Java™ The Complete Reference Thirteenth Edition

Comprehensive Coverage of the Java Language

FULLY UPDATED AND EXPANDED

Herbert Schildt • Dr. Danny Coward

Mc Graw Hill

Java™ The Complete Reference Thirteenth Edition

Comprehensive Coverage of the Java Language

FULLY UPDATED AND EXPANDED

Herbert Schildt • Dr. Danny Coward

Mc Graw Hill

About the Authors

Best-selling author **Herbert Schildt** has written extensively about programming for over three decades and is a leading authority on the Java language. Called "one of the world's foremost authors of books about programming" by *International Developer* magazine, his books have sold millions of copies worldwide and have been translated into all major foreign languages. He is the author of numerous books on Java, including *Java: A Beginner's Guide, Herb Schildt's Java Programming Cookbook, Introducing JavaFX 8 Programming*, and *Swing: A Beginner's Guide*. He has also written extensively about C, C++, and C#. Featured as one of the rock star programmers in Ed Burns' book *Secrets of the Rock Star Programmers: Riding the IT Crest*, Schildt is interested in all facets of computing, but his primary focus is computer languages. Schildt holds both BA and MCS degrees from the University of Illinois.

Dr. Danny Coward has worked on all editions of the Java platform. He led the definition of Java Servlets into the first version of the Java EE platform and beyond, web services into the Java ME platform, and the strategy and planning for Java SE 7. He helped found JavaFX technology and designed the Java WebSocket API in Java EE. From coding in Java, to designing APIs with industry experts, to serving for several years as an executive to the Java Community Process, he has a uniquely broad perspective into multiple aspects of Java technology. In addition, he is the author of two books on Java programming: *Java WebSocket Programming* and *Java EE 7: The Big Picture*. More recently, he has been applying his knowledge of Java to scale Java-based services for one of the world's most successful software companies. Dr. Coward holds bachelor's, master's, and doctorate degrees in mathematics from the University of Oxford.

About the Technical Editor

Simon Ritter is the deputy CTO of Azul. Simon joined Sun Microsystems in 1996 and spent time working in both Java development and consultancy. He has been presenting Java technologies to developers since 1999, focusing on the core Java platform as well as client and embedded

applications. At Azul, he continues to help people understand Java and the JVM.

Simon is a Java Champion and two-time recipient of the JavaOne Rockstar award. In addition, he represents Azul on the JCP Executive Committee, the OpenJDK Vulnerability Group, and the JSR Expert Group since Java SE 9.

The Complete Reference

Java™
Thirteenth Edition

Herbert Schildt
Dr. Danny Coward

New York Chicago San Francisco
Athens London Madrid Mexico City
Milan New Delhi Singapore Sydney Toronto

ISBN: 978-1-26-506270-5
MHID: 1-26-506270-6

The material in this eBook also appears in the print version of this title: ISBN: 978-1-26-505843-2, MHID: 1-26-505843-1.

eBook conversion by codeMantra
Version 1.0

this Work, and is not responsible for any errors or omissions.

TERMS OF USE

Contents at a Glance

Contents

Foreword

Having written extensively about programming for nearly four decades, the time has come for me to retire. As a result, this edition of *Java: The Complete Reference* was prepared by Dr. Danny Coward. Danny is a Java expert, a published author, and the technical editor on several previous editions of this book. When it came time for me to retire, he generously agreed to take on the task of fully revising this book for JDK 21. As a result, all revisions, updates, and new material, such as coverage of **record** patterns, pattern matching for **switch**, and sequenced collections, were prepared and written by Danny. Revising a book of this size is no small task, and I want thank Danny for all his efforts in preparing this, the thirteenth edition of *Java: The Complete Reference*.

I also want to thank everyone at McGraw Hill. To become a successful writer requires a successful publisher. Each needs the other to create a quality book. (In other words, writing and publishing form two sides of the same coin.) Of course, a publisher is only as good as its people. Over these many years, I have been fortunate to have worked with some of the very best. Although there are far too many to name them all, I want to give special thanks to Wendy Rinaldi, Lisa McClain, Patty Mon, Jeff Pepper, and Scott Rogers. Friends, it's been one grand adventure and a wonderful time. All the best, and many thanks!

—HERBERT SCHILDT

Preface

Java is one of the world's most important and widely used computer languages. Furthermore, it has held that distinction for many years. Unlike some other computer languages whose influence has waned with the passage of time, Java's has grown stronger. Java leapt to the forefront of Internet programming with its first release. Each subsequent version has solidified that position. Today, it is still the first and best choice for developing web-based applications. It is also a powerful, general-purpose programming language suitable for a wide variety of purposes. Simply put: much of the modern world runs on Java code. Java really is that important.

A key reason for Java's success is its agility. Since its original 1.0 release, Java has continually adapted to changes in the programming environment and to changes in the way that programmers program. Most importantly, it has not just followed the trends, it has helped create them. Java's ability to accommodate the fast rate of change in the computing world is a crucial part of why it has been and continues to be so successful.

Since this book was first published in 1996, it has gone through several editions, each reflecting the ongoing evolution of Java. This is the thirteenth edition, and it has been updated for Java SE 21 (JDK 21). As a result, this edition of the book contains a substantial amount of new material, updates, and changes. Of special interest are the discussions of the following key features that have been added to the Java platform since the previous edition of this book:

- Pattern matching in **switch**
- Record patterns
- Sequenced collections
- Virtual threads

Collectively, these constitute a substantial set of new features that significantly expand the range, scope, and expressiveness of the platform. The first two features are additions to the Java language itself. Pattern

matching in **switch** expands the expressiveness of the **switch** statement to look for values matching a number of patterns, while record patterns allows pattern matching with Java records in **instanceof** expressions for concise and powerful forms of data processing. The additional APIs in Sequenced Collections defines the idea of the *encounter order* on several of the familiar Java Collection classes, while also providing uniform APIs to manage their first and last elements and for processing their elements backwards. Virtual Threads allow developers to create Java threads in their traditional form but that are managed by Java, not by the underlying OS, thereby opening up the possibilities to scale applications that require large numbers of threads more gracefully. Collectively, these new features fundamentally expand the ways in which you can design and implement solutions.

A Book for All Programmers

This book is for all programmers, whether you are a novice or an experienced pro. The beginner will find its carefully paced discussions and many examples especially helpful. Its in-depth coverage of Java's more advanced features and libraries will appeal to the pro. For both, it offers a lasting resource and handy reference.

What's Inside

This book is a comprehensive guide to the Java language, describing its syntax, keywords, and fundamental programming principles. Significant portions of the Java API library are also examined. The book is divided into four parts, each focusing on a different aspect of the Java programming environment.

Part I presents an in-depth tutorial of the Java language. It begins with the basics, including such things as data types, operators, control statements, and classes. It then moves on to inheritance, packages, interfaces, exception handling, and multithreading including the new virtual threading. Next, it describes annotations, enumerations, autoboxing, generics, modules, and lambda expressions. I/O is also introduced. The final chapter in Part I covers several recently added features: records, sealed

classes and interfaces, the enhanced **switch**, including pattern matching, record patterns, pattern matching with **instanceof**, and text blocks.

Part II examines key aspects of Java's standard API library. Topics include strings, I/O, networking, the standard utilities, the Collections Framework (including the newly added Sequenced Collections), the AWT, event handling, imaging, concurrency (including the Fork/Join Framework), regular expressions, and the stream library.

Part III offers three chapters that introduce Swing.

Part IV contains two chapters that show examples of Java in action. The first discusses Java Beans. The second presents an introduction to servlets.

Special Thanks

I thank my many mentors as a programmer and designer in Java. Bill Shannon, for teaching me to always strive for the best way to solve a problem; Graham Hamilton, for showing me what it means to be a thought leader' and James Gosling, for bringing out the magic in everyday work.

Thank you to Simon Ritter for his thoughtful and expert technical reviews for the updates to this book, both the detail and the bigger picture.

I give special thanks to Herb Schildt. In creating and maintaining this valuable and comprehensive book with such care and expertise over some 12 editions, he has helped so very many of his readers become skilled and knowledgeable Java programmers. I have enjoyed working with him as a technical reviewer for a number of past editions of this work and am grateful for his thoughtful guidance in my updates to this thirteenth edition.

DANNY COWARD

PART I

The Java Language

CHAPTER

1

The History and Evolution of Java

To fully understand Java, one must understand the reasons behind its creation, the forces that shaped it, and the legacy that it inherits. Like the successful computer languages that came before, Java is a blend of the best elements of its rich heritage combined with the innovative concepts required by its unique mission. While the remaining chapters of this book describe the practical aspects of Java—including its syntax, key libraries, and applications—this chapter explains how and why Java came about, what makes it so important, and how it has evolved over the years.

Although Java has become inseparably linked with the online environment of the Internet, it is important to remember that Java is first and foremost a programming language. Computer language innovation and development occur for two fundamental reasons:

- To adapt to changing environments and uses
- To implement refinements and improvements in the art of programming

As you will see, the development of Java was driven by both elements in nearly equal measure.

Java's Lineage

Java is related to C++, which is a direct descendant of C. Much of the character of Java is inherited from these two languages. From C, Java derives its syntax. Many of Java's object-oriented features were influenced by C++. In fact, several of Java's defining characteristics come from—or are responses to—its predecessors. Moreover, the creation of Java was deeply rooted in the process of refinement and adaptation that has been occurring in computer programming languages for the past several decades. For these reasons, this section reviews the sequence of events and forces that led to Java. As you will see, each innovation in language design was driven by the need to solve a fundamental problem that the preceding languages could not solve. Java is no exception.

The Birth of Modern Programming: C

The C language shook the computer world. Its impact should not be underestimated, because it fundamentally changed the way programming was approached and thought about. The creation of C was a direct result of the need for a structured, efficient, high-level language that could replace assembly code when creating systems programs. As you may know, when a computer language is designed, trade-offs are often made, such as the following:

- Ease-of-use versus power
- Safety versus efficiency
- Rigidity versus extensibility

Prior to C, programmers usually had to choose between languages that optimized one set of traits or the other. For example, although FORTRAN could be used to write fairly efficient programs for scientific applications, it was not very good for system code. And while BASIC was easy to learn, it wasn't very powerful, and its lack of structure made its usefulness questionable for large programs. Assembly language can be used to produce highly efficient programs, but it is not easy to learn or use effectively. Further, debugging assembly code can be quite difficult.

Another compounding problem was that early computer languages such as BASIC, COBOL, and FORTRAN were not designed around structured principles. Instead, they relied upon the GOTO as a primary means of

program control. As a result, programs written using these languages tended to produce "spaghetti code"—a mass of tangled jumps and conditional branches that make a program virtually impossible to understand. While languages like Pascal are structured, they were not designed for efficiency, and failed to include certain features necessary to make them applicable to a wide range of programs. (Specifically, given the standard dialects of Pascal available at the time, it was not practical to consider using Pascal for systems-level code.)

