sign, such as *= or %=.)

NOTE: If the operator yields a value whose type is different from that of the left-hand side, then it is coerced to fit. For example, if x is an int, then the statement

```
x += 3.5;
```

is valid, setting x to (int)(x + 3.5).

3.5.6 Increment and Decrement Operators

Programmers, of course, know that one of the most common operations with a numeric variable is to add or subtract 1. Java, following in the footsteps of C and C++, has both increment and decrement operators: n++ adds 1 to the current value of the variable n, and n-- subtracts 1 from it. For example, the code

```
int n = 12;
n++;
```

changes n to 13. Since these operators change the value of a variable, they cannot be applied to numbers themselves. For example, 4++ is not a legal statement.

There are two forms of these operators; you've just seen the postfix form of the operator that is placed after the operand. There is also a prefix form, ++n. Both change the value of the variable by 1. The difference between the two appears only when they are used inside expressions. The prefix form does the addition first; the postfix form evaluates to the old value of the variable.

```
int m = 7;
int n = 7;
int a = 2 * ++m; // now a is 16, m is 8
int b = 2 * n++; // now b is 14, n is 8
```

We recommend against using ++ inside expressions because this often leads to confusing code and annoying bugs.

3.5.7 Relational and boolean Operators

Java has the full complement of relational operators. To test for equality, use a double equal sign, ==. For example, the value of

```
3 == 7
```

is false.

Use a != for inequality. For example, the value of

```
3 != 7
```

is true.

Finally, you have the usual < (less than), > (greater than), <= (less than or equal), and >= (greater than or equal) operators.

Java, following C++, uses && for the logical "and" operator and || for the logical "or" operator. As you can easily remember from the != operator, the exclamation point ! is the logical negation operator. The && and || operators are evaluated in "short circuit" fashion: The second argument is not evaluated if the first argument already determines the value. If you combine two expressions with the && operator,

expression$_1$ **&&** *expression*$_2$

and the truth value of the first expression has been determined to be false, then it is impossible for the result to be true. Thus, the value for the second expression is *not* calculated. This behavior can be exploited to avoid errors. For example, in the expression

```
x != 0 && 1 / x > x + y // no division by 0
```

the second part is never evaluated if x equals zero. Thus, 1 / x is not computed if x is zero, and no divide-by-zero error can occur.

Similarly, the value of *expression*$_1$ || *expression*$_2$ is automatically true if the first expression is true, without evaluating the second expression.

Finally, Java supports the ternary ?: operator that is occasionally useful. The expression

condition ? *expression*$_1$: *expression*$_2$

evaluates to the first expression if the condition is true, to the second expression otherwise. For example,

```
x < y ? x : y
```

gives the smaller of x and y.

3.5.8 Bitwise Operators

For any of the integer types, you have operators that can work directly with the bits that make up the integers. This means that you can use masking techniques to get at individual bits in a number. The bitwise operators are

`&` *("and")* `|` *("or")* `^` *("xor")* `~` *("not")*

These operators work on bit patterns. For example, if `n` is an integer variable, then

```
int fourthBitFromRight = (n & 0b1000) / 0b1000;
```

gives you a `1` if the fourth bit from the right in the binary representation of `n` is `1`, and `0` otherwise. Using `&` with the appropriate power of 2 lets you mask out all but a single bit.

NOTE: When applied to `boolean` values, the `&` and `|` operators yield a `boolean` value. These operators are similar to the `&&` and `||` operators, except that the `&` and `|` operators are not evaluated in "short circuit" fashion—that is, both arguments are evaluated before the result is computed.

There are also `>>` and `<<` operators which shift a bit pattern right or left. These operators are convenient when you need to build up bit patterns to do bit masking:

```
int fourthBitFromRight = (n & (1 << 3)) >> 3;
```

Finally, a `>>>` operator fills the top bits with zero, unlike `>>` which extends the sign bit into the top bits. There is no `<<<` operator.

CAUTION: The right-hand argument of the shift operators is reduced modulo 32 (unless the left-hand argument is a `long`, in which case the right-hand argument is reduced modulo 64). For example, the value of `1 << 35` is the same as `1 << 3` or `8`.

C++ NOTE: In C/C++, there is no guarantee as to whether `>>` performs an arithmetic shift (extending the sign bit) or a logical shift (filling in with zeroes). Implementors are free to choose whichever is more efficient. That means the C/C++ `>>` operator may yield implementation-dependent results for negative numbers. Java removes that uncertainty.

3.5.9 Parentheses and Operator Hierarchy

Table 3.4 shows the precedence of operators. If no parentheses are used, operations are performed in the hierarchical order indicated. Operators on the same level are processed from left to right, except for those that are right-associative, as indicated in the table. For example, && has a higher precedence than ||, so the expression

```
a && b || c
```

means

```
(a && b) || c
```

Table 3.4 Operator Precedence

Operators	Associativity
[] . () (method call)	Left to right
! ~ ++ -- + (unary) - (unary) () (cast) new	Right to left
* / %	Left to right
+ -	Left to right
<< >> >>>	Left to right
< <= > >= instanceof	Left to right
== !=	Left to right
&	Left to right
^	Left to right
|	Left to right
&&	Left to right
||	Left to right
?:	Right to left
= += -= *= /= %= &= |= ^= <<= >>= >>>=	Right to left

Since += associates right to left, the expression

```
a += b += c
```

means

```
a += (b += c)
```

That is, the value of `b += c` (which is the value of `b` after the addition) is added to `a`.

C++ NOTE: Unlike C or C++, Java does not have a comma operator. However, you can use a *comma-separated list of expressions* in the first and third slot of a `for` statement.

3.6 Strings

Conceptually, Java strings are sequences of Unicode characters. For example, the string `"Java\u2122"` consists of the five Unicode characters J, a, v, a, and ™. Java does not have a built-in string type. Instead, the standard Java library contains a predefined class called, naturally enough, `String`. Each quoted string is an instance of the `String` class:

```
String e = ""; // an empty string
String greeting = "Hello";
```

3.6.1 Substrings

You can extract a substring from a larger string with the `substring` method of the `String` class. For example,

```
String greeting = "Hello";
String s = greeting.substring(0, 3);
```

creates a string consisting of the characters `"Hel"`.

NOTE: Like C and C++, Java counts code units and code points in strings starting with `0`.

The second parameter of `substring` is the first position that you *do not* want to copy. In our case, we want to copy positions 0, 1, and 2 (from position 0 to position 2 inclusive). As `substring` counts it, this means from position 0 inclusive to position 3 *exclusive*.

There is one advantage to the way `substring` works: Computing the length of the substring is easy. The string `s.substring(a, b)` always has length `b – a`. For example, the substring `"Hel"` has length 3 – 0 = 3.

3.6.2 Concatenation

Java, like most programming languages, allows you to use + to join (concatenate) two strings.

```
String expletive = "Expletive";
String PG13 = "deleted";
String message = expletive + PG13;
```

The preceding code sets the variable `message` to the string `"Expletivedeleted"`. (Note the lack of a space between the words: The + operator joins two strings in the order received, *exactly* as they are given.)

When you concatenate a string with a value that is not a string, the latter is converted to a string. (As you will see in Chapter 5, every Java object can be converted to a string.) For example,

```
int age = 13;
String rating = "PG" + age;
```

sets `rating` to the string `"PG13"`.

This feature is commonly used in output statements. For example,

```
System.out.println("The answer is " + answer);
```

is perfectly acceptable and prints what you would expect (and with correct spacing because of the space after the word `is`).