So, just prior to the invention of C, no one language had reconciled the conflicting attributes that had dogged earlier efforts. Yet the need for such a language was pressing. By the early 1970s, the computer revolution was beginning to take hold, and the demand for software was rapidly outpacing programmers' ability to produce it. A great deal of effort was being expended in academic circles in an attempt to create a better computer language. But, and perhaps most importantly, a secondary force was beginning to be felt. Computer hardware was finally becoming common enough that a critical mass was being reached. No longer were computers kept behind locked doors. For the first time, programmers were gaining virtually unlimited access to their machines. This allowed the freedom to experiment. It also allowed programmers to begin to create their own tools. On the eve of C's creation, the stage was set for a quantum leap forward in computer languages.

Invented and first implemented by Dennis Ritchie on a DEC PDP-11 running the UNIX operating system, C was the result of a development process that started with an older language called BCPL, developed by Martin Richards. BCPL influenced a language called B, invented by Ken Thompson, which led to the development of C in the 1970s. For many years, the de facto standard for C was the one supplied with the UNIX operating system and described in *The C Programming Language* by Brian Kernighan and Dennis Ritchie (Prentice-Hall, 1978). C was formally standardized in December 1989, when the American National Standards Institute (ANSI) standard for C was adopted.

The creation of C is considered by many to have marked the beginning of the modern age of computer languages. It successfully synthesized the conflicting attributes that had so troubled earlier languages. The result was a powerful, efficient, structured language that was relatively easy to learn. It also included one other, nearly intangible aspect: it was a *programmer's*

language. Prior to the invention of C, computer languages were generally designed either as academic exercises or by bureaucratic committees. C is different. It was designed, implemented, and developed by real, working programmers, reflecting the way that they approached the job of programming. Its features were honed, tested, thought about, and rethought by the people who actually used the language. The result was a language that programmers liked to use. Indeed, C quickly attracted many followers who had a near-religious zeal for it. As such, it found wide and rapid acceptance in the programmer community. In short, C is a language designed by and for programmers. As you will see, Java inherited this legacy.

C++: The Next Step

During the late 1970s and early 1980s, C became the dominant computer programming language, and it is still widely used today. Since C is a successful and useful language, you might ask why a need for something else existed. The answer is *complexity*. Throughout the history of programming, the increasing complexity of programs has driven the need for better ways to manage that complexity. C++ is a response to that need. To better understand why managing program complexity is fundamental to the creation of C++, consider the following.

Approaches to programming have changed dramatically since the invention of the computer. For example, when computers were first invented, programming was done by manually toggling in the binary machine instructions by use of the front panel. As long as programs were just a few hundred instructions long, this approach worked. As programs grew, assembly language was invented so that a programmer could deal with larger, increasingly complex programs by using symbolic representations of the machine instructions. As programs continued to grow, high-level languages were introduced that gave the programmer more tools with which to handle complexity.

The first widespread language was, of course, FORTRAN. While FORTRAN was an impressive first step, at the time it was hardly a language that encouraged clear and easy-to-understand programs. The 1960s gave birth to *structured programming*. This is the method of programming championed by languages such as C. The use of structured

languages enabled programmers to write, for the first time, moderately complex programs fairly easily. However, even with structured programming methods, once a project reaches a certain size, its complexity exceeds what a programmer can manage. By the early 1980s, many projects were pushing the structured approach past its limits. To solve this problem, a new way to program was invented, called *object-oriented programming (OOP)*. Object-oriented programming is discussed in detail later in this book, but here is a brief definition: OOP is a programming methodology that helps organize complex programs through the use of inheritance, encapsulation, and polymorphism.

In the final analysis, although C is one of the world's great programming languages, there is a limit to its ability to handle complexity. Once the size of a program exceeds a certain point, it becomes so complex that it is difficult to grasp as a totality. While the precise size at which this occurs differs, depending upon both the nature of the program and the programmer, there is always a threshold at which a program becomes unmanageable. C++ added features that enabled this threshold to be broken, allowing programmers to comprehend and manage larger programs.

C++ was invented by Bjarne Stroustrup in 1979, while he was working at Bell Laboratories in Murray Hill, New Jersey. Stroustrup initially called the new language "C with Classes." However, in 1983, the name was changed to C++. C++ extends C by adding object-oriented features. Because C++ is built on the foundation of C, it includes all of C's features, attributes, and benefits. This is a crucial reason for the success of C++ as a language. The invention of C++ was not an attempt to create a completely new programming language. Instead, it was an enhancement to an already highly successful one.

The Stage Is Set for Java

By the end of the 1980s and the early 1990s, object-oriented programming using C++ took hold. Indeed, for a brief moment it seemed as if programmers had finally found the perfect language. Because C++ blended the high efficiency and stylistic elements of C with the object-oriented paradigm, it was a language that could be used to create a wide range of programs. However, just as in the past, forces were brewing that would, once again, drive computer language evolution forward. Within a few

years, the World Wide Web and the Internet would reach critical mass. This event would precipitate another revolution in programming.

The Creation of Java

Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc., in 1991. It took 18 months to develop the first working version. This language was initially called "Oak," but was renamed "Java" in 1995. Between the initial implementation of Oak in the fall of 1992 and the public announcement of Java in the spring of 1995, many more people contributed to the design and evolution of the language. Bill Joy, Arthur van Hoff, Jonathan Payne, Frank Yellin, and Tim Lindholm were key contributors to the maturing of the original prototype.

Somewhat surprisingly, the original impetus for Java was not the Internet! Instead, the primary motivation was the need for a platform-independent (that is, architecture-neutral) language that could be used to create software to be embedded in various consumer electronic devices, such as microwave ovens and remote controls. As you can probably guess, many different types of CPUs are used as controllers. The trouble with C and C++ (and most other languages at the time) is that they are designed to be compiled for a specific target. Although it is possible to compile a C++ program for just about any type of CPU, to do so requires a full C++ compiler targeted for that CPU. The problem is that compilers are expensive and time-consuming to create. An easier—and more cost-efficient—solution was needed. In an attempt to find such a solution, Gosling and others began work on a portable, platform-independent language that could be used to produce code that would run on a variety of CPUs under differing environments. This effort ultimately led to the creation of Java.

About the time that the details of Java were being worked out, a second, and ultimately more important, factor was emerging that would play a crucial role in the future of Java. This second force was, of course, the World Wide Web. Had the Web not taken shape at about the same time that Java was being implemented, Java might have remained a useful but obscure language for programming consumer electronics. However, with the emergence of the World Wide Web, Java was propelled to the forefront

of computer language design, because the Web, too, demanded portable programs.

Most programmers learn early in their careers that portable programs are as elusive as they are desirable. While the quest for a way to create efficient, portable (platform-independent) programs is nearly as old as the discipline of programming itself, it had taken a back seat to other, more pressing problems. Further, because (at that time) much of the computer world had divided itself into the three competing camps of Intel, Macintosh, and UNIX, most programmers stayed within their fortified boundaries, and the urgent need for portable code was reduced. However, with the advent of the Internet and the Web, the old problem of portability returned with a vengeance. After all, the Internet consists of a diverse, distributed universe populated with various types of computers, operating systems, and CPUs. Even though many kinds of platforms are attached to the Internet, users would like them all to be able to run the same program. What was once an irritating but low-priority problem had become a high-profile necessity.

By 1993, it became obvious to members of the Java design team that the problems of portability frequently encountered when creating code for embedded controllers are also found when attempting to create code for the Internet. In fact, the same problem that Java was initially designed to solve on a small scale could also be applied to the Internet on a large scale. This realization caused the focus of Java to switch from consumer electronics to Internet programming. So, while the desire for an architecture-neutral programming language provided the initial spark, the Internet ultimately led to Java's large-scale success.

As mentioned earlier, Java derives much of its character from C and C++. This is by intent. The Java designers knew that using the familiar syntax of C and echoing the object-oriented features of C++ would make their language appealing to the legions of experienced C/C++ programmers. In addition to the surface similarities, Java shares some of the other attributes that helped make C and C++ successful. First, Java was designed, tested, and refined by real, working programmers. It is a language grounded in the needs and experiences of the people who devised it. Thus, Java is a programmer's language. Second, Java is cohesive and logically consistent. Third, except for those constraints imposed by the Internet environment, Java gives you, the programmer, full control. If you program

well, your programs reflect it. If you program poorly, your programs reflect that, too. Put differently, Java is not a language with training wheels. It is a language for professional programmers.

Because of the similarities between Java and C++, it is tempting to think of Java as simply the "Internet version of C++." However, to do so would be a large mistake. Java has significant practical and philosophical differences. While it is true that Java was influenced by C++, it is not an enhanced version of C++. For example, Java is neither upwardly nor downwardly compatible with C++. Of course, the similarities with C++ are significant, and if you are a C++ programmer, then you will feel right at home with Java. One other point: Java was not designed to replace C++. Java was designed to solve a certain set of problems. C++ was designed to solve a different set of problems. Both will coexist for many years to come.

As mentioned at the start of this chapter, computer languages evolve for two reasons: to adapt to changes in environment and to implement advances in the art of programming. The environmental change that prompted Java was the need for platform-independent programs destined for distribution on the Internet. However, Java also embodies changes in the way that people approach the writing of programs. For example, Java enhanced and refined the object-oriented paradigm used by C++, added integrated support for multithreading, and provided a library that simplified Internet access. In the final analysis, though, it was not the individual features of Java that made it so remarkable. Rather, it was the language as a whole. Java was the perfect response to the demands of the then newly emerging, highly distributed computing universe. Java was to Internet programming what C was to system programming: a revolutionary force that changed the world.

The C# Connection

The reach and power of Java continues to be felt throughout the world of computer language development. Many of its innovative features, constructs, and concepts have become part of the baseline for any new language. The success of Java is simply too important to ignore.

Perhaps the most important example of Java's influence is C#. Created by Microsoft to support the .NET Framework, C# is closely related to Java. For example, both share the same general syntax, support distributed

programming, and utilize the same object model. There are, of course, differences between Java and C#, but the overall "look and feel" of these languages is very similar. This "cross-pollination" from Java to C# is the strongest testimonial to date that Java redefined the way we think about and use a computer language.

Longevity

Even as Java has broadened its range and power, other general-purpose computer languages have been created. Some are inspired by the success of Java, often emulating some of its features, even some running on its virtual machine. Some address perceived weaknesses of Java in novel ways. These newer languages include Kotlin, which uses the same bytecode representation but aims to be simpler and easier to learn, and Ruby, a dynamically interpreted language with a straightforward syntax to allow applications to be developed quickly. Other languages include Go, Scala, and Rust. With each year that passes the list grows. But one constant in the near three decades since Java was released has been its popularity with developers. This has kept it at or near the top of the most widely used languages for computing. Perhaps this is testimony to how well Java has continued to evolve to meet the changing needs of developers and the work they do.