If you need to put multiple strings together, separated by a delimiter, use the static `join` method:

```
String all = String.join(" / ", "S", "M", "L", "XL");
   // all is the string "S / M / L / XL"
```

As of Java 11, there is a `repeat` method:

```
String repeated = "Java".repeat(3); // repeated is "JavaJavaJava"
```

3.6.3 Strings Are Immutable

The `String` class gives no methods that let you *change* a character in an existing string. If you want to turn `greeting` into `"Help!"`, you cannot directly change the last positions of `greeting` into `'p'` and `'!'`. If you are a C programmer, this can make you feel pretty helpless. How are we going to modify the string? In Java, it is quite easy: Concatenate the substring that you want to keep with the characters that you want to replace.

```
greeting = greeting.substring(0, 3) + "p!";
```

This declaration changes the current value of the `greeting` variable to `"Help!"`.

Since you cannot change the individual characters in a Java string, the documentation refers to the objects of the `String` class as *immutable*. Just as the number `3` is always 3, the string `"Hello"` will always contain the code-unit sequence for the characters `H`, `e`, `l`, `l`, `o`. You cannot change these values. Yet you can, as you just saw, change the contents of the string *variable* `greeting` and make it refer to a different string, just as you can make a numeric variable currently holding the value `3` hold the value `4`.

Isn't that a lot less efficient? It would seem simpler to change the code units than to build up a whole new string from scratch. Well, yes and no. Indeed, it isn't efficient to generate a new string that holds the concatenation of `"Hel"` and `"p!"`. But immutable strings have one great advantage: The compiler can arrange that strings are *shared*.

To understand how this works, think of the various strings as sitting in a common pool. String variables then point to locations in the pool. If you copy a string variable, both the original and the copy share the same characters.

Overall, the designers of Java decided that the efficiency of sharing outweighs the inefficiency of string editing by extracting substrings and concatenating. Look at your own programs; we suspect that most of the time, you don't change strings—you just compare them. (There is one common exception—assembling strings from individual characters or from shorter strings that come from the keyboard or a file. For these situations, Java provides a separate class that we describe in Section 3.6.9, "Building Strings," on p. 74.)

C++ NOTE: C programmers are generally bewildered when they see Java strings for the first time because they think of strings as arrays of characters:

```
char greeting[] = "Hello";
```

That is a wrong analogy: A Java string is roughly analogous to a `char*` pointer,

```
char* greeting = "Hello";
```

When you replace `greeting` with another string, the Java code does roughly the following:

```
char* temp = malloc(6);
strncpy(temp, greeting, 3);
strncpy(temp + 3, "p!", 3);
greeting = temp;
```

Sure, now `greeting` points to the string `"Help!"`. And even the most hardened C programmer must admit that the Java syntax is more pleasant than a sequence of `strncpy` calls. But what if we make another assignment to `greeting`?

```
greeting = "Howdy";
```

Don't we have a memory leak? After all, the original string was allocated on the heap. Fortunately, Java does automatic garbage collection. If a block of memory is no longer needed, it will eventually be recycled.

If you are a C++ programmer and use the `string` class defined by ANSI C++, you will be much more comfortable with the Java `String` type. C++ `string` objects also perform automatic allocation and deallocation of memory. The memory management is performed explicitly by constructors, assignment operators, and destructors. However, C++ strings are mutable—you can modify individual characters in a string.

3.6.4 Testing Strings for Equality

To test whether two strings are equal, use the `equals` method. The expression

```
s.equals(t)
```

returns `true` if the strings `s` and `t` are equal, `false` otherwise. Note that `s` and `t` can be string variables or string literals. For example, the expression

```
"Hello".equals(greeting)
```

is perfectly legal. To test whether two strings are identical except for the upper/lowercase letter distinction, use the `equalsIgnoreCase` method.

```
"Hello".equalsIgnoreCase("hello")
```

Do *not* use the == operator to test whether two strings are equal! It only determines whether or not the strings are stored in the same location. Sure, if strings are in the same location, they must be equal. But it is entirely possible to store multiple copies of identical strings in different places.

```
String greeting = "Hello"; // initialize greeting to a string
if (greeting == "Hello") . . .
   // probably true
if (greeting.substring(0, 3) == "Hel") . . .
   // probably false
```

If the virtual machine always arranges for equal strings to be shared, then you could use the == operator for testing equality. But only string *literals* are shared, not strings that are the result of operations like + or `substring`. Therefore, *never* use == to compare strings lest you end up with a program with the worst kind of bug—an intermittent one that seems to occur randomly.