How Java Impacted the Internet

The Internet helped catapult Java to the forefront of programming, and Java, in turn, had a profound effect on the Internet. In addition to simplifying web programming in general, Java innovated a new type of networked program called the applet that changed the way the online world thought about content. Java also addressed some of the thorniest issues associated with the Internet: portability and security. Let's look more closely at each of these.

Java Applets

At the time of Java's creation, one of its most exciting features was the applet. An *applet* is a special kind of Java program that is designed to be

transmitted over the Internet and automatically executed inside a Java-compatible web browser. If the user clicks a link that contains an applet, the applet will download and run in the browser. Applets were intended to be small programs. They were typically used to display data provided by the server, handle user input, or provide simple functions, such as a loan calculator, that execute locally, rather than on the server. In essence, the applet allowed some functionality to be moved from the server to the client.

The creation of the applet was important because, at the time, it expanded the universe of objects that could move about freely in cyberspace. In general, there are two very broad categories of objects that are transmitted between the server and the client: passive information and dynamic, active programs. For example, when you read your e-mail, you are viewing passive data. Even when you download a program, the program's code is still only passive data until you execute it. By contrast, the applet is a dynamic, self-executing program. Such a program is an active agent on the client computer, yet it is initiated by the server.

In the early days of Java, applets were a crucial part of Java programming. They illustrated the power and benefits of Java, added an exciting dimension to web pages, and enabled programmers to explore the full extent of what was possible with Java. Although it is likely that there are still applets in use today, over time they became less important. For reasons that will be explained, beginning with JDK 9, the phase-out of applets began, with applet support being removed by JDK 11.

Security

As desirable as dynamic, networked programs are, they can also present serious problems in the areas of security and portability. Obviously, a program that downloads and executes on the client computer must be prevented from doing harm. It must also be able to run in a variety of different environments and under different operating systems. As you will see, Java solved these problems in an effective and elegant way. Let's look a bit more closely at each, beginning with security.

As you are likely aware, every time you download a "normal" program, you are taking a risk, because the code you are downloading might contain a virus, Trojan horse, or other harmful code. At the core of the problem is the fact that malicious code can cause its damage because it has gained

unauthorized access to system resources. For example, a virus program might gather private information, such as credit card numbers, bank account balances, and passwords, by searching the contents of your computer's local file system. In order for Java to enable programs to be safely downloaded and executed on the client computer, it was necessary to prevent them from launching such an attack.

Java achieved this protection by enabling you to confine an application to the Java execution environment and prevent it from accessing other parts of the computer. (You will see how this is accomplished shortly.) The ability to download programs with a degree of confidence that no harm will be done may have been the single most innovative aspect of Java.

Portability

Portability is a major aspect of the Internet because there are many different types of computers and operating systems connected to it. If a Java program were to be run on virtually any computer connected to the Internet, there needed to be some way to enable that program to execute on different systems. In other words, a mechanism that allows the same application to be downloaded and executed by a wide variety of CPUs, operating systems, and browsers is required. It is not practical to have different versions of the application for different computers. The *same* application code must work on *all* computers. Therefore, some means of generating portable executable code was needed. As you will soon see, the same mechanism that helps ensure security also helps create portability.

Java's Magic: The Bytecode

The key that allowed Java to solve both the security and the portability problems just described is that the output of a Java compiler is not executable code. Rather, it is bytecode. *Bytecode* is a highly optimized set of instructions designed to be executed by what is called the *Java Virtual Machine (JVM)*, which is part of the Java Runtime Environment (JRE). In essence, the original JVM was designed as an *interpreter for bytecode*. This may come as a bit of a surprise since many modern languages are designed to be compiled into executable code because of performance concerns.

However, the fact that a Java program is executed by the JVM helps solve the major problems associated with web-based programs. Here is why.

Translating a Java program into bytecode makes it much easier to run a program in a wide variety of environments because only the JVM needs to be implemented for each platform. Once a JRE exists for a given system, any Java program can run on it. Remember, although the details of the JVM will differ from platform to platform, all understand the same Java bytecode. If a Java program were compiled to native code, then different versions of the same program would have to exist for each type of CPU connected to the Internet. This is, of course, not a feasible solution. Thus, the execution of bytecode by the JVM is the easiest way to create truly portable programs.

The fact that a Java program is executed by the JVM also helps to make it secure. Because the JVM is in control, it manages program execution. Thus, it is possible for the JVM to create a restricted execution environment, called the *sandbox*, that contains the program, preventing unrestricted access to the machine. Safety is also enhanced by certain restrictions that exist in the Java language.

In general, when a program is compiled to an intermediate form and then interpreted by a virtual machine, it runs slower than it would run if compiled to executable code. However, with Java, the differential between the two is not so great. Because bytecode has been highly optimized, the use of bytecode enables the JVM to execute programs much faster than you might expect.

Although Java was designed as an interpreted language, there is nothing about Java that prevents on-the-fly compilation of bytecode into native code in order to boost performance. For this reason, the HotSpot technology was introduced not long after Java's initial release. HotSpot provides a just-in-time (JIT) compiler for bytecode. When a JIT compiler is part of the JVM, selected portions of bytecode are compiled into executable code in real time, on a piece-by-piece, demand basis. It is important to understand that an entire Java program is not compiled into executable code all at once. Instead, a JIT compiler compiles code as it is needed, during execution. Furthermore, not all sequences of bytecode are compiled—only those that will benefit from compilation. The remaining code is simply interpreted. However, the just-in-time approach still yields a significant performance boost. Even when dynamic compilation is applied to bytecode,

the portability and safety features still apply, because the JVM is still in charge of the execution environment.

One other point: There has been experimentation with an *ahead-of-time* compiler for Java. Such a compiler can be used to compile bytecode into native code *prior* to execution by the JVM, rather than on-the-fly. Some previous versions of the JDK supplied an experimental ahead-of-time compiler; however, JDK 17 has removed it. Ahead-of-time compilation is a specialized feature, and it does not replace Java's traditional approach just described. Because of the highly specialized nature of ahead-of-time compilation, it is not discussed further in this book.

Moving Beyond Applets

At the time of this writing, it has been nearly three decades since Java's original release. Over those years, many changes have taken place. At the time of Java's creation, the Internet was a new and exciting innovation; web browsers were undergoing rapid development and refinement; the modern form of the smart phone had not yet been invented; and the near ubiquitous use of computers was still a few years off. As you would expect, Java has also changed and so, too, has the way that Java is used. Perhaps nothing illustrates the ongoing evolution of Java better than the applet.

As explained previously, in the early years of Java, applets were a crucial part of Java programming. They not only added excitement to a web page, they were also a highly visible part of Java, which added to its charisma. However, applets rely on a Java browser plug-in. Thus, for an applet to work, the browser must support it. Over the past few years, support for the Java browser plug-in has been waning. Simply put, without browser support, applets are not viable. Because of this, beginning with JDK 9, the phase-out of applets was begun, with support for applets being deprecated. In the language of Java, *deprecated* means that a feature is still available but flagged as obsolete. Thus, a deprecated feature should not be used for new code. The phase-out became complete with the release of JDK 11 because run-time support for applets was removed. Beginning with JDK 17, the entire Applet API was deprecated for removal.

As a point of interest, a few years after Java's creation an alternative to applets was added to Java. Called Java Web Start, it enabled an application to be dynamically downloaded from a web page. It was a deployment

mechanism that was especially useful for larger Java applications that were not appropriate for applets. The difference between an applet and a Web Start application is that a Web Start application runs on its own, not inside the browser. Thus, it looks much like a "normal" application. It does, however, require that a stand-alone JRE that supports Web Start is available on the host system. Beginning with JDK 11, Java Web Start support has been removed.

Given that neither applets nor Java Web Start are supported by modern versions of Java, you might wonder what mechanism should be used to deploy a Java application. At the time of this writing, part of the answer is to use the **jlink** tool added by JDK 9. It can create a complete run-time image that includes all necessary support for your program, including the JRE. Another part of the answer is the **jpackage** tool. Added by JDK 16, it can be used to create a ready-to-install application. Although a detailed discussion of deployment strategies is outside the scope of this book, it is something that you will want to pay close attention to going forward.

A Faster Release Schedule

Another major change has recently occurred in Java, but it does not involve changes to the language or the run-time environment. Rather, it relates to the way that Java releases are scheduled. In the past, major Java releases were typically separated by two or more years. However, subsequent to the release of JDK 9, the time between major Java releases has been decreased. Today, major releases occur every six months.

Each major release, called a *feature release*, includes those features ready at the time of the release. This increased *release cadence* enables new features and enhancements to be available to Java programmers in a timely fashion. Furthermore, it allows Java to respond quickly to the demands of an ever-changing programming environment. Simply put, the faster release schedule promises to be a very positive development for Java programmers.

In addition, feature releases include new features that are available to developers if they explicitly enable them to try them out but that have not yet been finalized. There are two types of these features. First, Preview Features are APIs and/or new language syntax that are fully designed and implemented but may yet be refined before being formally included in a future release based on feedback from developers trying them out. The

second type of such experimental features is Incubator Modules, which is a feature that is still evolving significantly and may or may not make it into a future release. Both types of "nonfinal" features give strong indications of what is to come next in feature releases.

With a cadence that is now every two years, a feature release will also be one that is supported (and thus remains viable) for a period of time longer than the six months until the next feature release. Such a release is called a long-term support release (LTS). An LTS release will be supported (and thus remain viable) for a period of time longer than six months. The first LTS release was JDK 11. The second LTS release was JDK 17 and the third JDK 21, for which this book has been updated. Because of the stability that an LTS release offers, it is likely that its feature set will define a baseline of functionality for a number of years. Consult Oracle for the latest information concerning long-term support and the LTS release schedule.

Currently, feature releases are scheduled for March and September of each year. As a result, JDK 10 was released in March 2018, which was six months after the release of JDK 9. The next release (JDK 11) was in September 2018. JDK 11 was an LTS release. This was followed by JDK 12 in March 2019, JDK 13 in September 2019, and so on. At the time of this writing, the latest release is JDK 21, which is an LTS release. Again, it is anticipated that every six months a new feature release will take place. Of course, you will want to consult the latest release schedule information.

At the time of this writing, there are a number of new Java features on the horizon. Because of the faster release schedule, it is very likely that several of them will be added to Java over the next few years. You will want to review the information and release notes provided by each six-month release in detail. It is truly an exciting time to be a Java programmer!

Servlets: Java on the Server Side

Client-side code is just one half of the client/server equation. Not long after the initial release of Java, it became obvious that Java would also be useful on the server side. One result was the *servlet*. A servlet is a small program that executes on the server.

Servlets are used to create dynamically generated content that is then served to the client. For example, an online store might use a servlet to look up the price for an item in a database. The price information is then used to

dynamically generate a web page that is sent to the browser. Although dynamically generated content was available through mechanisms such as CGI (Common Gateway Interface), the servlet offered several advantages, including increased performance.

Because servlets (like all Java programs) are compiled into bytecode and executed by the JVM, they are highly portable. Thus, the same servlet can be used in a variety of different server environments. The only requirements are that the server support the JVM and a servlet container. Today, server-side code in general constitutes a major use of Java.

The Java Buzzwords

No discussion of Java's history is complete without a look at the Java buzzwords. Although the fundamental forces that necessitated the invention of Java are portability and security, other factors also played an important role in molding the final form of the language. The key considerations were summed up by the Java team in the following list of buzzwords:

- Simple
- Secure
- Portable
- Object-oriented
- Robust
- Multithreaded
- Architecture-neutral
- Interpreted
- High performance
- Distributed
- Dynamic

Two of these buzzwords have already been discussed: secure and portable. Let's examine what each of the others implies.

Simple

Java was designed to be easy for the professional programmer to learn and use effectively. Assuming that you have some programming experience, you will not find Java hard to master. If you already understand the basic concepts of object-oriented programming, learning Java will be even easier. Best of all, if you are an experienced C++ programmer, moving to Java will require very little effort. Because Java inherits the C/C++ syntax and many of the object-oriented features of C++, most programmers have little trouble learning Java.

Object-Oriented

Although influenced by its predecessors, Java was not designed to be source-code compatible with any other language. This allowed the Java team the freedom to design with a blank slate. One outcome of this was a clean, usable, pragmatic approach to objects. Borrowing liberally from many seminal object-software environments of the last few decades, Java managed to strike a balance between the purist's "everything is an object" paradigm and the pragmatist's "stay out of my way" model. The object model in Java is simple and easy to extend, while primitive types, such as integers, were kept as high-performance nonobjects.

Robust

The multiplatformed environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems. Thus, the ability to create robust programs was given a high priority in the design of Java. To gain reliability, Java restricts you in a few key areas to force you to find your mistakes early in program development. At the same time, Java frees you from having to worry about many of the most common causes of programming errors. Because Java is a strictly typed language, it checks your code at compile time. However, it also checks your code at run time. Many hard-to-track-down bugs that often turn up in hard-to-reproduce run-time situations are simply impossible to create in Java. Knowing that what you have written will behave in a predictable way under diverse conditions is a key feature of Java.

To better understand how Java is robust, consider two of the main reasons for program failure: memory management mistakes and mishandled

exceptional conditions (that is, run-time errors). Memory management can be a difficult, tedious task in traditional programming environments. For example, in C/C++, the programmer will often manually allocate and free dynamic memory. This sometimes leads to problems, because programmers will either forget to free memory that has been previously allocated or, worse, try to free some memory that another part of their code is still using. Java virtually eliminates these problems by managing memory allocation and deallocation for you. (In fact, deallocation is completely automatic, because Java provides garbage collection for unused objects.) Exceptional conditions in traditional environments often arise in situations such as division by zero or "file not found," and they must be managed with clumsy and hard-to-read constructs. Java helps in this area by providing object-oriented exception handling. In a well-written Java program, all run-time errors can—and should—be managed by your program.

Multithreaded

Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things simultaneously. The Java run-time system comes with an elegant yet sophisticated solution for multiprocess synchronization that enables you to construct smoothly running interactive systems. Java's easy-to-use approach to multithreading allows you to think about the specific behavior of your program, not the multitasking subsystem.

Architecture-Neutral

A central issue for the Java designers was that of code longevity and portability. At the time of Java's creation, one of the main problems facing programmers was that no guarantee existed that if you wrote a program today, it would run tomorrow—even on the same machine. Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction. The Java designers made several hard decisions in the Java language and the Java Virtual Machine in an attempt to alter this situation. Their goal was "write once; run anywhere, any time, forever." To a great extent, this goal was accomplished.

Interpreted and High Performance

As described earlier, Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. This code can be executed on any system that implements the Java Virtual Machine. Most previous attempts at cross-platform solutions have done so at the expense of performance. As explained earlier, the Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler. Java run-time systems that provide this feature lose none of the benefits of the platform-independent code.

Distributed

Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols. In fact, accessing a resource using a URL is not much different from accessing a file. Java also supports *Remote Method Invocation (RMI)*. This feature enables a program to invoke methods across a network.

Dynamic

Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time. This makes it possible to dynamically link code in a safe and expedient manner. This is crucial to the robustness of the Java environment, in which small fragments of bytecode may be dynamically updated on a running system.

The Evolution of Java

The initial release of Java was nothing short of revolutionary, but it did not mark the end of Java's era of rapid innovation. Unlike most other software systems that usually settle into a pattern of small, incremental improvements, Java continued to evolve at an explosive pace. Soon after the release of Java 1.0, the designers of Java had already created Java 1.1. The features added by Java 1.1 were more significant and substantial than

the increase in the minor revision number would have you think. Java 1.1 added many new library and language features, introduced anonymous and inner classes, redefined the way events are handled, and reconfigured many features of the 1.0 library. It also deprecated (rendered obsolete) several features originally defined by Java 1.0. Thus, Java 1.1 both added to and subtracted from attributes of its original specification.

The next major release of Java was Java 2, where the "2" indicates "second generation." The creation of Java 2 was a watershed event, marking the beginning of Java's "modern age." The first release of Java 2 carried the version number 1.2. It may seem odd that the first release of Java 2 used the 1.2 version number. The reason is that it originally referred to the internal version number of the Java libraries, but then was generalized to refer to the entire release. With Java 2, Sun repackaged the Java product as J2SE (Java 2 Platform Standard Edition), and the version numbers began to be applied to that product.

Java 2 added support for a number of new features, such as Swing and the Collections Framework, and it enhanced the Java Virtual Machine and various programming tools. Java 2 also contained a few deprecations. The most important affected the **Thread** class in which the methods **suspend( )**, **resume( )**, and **stop( )** were deprecated.

J2SE 1.3 was the first major upgrade to the original Java 2 release. For the most part, it added to existing functionality and "tightened up" the development environment. In general, programs written for version 1.2 and those written for version 1.3 are source-code compatible. Although version 1.3 contained a smaller set of changes than the preceding three major releases, it was nevertheless important.

The release of J2SE 1.4 further enhanced Java. This release contained several important upgrades, enhancements, and additions. For example, it added the new keyword **assert**, chained exceptions, and a channel-based I/O subsystem. It also made changes to the Collections Framework and the networking classes. In addition, numerous small changes were made throughout. Despite the significant number of new features, version 1.4 maintained nearly 100 percent source-code compatibility with prior versions.

The next release of Java was J2SE 5, and it was revolutionary. Unlike most of the previous Java upgrades, which offered important, but measured improvements, J2SE 5 fundamentally expanded the scope, power, and

range of the language. To grasp the magnitude of the changes that J2SE 5 made to Java, consider the following list of its major new features:

- Generics
- Annotations
- Autoboxing and auto-unboxing
- Enumerations
- Enhanced, for-each style **for** loop
- Variable-length arguments (varargs)
- Static import
- Formatted I/O
- Concurrency utilities

This is not a list of minor tweaks or incremental upgrades. Each item in the list represented a significant addition to the Java language. Some, such as generics, the enhanced **for**, and varargs, introduced new syntax elements. Others, such as autoboxing and auto-unboxing, altered the semantics of the language. Annotations added an entirely new dimension to programming. In all cases, the impact of these additions went beyond their direct effects. They changed the very character of Java itself.

The importance of these new features is reflected in the use of the version number "5." The next version number for Java would normally have been 1.5. However, the new features were so significant that a shift from 1.4 to 1.5 just didn't seem to express the magnitude of the change. Instead, Sun elected to increase the version number to 5 as a way of emphasizing that a major event was taking place. Thus, it was named J2SE 5, and the developer's kit was called JDK 5. However, in order to maintain consistency, Sun decided to use 1.5 as its internal version number, which is also referred to as the *developer version* number. The "5" in J2SE 5 is called the *product version* number.

The next release of Java was called Java SE 6. Sun once again decided to change the name of the Java platform. First, notice that the "2" was dropped. Thus, the platform was now named *Java SE*, and the official product name was *Java Platform, Standard Edition 6*. The Java Development Kit was called JDK 6. As with J2SE 5, the 6 in Java SE 6 is the product version number. The internal, developer version number is 1.6.

Java SE 6 built on the base of J2SE 5, adding incremental improvements. Java SE 6 added no major features to the Java language proper, but it did enhance the API libraries, added several new packages, and offered improvements to the run time. It also went through several updates during its (in Java terms) long life cycle, with several upgrades added along the way. In general, Java SE 6 served to further solidify the advances made by J2SE 5.

Java SE 7 was the next release of Java, with the Java Development Kit being called JDK 7, and an internal version number of 1.7. Java SE 7 was the first major release of Java after Sun Microsystems was acquired by Oracle. Java SE 7 contained many new features, including significant additions to the language and the API libraries. Upgrades to the Java run-time system that support non-Java languages were also included, but it is the language and library additions that were of most interest to Java programmers.

The new language features were developed as part of *Project Coin*. The purpose of Project Coin was to identify a number of small changes to the Java language that would be incorporated into JDK 7. Although these features were collectively referred to as "small," the effects of these changes have been quite large in terms of the code they impact. In fact, for many programmers, these changes may well have been the most important new features in Java SE 7. Here is a list of the language features added by JDK 7:

- A **String** can now control a **switch** statement.
- Binary integer literals.
- Underscores in numeric literals.
- An expanded **try** statement, called ***try****-with-resources*, that supports automatic resource management. (For example, streams can be closed automatically when they are no longer needed.)
- Type inference (via the *diamond* operator) when constructing a generic instance.
- Enhanced exception handling in which two or more exceptions can be caught by a single **catch** (multi-catch) and better type checking for exceptions that are rethrown.
- Although not a syntax change, the compiler warnings associated with some types of varargs methods were improved, and you have more

control over the warnings.

As you can see, even though the Project Coin features were considered small changes to the language, their benefits were much larger than the qualifier "small" would suggest. In particular, the **try**-*with-resources* statement has profoundly affected the way that stream-based code is written. Also, the ability to use a **String** to control a **switch** statement was a long-desired improvement that simplified coding in many situations.

Java SE 7 made several additions to the Java API library. Two of the most important were the enhancements to the NIO Framework and the addition of the Fork/Join Framework. NIO (which originally stood for *New I/O*) was added to Java in version 1.4. However, the changes added by Java SE 7 fundamentally expanded its capabilities. So significant were the changes that the term *NIO.2* is often used.