C++ NOTE: If you are used to the C++ `string` class, you have to be particularly careful about equality testing. The C++ `string` class does overload the == operator to test for equality of the string contents. It is perhaps unfortunate that Java goes out of its way to give strings the same "look and feel" as numeric values but then makes strings behave like pointers for equality testing. The language designers could have redefined == for strings, just as they made a special arrangement for +. Oh well, every language has its share of inconsistencies.

C programmers never use == to compare strings but use `strcmp` instead. The Java method `compareTo` is the exact analog of `strcmp`. You can use

```
if (greeting.compareTo("Hello") == 0) . . .
```

but it seems clearer to use `equals` instead.

3.6.5 Empty and Null Strings

The empty string `""` is a string of length 0. You can test whether a string is empty by calling

```
if (str.length() == 0)
```

or

```
if (str.equals(""))
```

An empty string is a Java object which holds the string length (namely, `0`) and an empty contents. However, a `String` variable can also hold a special value, called `null`, that indicates that no object is currently associated with the variable. (See Chapter 4 for more information about `null`.) To test whether a string is `null`, use

```
if (str == null)
```

Sometimes, you need to test that a string is neither `null` nor empty. Then use

```
if (str != null && str.length() != 0)
```

You need to test that `str` is not `null` first. As you will see in Chapter 4, it is an error to invoke a method on a `null` value.

3.6.6 Code Points and Code Units

Java strings are sequences of `char` values. As we discussed in Section 3.3.3, "The `char` Type," on p. 46, the `char` data type is a code unit for representing Unicode code points in the UTF-16 encoding. The most commonly used Unicode characters can be represented with a single code unit. The supplementary characters require a pair of code units.

The `length` method yields the number of code units required for a given string in the UTF-16 encoding. For example:

```
String greeting = "Hello";
int n = greeting.length(); // is 5
```

To get the true length—that is, the number of code points—call

```
int cpCount = greeting.codePointCount(0, greeting.length());
```

The call `s.charAt(n)` returns the code unit at position `n`, where `n` is between `0` and `s.length()` − 1. For example:

```
char first = greeting.charAt(0); // first is 'H'
char last = greeting.charAt(4); // last is 'o'
```

To get at the `i`th code point, use the statements

```
int index = greeting.offsetByCodePoints(0, i);
int cp = greeting.codePointAt(index);
```

Why are we making a fuss about code units? Consider the sentence

𝕆 is the set of octonions.

The character 𝕆 (`U+1D546`) requires two code units in the UTF-16 encoding. Calling

```
char ch = sentence.charAt(1)
```

doesn't return a space but the second code unit of 𝕆. To avoid this problem, you should not use the `char` type. It is too low-level.

NOTE: Don't think that you can ignore exotic characters with code units above U+FFFF. Your emoji-loving users may put characters such as 🍺 (U+1F37A, beer mug) into strings.

If your code traverses a string, and you want to look at each code point in turn, you can use these statements:

```
int cp = sentence.codePointAt(i);
if (Character.isSupplementaryCodePoint(cp)) i += 2;
else i++;
```

You can move backwards with the following statements:

```
i--;
if (Character.isSurrogate(sentence.charAt(i))) i--;
int cp = sentence.codePointAt(i);
```

Obviously, that is quite painful. An easier way is to use the `codePoints` method that yields a "stream" of `int` values, one for each code point. (We will discuss streams in Chapter 2 of Volume II.) You can just turn the stream into an array (see Section 3.10, "Arrays," on p. 108) and traverse that.

```
int[] codePoints = str.codePoints().toArray();
```

Conversely, to turn an array of code points to a string, use a *constructor*. (We discuss constructors and the `new` operator in detail in Chapter 4.)

```
String str = new String(codePoints, 0, codePoints.length);
```

NOTE: The virtual machine does not have to implement strings as sequences of code units. In Java 9, strings that hold only single-byte code units use a `byte` array, and all others a `char` array.