The Fork/Join Framework provides important support for *parallel programming*. Parallel programming is the name commonly given to the techniques that make effective use of computers that contain more than one processor, including multicore systems. The advantage that multicore environments offer is the prospect of significantly increased program performance. The Fork/Join Framework addressed parallel programming by:

- Breaking down large computing tasks into a number of smaller tasks that can be executed in parallel
- Automatically making use of multiple processors

Therefore, by using the Fork/Join Framework, you can design applications that automatically take advantage of the processors available in the execution environment. Of course, not all algorithms lend themselves to parallelization, but for those that do, a significant improvement in execution speed can be obtained.

The next release of Java was Java SE 8, with the developer's kit being called JDK 8. It has an internal version number of 1.8. JDK 8 was a significant upgrade to the Java language because of the inclusion of a far-reaching new language feature: the *lambda expression*. The impact of lambda expressions was, and will continue to be, profound, changing both the way that programming solutions are conceptualized and how Java code

is written. As explained in detail in Chapter 15, lambda expressions add functional programming features to Java. In the process, lambda expressions can simplify and reduce the amount of source code needed to create certain constructs, such as some types of anonymous classes. The addition of lambda expressions also caused a new operator (the –>) and a new syntax element to be added to the language.

The inclusion of lambda expressions has also had a wide-ranging effect on the Java libraries, with new features being added to take advantage of them. One of the most important was the new stream API, which is packaged in **java.util.stream**. The stream API supports pipeline operations on data and is optimized for lambda expressions. Another new package was **java.util.function**. It defines a number of *functional interfaces*, which provide additional support for lambda expressions. Other new lambda-related features are found throughout the API library.

Another lambda-inspired feature affects **interface**. Beginning with JDK 8, it is now possible to define a default implementation for a method specified by an interface. If no implementation for a default method is created, then the default defined by the interface is used. This feature enables interfaces to be gracefully evolved over time because a new method can be added to an interface without breaking existing code. It can also streamline the implementation of an interface when the defaults are appropriate. Other new features in JDK 8 include a new time and date API, type annotations, and the ability to use parallel processing when sorting an array, among others.

The next release of Java was Java SE 9. The developer's kit was called JDK 9. With the release of JDK 9, the internal version number is also 9. JDK 9 represented a major Java release, incorporating significant enhancements to both the Java language and its libraries. Like the JDK 5 and JDK 8 releases, JDK 9 affected the Java language and its API libraries in fundamental ways.

The primary new JDK 9 feature was *modules*, which enable you to specify the relationship and dependencies of the code that comprises an application. Modules also add another dimension to Java's access control features. The inclusion of modules caused a new syntax element and several keywords to be added to Java. Furthermore, a tool called **jlink** was added to the JDK, which enables a programmer to create a run-time image of an application that contains only the necessary modules. A new file type,

called JMOD, was created. Modules also have a profound affect on the API library because, beginning with JDK 9, the library packages are now organized into modules.

Although modules constitute a major Java enhancement, they are conceptually simple and straightforward. Furthermore, because pre-module legacy code is fully supported, modules can be integrated into the development process on your timeline. There is no need to immediately change any preexisting code to handle modules. In short, modules added substantial functionality without altering the essence of Java.

In addition to modules, JDK 9 included many other new features. One of particular interest is JShell, which is a tool that supports interactive program experimentation and learning. (An introduction to JShell is found in Appendix B.) Another interesting upgrade is support for private interface methods. Their inclusion further enhanced JDK 8's support for default methods in interfaces. JDK 9 added a search feature to the **javadoc** tool and a new tag called **@index** to support it. As with previous releases, JDK 9 contained a number of enhancements to Java's API libraries.

As a general rule, in any Java release, it is the new features that receive the most attention. However, there was one high-profile aspect of Java that was deprecated by JDK 9: applets. Beginning with JDK 9, applets were no longer recommended for new projects. As explained earlier in this chapter, because of waning browser support for applets (and other factors), JDK 9 deprecated the entire applet API.

The next release of Java was Java SE 10 (JDK 10), which was released in March 2018. The primary new language feature added by JDK 10 was support for *local variable type inference*. With local variable type inference, it is now possible to let the type of a local variable be inferred from the type of its initializer, rather than being explicitly specified. To support this new capability, the context-sensitive keyword **var** was added to Java. Type inference can streamline code by eliminating the need to redundantly specify a variable's type when it can be inferred from its initializer. It can also simplify declarations in cases in which the type is difficult to discern or cannot be explicitly specified. Local variable type inference has become a common part of the contemporary programming environment. Its inclusion in Java helps keep Java up to date with evolving trends in language design. Along with a number of other changes, JDK 10 also

redefined the Java version string, changing the meaning of the version numbers so that they better align with the new time-based release schedule.

The next version of Java was Java SE 11 (JDK 11). It was released in September 2018, which was six months after JDK 10. JDK 11 was an LTS release. The primary new language feature in JDK 11 was support for the use of **var** in a lambda expression. Along with a number of tweaks and updates to the API in general, JDK 11 added a new networking API, which will be of interest to a wide range of developers. Called the *HTTP Client API*, it is packaged in **java.net.http**, and it provides enhanced, updated, and improved networking support for HTTP clients. Also, another execution mode was added to the Java launcher that enables it to directly execute simple single-file programs. JDK 11 also removed some features. Perhaps of the greatest interest because of its historical significance is the removal of support for applets. Recall that applets were first deprecated by JDK 9. With the release of JDK 11, applet support has been removed. Support for another deployment-related technology called Java Web Start was also removed from JDK 11. As the execution environment has continued to evolve, both applets and Java Web Start were rapidly losing relevance. Another key change in JDK 11 is that JavaFX was no longer included in the JDK. Instead, this GUI framework has become a separate open-source project. Because these features are no longer part of the JDK, they are not discussed in this book.

Between the JDK 11 LTS and the next LTS release (JDK 17) were five feature releases: JDK 12 through JDK 16. JDK 14 added support for the **switch** expression, which is a **switch** that produces a value and which was included in JDK 12 in preview form. Other enhancements to **switch** were also included. Text blocks, which are essentially string literals that can span more than one line, were added by JDK 15, after being previewed in JDK 13. JDK 16 enhanced **instanceof** with pattern matching and added a new type of class called a *record* along with the new context-sensitive keyword **record**. A record provides a convenient means of aggregating data. JDK 16 also supplied a new application packaging tool called **jpackage**.

JDK 17 was the second LTS release. Its major new feature was the ability to seal classes and interfaces. Sealing gives you control over the inheritance of a class and the inheritance and implementation of an interface. To this end, it adds the context-sensitive keywords **sealed**,

permits, and **non-sealed**, which is the first hyphenated Java keyword. JDK 17 marks the applet API as deprecated for removal in a future release.

At the time of this writing, Java SE 21 (JDK 21) is the latest version of Java and the third LTS Java release. Thus, it is of particular importance. Its major new features are both to the language and to the APIs. The language additions of patterns in **switch** statements and record patterns enhance the power of Java to express complex data processing procedures concisely. The API additions of Sequenced Collections bring a uniform way to deal with collections with a well-defined order in which the elements are processed, while virtual threads offer a new way for the Java platform to manage Java threads efficiently, with an API model that is entirely consistent with the existing operating system managed thread model. JDK 21 also includes several preview features, such as string templates and structured concurrency. These features are still undergoing development and may evolve further in future releases, but are available for experimentation by developers in this release.

One other point about the evolution of Java: Beginning in 2006, the process of open-sourcing Java began, which resulted in the OpenJDK project: a home for the reference implementation of Java and now also home for the ongoing evolution of Java through its formal process for proposing, refining, and integrating new features in Java, called Java Enhancement Proposals (JEP). Today, other open-source implementations of the JDK are also available. Open-sourcing further contributes to the dynamic nature of Java development. In the final analysis, Java's legacy of innovation is secure. Java remains the vibrant, nimble language that the programming world has come to expect.

The material in this book has been updated through JDK 21. Many new Java features, updates, and additions are described throughout. As the preceding discussion has highlighted, however, the history of Java programming is marked by dynamic change. You will want to review the new features in each subsequent Java release. Simply put: The evolution of Java continues!

A Culture of Innovation

Since the beginning, Java has been at the center of a culture of innovation. Its original release redefined programming for the Internet. The Java Virtual

Machine (JVM) and bytecode changed the way we think about security and portability. Portable code made the Web come alive. The Java Community Process (JCP) redefined the way that new ideas are assimilated into the language. The world of Java has never stood still for very long. JDK 21 is the latest release in Java's ongoing, dynamic history.

CHAPTER

2

An Overview of Java

As in all other computer languages, the elements of Java do not exist in isolation. Rather, they work together to form the language as a whole. However, this interrelatedness can make it difficult to describe one aspect of Java without involving several others. Often a discussion of one feature implies prior knowledge of another. For this reason, this chapter presents a quick overview of several key features of Java. The material described here will give you a foothold that will allow you to write and understand simple programs. Most of the topics discussed will be examined in greater detail in the remaining chapters of Part I.

Object-Oriented Programming

Object-oriented programming (OOP) is at the core of Java. In fact, all Java programs are to at least some extent object-oriented. OOP is so integral to Java that it is best to understand its basic principles before you begin writing even simple Java programs. Therefore, this chapter begins with a discussion of the theoretical aspects of OOP.

Two Paradigms

All computer programs consist of two elements: code and data. Furthermore, a program can be conceptually organized around its code or around its data. That is, some programs are written around "what is happening" and others are written around "who is being affected." These are the two paradigms that govern how a program is constructed. The first

way is called the *process-oriented model*. This approach characterizes a program as a series of linear steps (that is, code). The process-oriented model can be thought of as *code acting on data*. Procedural languages such as C employ this model to considerable success. However, as mentioned in Chapter 1, problems with this approach appear as programs grow larger and more complex.

To manage increasing complexity, the second approach, called *object-oriented programming*, was conceived. Object-oriented programming organizes a program around its data (that is, objects) and a set of well-defined interfaces to that data. An object-oriented program can be characterized as *data controlling access to code*. As you will see, by switching the controlling entity to data, you can achieve several organizational benefits.

Abstraction

An essential element of object-oriented programming is *abstraction*. Humans manage complexity through abstraction. For example, people do not think of a car as a set of tens of thousands of individual parts. They think of it as a well-defined object with its own unique behavior. This abstraction allows people to use a car to drive to the grocery store without being overwhelmed by the complexity of the individual parts. They can ignore the details of how the engine, transmission, and braking systems work. Instead, they are free to utilize the object as a whole.