3.6.7 The `String` API

The `String` class in Java contains more than 50 methods. A surprisingly large number of them are sufficiently useful that we can imagine using them frequently. The following API note summarizes the ones we found most useful.

These API notes, found throughout the book, will help you understand the Java Application Programming Interface (API). Each API note starts with the name of a class, such as `java.lang.String`. (The significance of the so-called *package* name `java.lang` is explained in Chapter 4.) The class name is followed by the names, explanations, and parameter descriptions of one or more methods.

We typically do not list all methods of a particular class but select those that are most commonly used and describe them in a concise form. For a full listing, consult the online documentation (see Section 3.6.8, "Reading the Online API Documentation," on p. 71).

We also list the version number in which a particular class was introduced. If a method has been added later, it has a separate version number.

`java.lang.String` 1.0

- `char charAt(int index)`

  returns the code unit at the specified location. You probably don't want to call this method unless you are interested in low-level code units.

(Continues)

java.lang.String 1.0 *(Continued)*

- `int codePointAt(int index)` 5

  returns the code point that starts at the specified location.

- `int offsetByCodePoints(int startIndex, int cpCount)` 5

  returns the index of the code point that is `cpCount` code points away from the code point at `startIndex`.

- `int compareTo(String other)`

  returns a negative value if the string comes before `other` in dictionary order, a positive value if the string comes after `other` in dictionary order, or `0` if the strings are equal.

- `IntStream codePoints()` 8

  returns the code points of this string as a stream. Call `toArray` to put them in an array.

- `new String(int[] codePoints, int offset, int count)` 5

  constructs a string with the `count` code points in the array starting at `offset`.

- `boolean isEmpty()`

  `boolean isBlank()` 11

  returns `true` if the string is empty or consists of whitespace.

- `boolean equals(Object other)`

  returns `true` if the string equals `other`.

- `boolean equalsIgnoreCase(String other)`

  returns `true` if the string equals `other`, except for upper/lowercase distinction.

- `boolean startsWith(String prefix)`
- `boolean endsWith(String suffix)`

  returns `true` if the string starts with `prefix` or ends with `suffix`.

- `int indexOf(String str)`
- `int indexOf(String str, int fromIndex)`
- `int indexOf(int cp)`
- `int indexOf(int cp, int fromIndex)`

  returns the start of the first substring equal to the string `str` or the code point `cp`, starting at index `0` or at `fromIndex`, or `-1` if `str` does not occur in this string.

(Continues)

java.lang.String 1.0 *(Continued)*

- `int lastIndexOf(String str)`
- `int lastIndexOf(String str, int fromIndex)`
- `int lastindexOf(int cp)`
- `int lastindexOf(int cp, int fromIndex)`

  returns the start of the last substring equal to the string `str` or the code point `cp`, starting at the end of the string or at `fromIndex`.

- `int length()`

  returns the number of code units of the string.

- `int codePointCount(int startIndex, int endIndex)` **5**

  returns the number of code points between `startIndex` and `endIndex` – 1.

- `String replace(CharSequence oldString, CharSequence newString)`

  returns a new string that is obtained by replacing all substrings matching `oldString` in the string with the string `newString`. You can supply `String` or `StringBuilder` objects for the `CharSequence` parameters.

- `String substring(int beginIndex)`
- `String substring(int beginIndex, int endIndex)`

  returns a new string consisting of all code units from `beginIndex` until the end of the string or until `endIndex` – 1.

- `String toLowerCase()`
- `String toUpperCase()`

  returns a new string containing all characters in the original string, with uppercase characters converted to lowercase, or lowercase characters converted to uppercase.

- `String trim()`

  `String strip()` **11**

  returns a new string by eliminating all leading and trailing characters that are ≤ U+0020 (`trim`) or whitespace (`strip`) in the original string.

- `String join(CharSequence delimiter, CharSequence... elements)` **8**

  returns a new string joining all elements with the given delimiter.

- `String repeat(int count)` **11**

  returns a string that repeats this string `count` times.

NOTE: In the API notes, there are a few parameters of type `CharSequence`. This is an *interface* type to which all strings belong. You will learn about interface types in Chapter 6. For now, you just need to know that you can pass arguments of type `String` whenever you see a `CharSequence` parameter.

3.6.8 Reading the Online API Documentation

As you just saw, the `String` class has lots of methods. Furthermore, there are thousands of classes in the standard libraries, with many more methods. It is plainly impossible to remember all useful classes and methods. Therefore, it is essential that you become familiar with the online API documentation that lets you look up all classes and methods in the standard library. You can download the API documentation from Oracle and save it locally, or you can point your browser to `https://docs.oracle.com/en/java/javase/11/docs/api`.

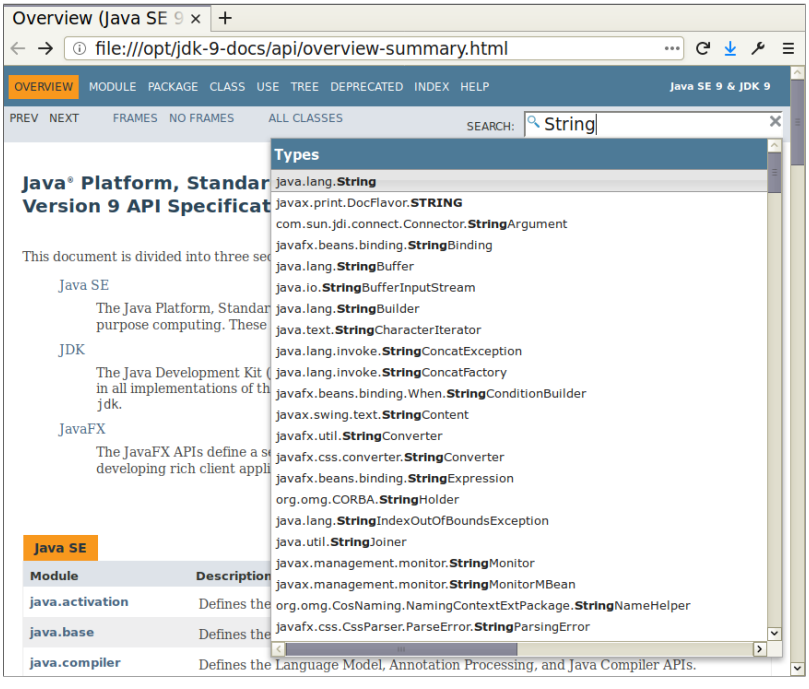

Figure 3.2 The Java API documentation

As of Java 9, the API documentation has a search box (see Figure 3.2). Older versions have frames with lists of packages and classes. You can still get those lists by clicking on the Frames menu item. For example, to get more information on the methods of the `String` class, type "String" into the search box and select the type `java.lang.String`, or locate the link in the frame with class names and click it. You get the class description, as shown in Figure 3.3.