A powerful way to manage abstraction is through the use of hierarchical classifications. This allows you to layer the semantics of complex systems, breaking them into more manageable pieces. From the outside, the car is a single object. Once inside, you see that the car consists of several subsystems: steering, brakes, sound system, seat belts, heating, cellular phone, and so on. In turn, each of these subsystems is made up of more specialized units. For instance, the system you interact with through a car's dashboard controls may consist of a media system, navigation interface, and vehicle monitoring display. The point is that you manage the complexity of the car (or any other complex system) through the use of hierarchical abstractions.

Hierarchical abstractions of complex systems can also be applied to computer programs. The data from a traditional process-oriented program

can be transformed by abstraction into its component objects. A sequence of process steps can become a collection of messages between these objects. Thus, each of these objects describes its own unique behavior. You can treat these objects as concrete entities that respond to messages telling them to *do something*. This is the essence of object-oriented programming.

Object-oriented concepts form the heart of Java just as they form the basis for human understanding. It is important that you understand how these concepts translate into programs. As you will see, object-oriented programming is a powerful and natural paradigm for creating programs that survive the inevitable changes accompanying the life cycle of any major software project, including conception, growth, and aging. For example, once you have well-defined objects and clean, reliable interfaces to those objects, you can gracefully decommission or replace parts of an older system without fear.

The Three OOP Principles

All object-oriented programming languages provide mechanisms that help you implement the object-oriented model. They are encapsulation, inheritance, and polymorphism. Let's take a look at these concepts now.

Encapsulation

Encapsulation is the mechanism that binds together code and the data it manipulates, and it keeps both safe from outside interference and misuse. One way to think about encapsulation is as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper. Access to the code and data inside the wrapper is tightly controlled through a well-defined interface. To relate this to the real world, consider the automatic transmission on an automobile. It encapsulates hundreds of bits of information about your engine, such as how much you are accelerating, the pitch of the surface you are on, and the position of the shift lever. You, as the user, have only one method of affecting this complex encapsulation: by moving the gear-shift lever. You can't affect the transmission by using the turn signal or windshield wipers, for example. Thus, the gear-shift lever is a well-defined (indeed, unique) interface to the transmission. Further, what occurs inside the transmission does not affect objects outside the transmission. For example, shifting gears

does not turn on the headlights! Because an automatic transmission is encapsulated, dozens of car manufacturers can implement one in any way they please. However, from the driver's point of view, they all work the same. This same idea can be applied to programming. The power of encapsulated code is that everyone knows how to access it and thus can use it regardless of the implementation details—and without fear of unexpected side effects.

In Java, the basis of encapsulation is the class. Although the class will be examined in great detail later in this book, the following brief discussion will be helpful now. A *class* defines the structure and behavior (data and code) that will be shared by a set of objects. Each object of a given class contains the structure and behavior defined by the class, as if it were stamped out by a mold in the shape of the class. For this reason, objects are sometimes referred to as *instances of a class*. Thus, a class is a logical construct; an object has physical reality.

When you create a class, you will specify the code and data that constitute that class. Collectively, these elements are called *members* of the class. Specifically, the data defined by the class are referred to as *member variables* or *instance variables*. The code that operates on that data is referred to as *member methods* or just *methods*. (If you are familiar with C/C++, it may help to know that what a Java programmer calls a *method*, a C/C++ programmer calls a *function*.) In properly written Java programs, the methods define how the member variables can be used. This means that the behavior and interface of a class are defined by the methods that operate on its instance data.

Since the purpose of a class is to encapsulate complexity, there are mechanisms for hiding the complexity of the implementation inside the class. Each method or variable in a class may be marked private or public. The *public* interface of a class represents everything that external users of the class need to know, or may know. The *private* methods and data can only be accessed by code that is a member of the class. Therefore, any other code that is not a member of the class cannot access a private method or variable. Since the private members of a class may only be accessed by other parts of your program through the class's public methods, you can ensure that no improper actions take place. Of course, this means that the public interface should be carefully designed not to expose too much of the inner workings of a class (see Figure 2-1).

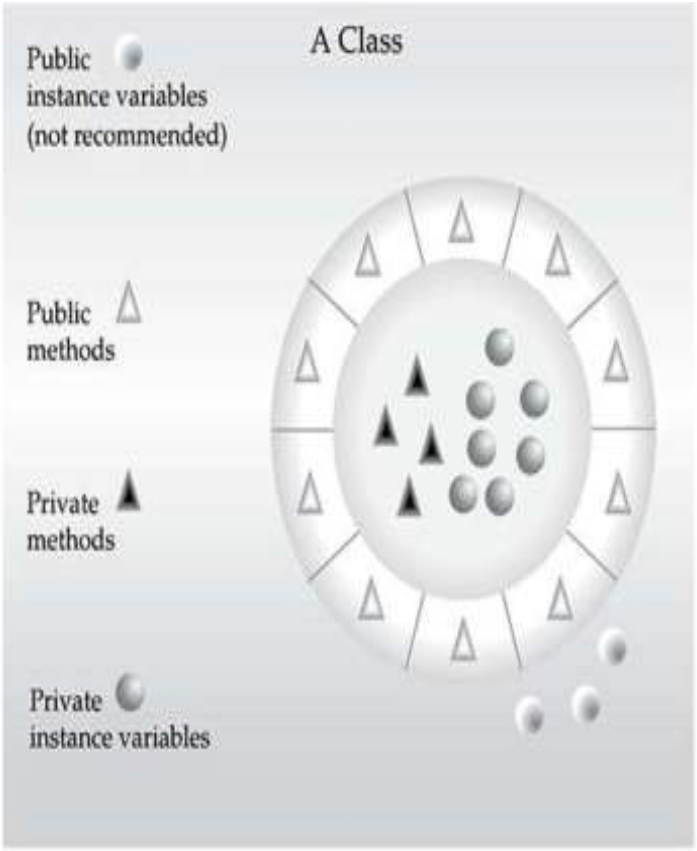


Figure 2-1 Encapsulation: public methods can be used to protect private data.

Inheritance

Inheritance is the process by which one object acquires the properties of another object. This is important because it supports the concept of hierarchical classification. As mentioned earlier, most knowledge is made manageable by hierarchical (that is, top-down) classifications. For example, a Golden Retriever is part of the classification *dog*, which in turn is part of the *mammal* class, which is under the larger class *animal*. Without the use of hierarchies, each object would need to define all of its characteristics explicitly. However, by use of inheritance, an object need only define those qualities that make it unique within its class. It can inherit its general attributes from its parent. Thus, it is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case. Let's take a closer look at this process.

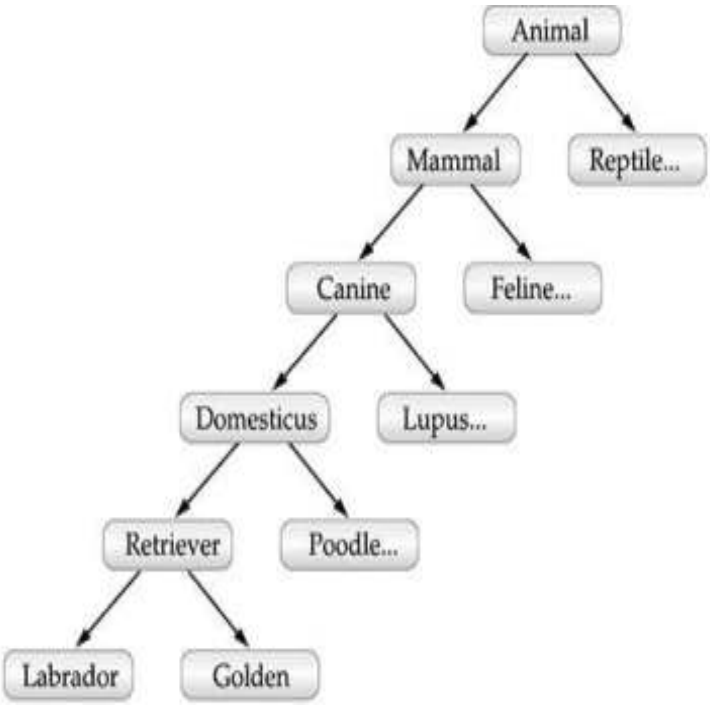
Most people naturally view the world as made up of objects that are related to each other in a hierarchical way, such as animals, mammals, and

dogs. If you wanted to describe animals in an abstract way, you would say they have some attributes, such as size, intelligence, and type of skeletal system. Animals also have certain behavioral aspects; they eat, breathe, and sleep. This description of attributes and behavior is the class definition for animals.

If you wanted to describe a more specific class of animals, such as mammals, they would have more specific attributes, such as type of teeth and mammary glands. This is known as a *subclass* of animals, where animals are referred to as mammals' *superclass*.

Since mammals are simply more precisely specified animals, they *inherit* all of the attributes from animals. A deeply inherited subclass inherits all of the attributes from each of its ancestors in the *class hierarchy*.

Inheritance interacts with encapsulation as well. If a given class encapsulates some attributes, then any subclass will have the same attributes *plus* any that it adds as part of its specialization (see Figure 2-2). This is a key concept that lets object-oriented programs grow in complexity linearly rather than geometrically. A new subclass inherits all of the attributes of all of its ancestors. It does not have unpredictable interactions with the majority of the rest of the code in the system.