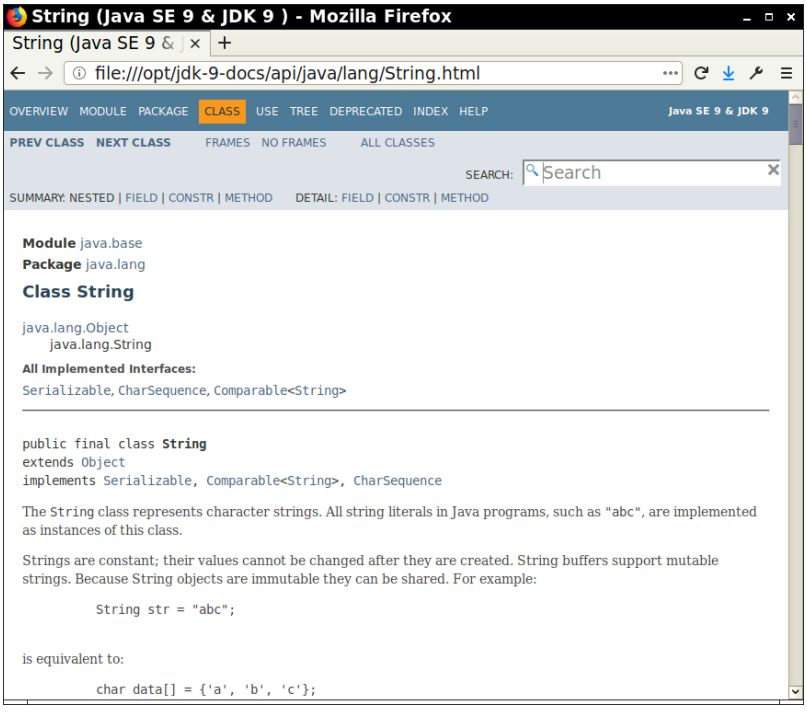

Figure 3.3 Class description for the `String` class

When you scroll down, you reach a summary of all methods, sorted in alphabetical order (see Figure 3.4). Click on any method name for a detailed description of that method (see Figure 3.5). For example, if you click on the compareToIgnoreCase link, you'll get the description of the `compareToIgnoreCase` method.

TIP: If you have not already done so, download the JDK documentation, as described in Chapter 2. Bookmark the `jdk-11-docs/index.html` page in your browser right now.