Animal
Mammal
Reptile...
Canine
Feline...
Domesticus
Lupus...
Retriever
Poodle...
Labrador
Golden

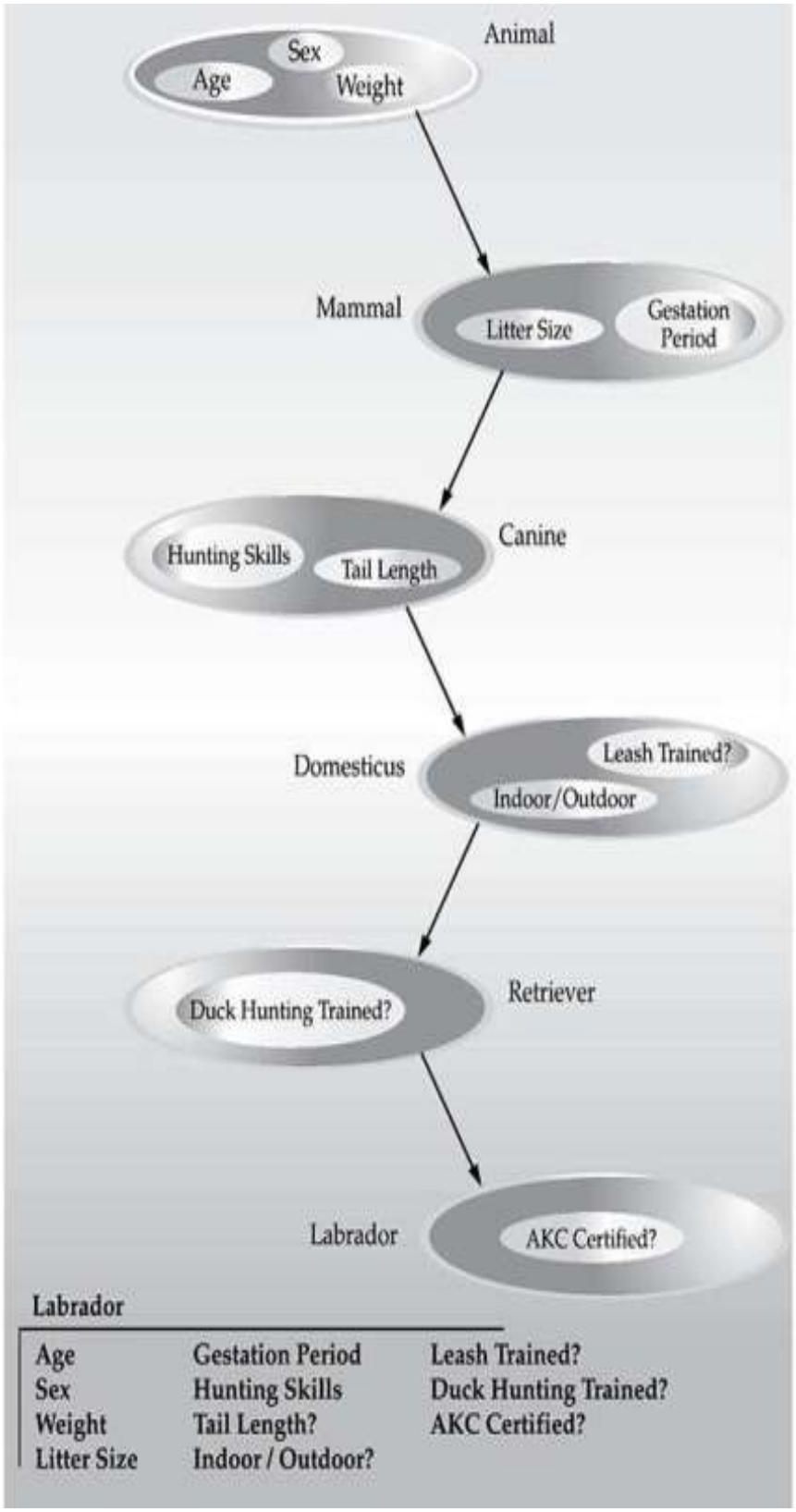


Figure 2-2 Labrador inherits the encapsulation of all its superclasses.

Polymorphism

Polymorphism (from Greek, meaning "many forms") is a feature that allows one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation. Consider a stack (which is a last-in, first-out list). You might have a program that requires three types of stacks. One stack is used for integer values, one for floating-point values, and one for characters. The algorithm that implements each stack is the same, even though the data being stored differs. In a non-object-oriented language, you would be required to create three different sets of stack routines, with each set using different names. However, because of polymorphism, in Java you can specify a general set of stack routines that all share the same names.

More generally, the concept of polymorphism is often expressed by the phrase "one interface, multiple methods." This means that it is possible to design a generic interface to a group of related activities. This helps reduce complexity by allowing the same interface to be used to specify a *general class of action*. It is the compiler's job to select the *specific action* (that is, method) as it applies to each situation. You, the programmer, do not need to make this selection manually. You need only remember and utilize the general interface.

Extending the dog analogy, a dog's sense of smell is polymorphic. If the dog smells a cat, it will bark and run after it. If the dog smells its food, it will salivate and run to its bowl. The same sense of smell is at work in both situations. The difference is what is being smelled—that is, the type of data being operated upon by the dog's nose! This same general concept can be implemented in Java as it applies to methods within a Java program.

Polymorphism, Encapsulation, and Inheritance Work Together

When properly applied, polymorphism, encapsulation, and inheritance combine to produce a programming environment that supports the development of far more robust and scalable programs than does the process-oriented model. A well-designed hierarchy of classes is the basis for reusing the code in which you have invested time and effort developing and testing. Encapsulation allows you to migrate your implementations over time without breaking the code that depends on the public interface of

your classes. Polymorphism allows you to create clean, sensible, readable, and resilient code.

Of the two real-world examples, the automobile more completely illustrates the power of object-oriented design. Dogs are fun to think about from an inheritance standpoint, but cars are more like programs. All drivers rely on inheritance to drive different types (subclasses) of vehicles. Whether the vehicle is a school bus, a Mercedes sedan, a Porsche, or the family minivan, drivers can all more or less find and operate the steering wheel, the brakes, and the accelerator. After a bit of gear grinding, most people can even manage the difference between a stick shift and an automatic, because they fundamentally understand their common superclass, the transmission.

People interface with encapsulated features on cars all the time. The brake and gas pedals hide an incredible array of complexity with an interface so simple you can operate them with your feet! The implementation of the engine, the style of brakes, and the size of the tires have no effect on how you interface with the class definition of the pedals.

The final attribute, polymorphism, is clearly reflected in the ability of car manufacturers to offer a wide array of options on basically the same vehicle. For example, you can get an antilock braking system or traditional brakes, power or rack-and-pinion steering, and a 4-, 6-, or 8-cylinder engine, or an EV. Either way, you will still press the brake pedal to stop, turn the steering wheel to change direction, and press the accelerator when you want to move. The same interface can be used to control a number of different implementations.

As you can see, it is through the application of encapsulation, inheritance, and polymorphism that the individual parts are transformed into the object known as a car. The same is also true of computer programs. By the application of object-oriented principles, the various parts of a complex program can be brought together to form a cohesive, robust, maintainable whole.

As mentioned at the start of this section, every Java program is object-oriented. Or, put more precisely, every Java program involves encapsulation, inheritance, and polymorphism. Although the short sample programs shown in the rest of this chapter and in the next few chapters may not seem to exhibit all of these features, they are nevertheless present. As you will see, many of the features supplied by Java are part of its built-in

class libraries, which do make extensive use of encapsulation, inheritance, and polymorphism.

A First Simple Program

Now that the basic object-oriented underpinning of Java has been discussed, let's look at some actual Java programs. Let's start by compiling and running the short sample program shown here. As you will see, this involves a little more work than you might imagine.

```
/*
   This is a simple Java program.
   Call this file "Example.java".
*/
class Example {
  // Your program begins with a call to main().
  public static void main(String[] args) {
    System.out.println("This is a simple Java program.");
  }
}
```

NOTE The descriptions that follow use the standard Java SE Development Kit (JDK), which is available from Oracle. (Open source versions are also available.) If you are using an integrated development environment (IDE), then you will need to follow a different procedure for compiling and executing Java programs. In this case, consult your IDE's documentation for details.

Entering the Program

For some computer languages, the name of the file that holds the source code to a program is immaterial. However, this is not the case with Java. The first thing that you must learn about Java is that the name you give to a source file is very important. For this example, the name of the source file should be **Example.java**. Let's see why.

In Java, a source file is officially called a *compilation unit*. It is a text file that contains (among other things) one or more class definitions. (For now, we will be using source files that contain only one class.) The Java compiler requires that a source file use the **.java** filename extension.

As you can see by looking at the program, the name of the class defined by the program is also **Example**. This is not a coincidence. In Java, all code must reside inside a class. By convention, the name of the main class should match the name of the file that holds the program. You should also make sure that the capitalization of the filename matches the class name. The reason for this is that Java is case-sensitive. At this point, the convention that filenames correspond to class names may seem arbitrary. However, this convention makes it easier to maintain and organize your programs. Furthermore, as you will see later in this book, in some cases, it is required.

Compiling the Program

To compile the **Example** program, execute the compiler, **javac**, specifying the name of the source file on the command line, as shown here:

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

The **javac** compiler creates a file called **Example.class** that contains the bytecode version of the program. As discussed earlier, the Java bytecode is the intermediate representation of your program that contains instructions the Java Virtual Machine will execute. Thus, the output of **javac** is not code that can be directly executed.

To actually run the program, you must use the Java application launcher called **java**. To do so, pass the class name **Example** as a command-line argument, as shown here:

```
C:\>java Example
```

When the program is run, the following output is displayed:

```
This is a simple Java program.
```

When Java source code is compiled, each individual class is put into its own output file named after the class and using the **.class** extension. This is why it is a good idea to give your Java source files the same name as the class they contain—the name of the source file will match the name of the **.class** file. When you execute **java** as just shown, you are actually

specifying the name of the class that you want to execute. It will automatically search for a file by that name that has the **.class** extension. If it finds the file, it will execute the code contained in the specified class.

NOTE Beginning with JDK 11, Java provides a way to run some types of simple programs directly from a source file, without explicitly invoking **javac**. This technique, which can be useful in some situations, is described in Appendix C. For the purposes of this book, it is assumed that you are using the normal compilation process just described.

A Closer Look at the First Sample Program

Although **Example.java** is quite short, it includes several key features that are common to all Java programs. Let's closely examine each part of the program.

The program begins with the following lines:

```
/*
   This is a simple Java program.
   Call this file "Example.java".
*/
```

This is a *comment*. Like most other programming languages, Java lets you enter a remark into a program's source file. The contents of a comment are ignored by the compiler. Instead, a comment describes or explains the operation of the program to anyone who is reading its source code. In this case, the comment describes the program and reminds you that the source file should be called **Example.java**. Of course, in real applications, comments generally explain how some part of the program works or what a specific feature does.

Java supports three styles of comments. The one shown at the top of the program is called a *multiline comment*. This type of comment must begin with /* and end with */. Anything between these two comment symbols is ignored by the compiler. As the name suggests, a multiline comment may be several lines long.

The next line of code in the program is shown here:

```
class Example {
```

This line uses the keyword **class** to declare that a new class is being defined. **Example** is an *identifier* that is the name of the class. The entire class definition, including all of its members, will be between the opening curly brace ({) and the closing curly brace (}). For the moment, don't worry too much about the details of a class except to note that in Java, all program activity occurs within one. This is one reason why all Java programs are (at least a little bit) object-oriented.

The next line in the program is the *single-line comment*, shown here:

```
// Your program begins with a call to main().
```

This is the second type of comment supported by Java. A *single-line comment* begins with a // and ends at the end of the line. As a general rule, programmers use multiline comments for longer remarks and single-line comments for brief, line-by-line descriptions. The third type of comment, a *documentation comment*, will be discussed in the "Comments" section later in this chapter.

The next line of code is shown here:

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

This line begins the **main( )** method. As the comment preceding it suggests, this is the line at which the program will begin executing. As a general rule, a Java program begins execution by calling **main( )**. The full meaning of each part of this line cannot be given now, since it involves a detailed understanding of Java's approach to encapsulation. However, since most of the examples in the first part of this book will use this line of code, let's take a brief look at each part now.