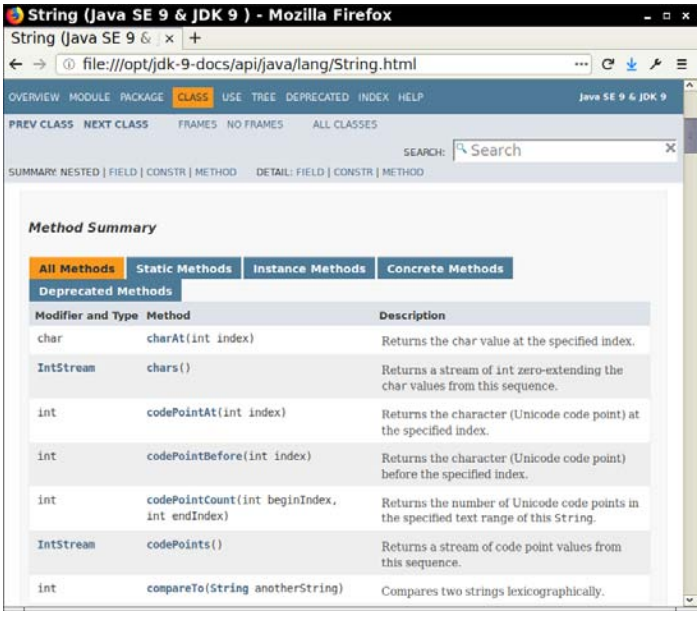

Figure 3.4 Method summary of the `String` class

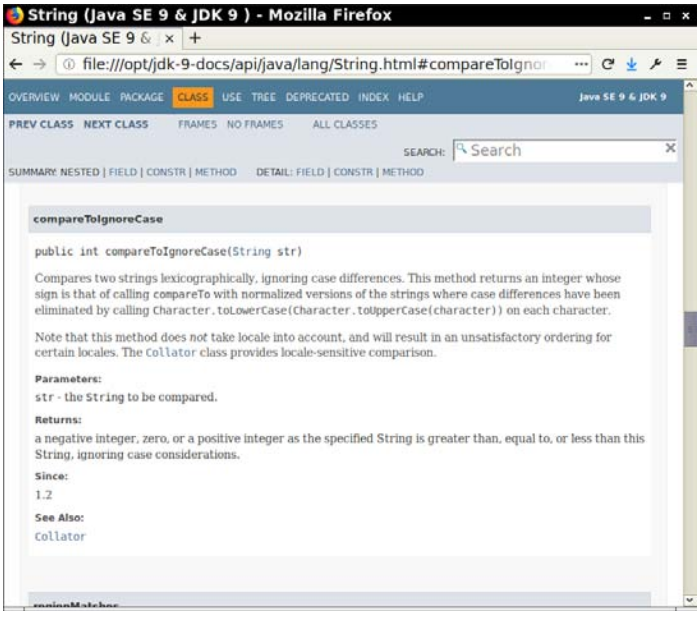

Figure 3.5 Detailed description of a `String` method

3.6.9 Building Strings

Occasionally, you need to build up strings from shorter strings, such as keystrokes or words from a file. It would be inefficient to use string concatenation for this purpose. Every time you concatenate strings, a new `String` object is constructed. This is time consuming and wastes memory. Using the `StringBuilder` class avoids this problem.

Follow these steps if you need to build a string from many small pieces. First, construct an empty string builder:

```
StringBuilder builder = new StringBuilder();
```

Each time you need to add another part, call the `append` method.

```
builder.append(ch); // appends a single character
builder.append(str); // appends a string
```

When you are done building the string, call the `toString` method. You will get a `String` object with the character sequence contained in the builder.

```
String completedString = builder.toString();
```

NOTE: The `StringBuilder` class was introduced in Java 5. Its predecessor, `StringBuffer`, is slightly less efficient, but it allows multiple threads to add or remove characters. If all string editing happens in a single thread (which is usually the case), you should use `StringBuilder` instead. The APIs of both classes are identical.

The following API notes contain the most important methods for the `StringBuilder` class.

java.lang.StringBuilder 5

- `StringBuilder()`
  constructs an empty string builder.
- `int length()`
  returns the number of code units of the builder or buffer.
- `StringBuilder append(String str)`
  appends a string and returns `this`.

(Continues)

java.lang.StringBuilder 5 *(Continued)*

- `StringBuilder append(char c)`

  appends a code unit and returns `this`.
- `StringBuilder appendCodePoint(int cp)`

  appends a code point, converting it into one or two code units, and returns `this`.
- `void setCharAt(int i, char c)`

  sets the `i`th code unit to `c`.
- `StringBuilder insert(int offset, String str)`

  inserts a string at position `offset` and returns `this`.
- `StringBuilder insert(int offset, char c)`

  inserts a code unit at position `offset` and returns `this`.
- `StringBuilder delete(int startIndex, int endIndex)`

  deletes the code units with offsets `startIndex` to `endIndex – 1` and returns `this`.
- `String toString()`

  returns a string with the same data as the builder or buffer contents.

3.7 Input and Output

To make our example programs more interesting, we want to accept input and properly format the program output. Of course, modern programs use a GUI for collecting user input. However, programming such an interface requires more tools and techniques than we have at our disposal at this time. Our first order of business is to become more familiar with the Java programming language, so we use the humble console for input and output.

3.7.1 Reading Input

You saw that it is easy to print output to the "standard output stream" (that is, the console window) just by calling `System.out.println`. Reading from the "standard input stream" `System.in` isn't quite as simple. To read console input, you first construct a `Scanner` that is attached to `System.in`:

```
Scanner in = new Scanner(System.in);
```

(We discuss constructors and the `new` operator in detail in Chapter 4.)