The **public** keyword is an *access modifier*, which allows the programmer to control the visibility of class members. When a class member is preceded by **public**, then that member may be accessed by code outside the class in which it is declared. (The opposite of **public** is **private**, which prevents a member from being used by code defined outside of its class.) In this case, **main( )** must be declared as **public**, since it must be called by code outside of its class when the program is started. The keyword **static** allows **main( )** to be called without having to instantiate a particular instance of the class. This is necessary since **main( )** is called by

the Java Virtual Machine before any objects are made. The keyword **void** simply tells the compiler that **main( )** does not return a value. As you will see, methods may also return values. If all this seems a bit confusing, don't worry. All of these concepts will be discussed in detail in subsequent chapters.

As stated, **main( )** is the method called when a Java application begins. Keep in mind that Java is case-sensitive. Thus, **Main** is different from **main**. It is important to understand that the Java compiler will compile classes that do not contain a **main( )** method. But **java** has no way to run these classes. So, if you had typed **Main** instead of **main**, the compiler would still compile your program. However, **java** would report an error because it would be unable to find the **main( )** method.

Any information that you need to pass to a method is received by variables specified within the set of parentheses that follow the name of the method. These variables are called *parameters*. If there are no parameters required for a given method, you still need to include the empty parentheses. In **main( )**, there is only one parameter, albeit a complicated one. **String[ ] args** declares a parameter named **args**, which is an array of instances of the class **String**. (*Arrays* are collections of objects of the same type.) Objects of type **String** store character strings. In this case, **args** receives any command-line arguments present when the program is executed. This program does not make use of this information, but other programs shown later in this book will.

The last character on the line is the {. This signals the start of **main( )**'s body. All of the code that comprises a method will occur between the method's opening curly brace and its closing curly brace.

One other point: **main( )** is simply a starting place for your program. A complex program will have dozens of classes, only one of which will need to have a **main( )** method to get things started. Furthermore, for some types of programs, you won't need **main( )** at all. However, for most of the programs shown in this book, **main( )** is required.

The next line of code is shown here. Notice that it occurs inside **main( )**.

```
System.out.println("This is a simple Java program.");
```

This line outputs the string "This is a simple Java program." followed by a new line on the screen. Output is actually accomplished by the built-in

println( ) method. In this case, **println( )** displays the string that is passed to it. As you will see, **println( )** can be used to display other types of information, too. The line begins with **System.out**. While too complicated to explain in detail at this time, briefly, **System** is a predefined class that provides access to the system, and **out** is the output stream that is connected to the console.

As you have probably guessed, console output (and input) is not used frequently in most real-world Java applications. Since most modern computing environments are graphical in nature, console I/O is used mostly for simple utility programs, demonstration programs, and server-side code. Later in this book, you will learn other ways to generate output using Java. But for now, we will continue to use the console I/O methods.

Notice that the **println( )** statement ends with a semicolon. Many statements in Java end with a semicolon. As you will see, the semicolon is an important part of the Java syntax.

The first } in the program ends **main( )**, and the last } ends the **Example** class definition.

A Second Short Program

Perhaps no other concept is more fundamental to a programming language than that of a variable. As you may know, a variable is a named memory location that may be assigned a value by your program. The value of a variable may be changed during the execution of the program. The next program shows how a variable is declared and how it is assigned a value. The program also illustrates some new aspects of console output. As the comments at the top of the program state, you should call this file **Example2.java**.

```
/*
   Here is another short example.
   Call this file "Example2.java".
*/

class Example2 {
  public static void main(String[] args) {
    int num; // this declares a variable called num

    num = 100; // this assigns num the value 100

    System.out.println("This is num: " + num);

    num = num * 2;

    System.out.print("The value of num * 2 is ");
    System.out.println(num);
  }
}
```

When you run this program, you will see the following output:

```
This is num: 100
The value of num * 2 is 200
```

Let's take a close look at why this output is generated. The first new line in the program is shown here:

```
int num; // this declares a variable called num
```

This line declares an integer variable called **num**. Java (like many other languages) requires that variables be declared before they are used.

Following is the general form of a variable declaration:

type var-name;

Here, *type* specifies the type of variable being declared, and *var-name* is the name of the variable. If you want to declare more than one variable of the specified type, you may use a comma-separated list of variable names. Java

defines several data types, including integer, character, and floating-point. The keyword **int** specifies an integer type.

In the program, the line

```
num = 100; // this assigns num the value 100
```

assigns to **num** the value 100. In Java, the assignment operator is a single equal sign.

The next line of code outputs the value of **num** preceded by the string "This is num:".

```
System.out.println("This is num: " + num);
```

In this statement, the plus sign causes the value of **num** to be appended to the string that precedes it, and then the resulting string is output. (Actually, **num** is first converted from an integer into its string equivalent and then concatenated with the string that precedes it. This process is described in detail later in this book.) This approach can be generalized. Using the + operator, you can join together as many items as you want within a single **println( )** statement.

The next line of code assigns **num** the value of **num** times 2. Like most other languages, Java uses the * operator to indicate multiplication. After this line executes, **num** will contain the value 200.

Here are the next two lines in the program:

```
System.out.print ("The value of num * 2 is ");
System.out.println (num);
```

Several new things are occurring here. First, the built-in method **print( )** is used to display the string "The value of num * 2 is ". This string is not followed by a newline. This means that when the next output is generated, it will start on the same line. The **print( )** method is just like **println( )**, except that it does not output a newline character after each call. Now look at the call to **println( )**. Notice that **num** is used by itself. Both **print( )** and **println( )** can be used to output values of any of Java's built-in types.

Two Control Statements

Although Chapter 5 will look closely at control statements, two are briefly introduced here so that they can be used in sample programs in Chapters 3 and 4. They will also help illustrate an important aspect of Java: blocks of code.

The if Statement

The Java **if** statement works much like the IF statement in any other language. It determines the flow of execution based on whether some condition is true or false. Its simplest form is shown here:

if*(condition) statement;*

Here, *condition* is a Boolean expression. (A Boolean expression is one that evaluates to either true or false.) If *condition* is true, then the statement is executed. If *condition* is false, then the statement is bypassed. Here is an example:

```
if(num < 100) System.out.println("num is less than 100");
```

In this case, if **num** contains a value that is less than 100, the conditional expression is true, and **println( )** will execute. If **num** contains a value greater than or equal to 100, then the **println( )** method is bypassed.

As you will see in Chapter 4, Java defines a full complement of relational operators that may be used in a conditional expression. Here are a few:

Operator	Meaning
<	Less than
>	Greater than
==	Equal to

Notice that the test for equality is the double equal sign.

Here is a program that illustrates the **if** statement:

```
/*
  Demonstrate the if.

  Call this file "IfSample.java".
*/
class IfSample {
  public static void main(String[] args) {
    int x, y;

    x = 10;
    y = 20;

    if(x < y) System.out.println("x is less than y");

    x = x * 2;
    if(x == y) System.out.println("x now equal to y");

    x = x * 2;
    if(x > y) System.out.println("x now greater than y");

    // this won't display anything
    if(x == y) System.out.println("you won't see this");
  }
}
```

The output generated by this program is shown here:

```
x is less than y
x now equal to y
x now greater than y
```

Notice one other thing in this program. The line

```
int x, y;
```

declares two variables, **x** and **y**, by use of a comma-separated list.

The for Loop

Loop statements are an important part of nearly any programming language because they provide a way to repeatedly execute some task. As you will see in Chapter 5, Java supplies a powerful assortment of loop constructs. Perhaps the most versatile is the **for** loop. The simplest form of the **for** loop is shown here:

for*(initialization; condition; iteration) statement;*

In its most common form, the *initialization* portion of the loop sets a loop control variable to an initial value. The *condition* is a Boolean expression that tests the loop control variable. If the outcome of that test is true, *statement* executes and the **for** loop continues to iterate. If it is false, the loop terminates. The *iteration* expression determines how the loop control variable is changed each time the loop iterates. Here is a short program that illustrates the **for** loop:

```
/*
  Demonstrate the for loop.

  Call this file "ForTest.java".
*/
class ForTest {
  public static void main(String[] args) {
    int x;

    for(x = 0; x<10; x = x+1)
      System.out.println("This is x: " + x);
  }
}
```

This program generates the following output:

```
This is x: 0
This is x: 1
This is x: 2
This is x: 3
This is x: 4
This is x: 5
This is x: 6
This is x: 7
This is x: 8
This is x: 9
```

In this example, **x** is the loop control variable. It is initialized to zero in the initialization portion of the **for**. At the start of each iteration (including the first one), the conditional test **x < 10** is performed. If the outcome of this test is true, the **println( )** statement is executed, and then the iteration portion of the loop is executed, which increases **x** by 1. This process continues until the conditional test is false.

As a point of interest, in professionally written Java programs you will almost never see the iteration portion of the loop written as shown in the preceding program. That is, you will seldom see statements like this:

```
x = x + 1;
```

The reason is that Java includes a special increment operator that performs this operation more efficiently. The increment operator is ++. (That is, two plus signs back to back.) The increment operator increases its operand by one. By use of the increment operator, the preceding statement can be written like this:

```
x++;
```

Thus, the **for** in the preceding program will usually be written like this:

```
for(x = 0; x<10; x++)
```

You might want to try this. As you will see, the loop still runs exactly the same as it did before.

Java also provides a decrement operator, which is specified as – –. This operator decreases its operand by one.

Using Blocks of Code

Java allows two or more statements to be grouped into *blocks of code*, also called *code blocks*. This is done by enclosing the statements between opening and closing curly braces. Once a block of code has been created, it becomes a logical unit that can be used any place that a single statement can. For example, a block can be a target for Java's **if** and **for** statements. Consider this **if** statement:

```
if(x < y) { // begin a block
  x = y;
  y = 0;
} // end of block
```

Here, if **x** is less than **y**, then both statements inside the block will be executed. Thus, the two statements inside the block form a logical unit, and one statement cannot execute without the other also executing. The key point here is that whenever you need to logically link two or more statements, you do so by creating a block.

Let's look at another example. The following program uses a block of code as the target of a **for** loop:

```
/*
  Demonstrate a block of code.

  Call this file "BlockTest.java"
*/
class BlockTest {
  public static void main(String[] args) {
    int x, y;

    y = 20;

    // the target of this loop is a block
    for(x = 0; x<10; x++) {
      System.out.println("This is x: " + x);
      System.out.println("This is y: " + y);
      y = y - 2;
    }
  }
}
```

The output generated by this program is shown here:

```
This is x: 0
This is y: 20
This is x: 1
This is y: 18
This is x: 2
This is y: 16
This is x: 3
This is y: 14
This is x: 4
This is y: 12
This is x: 5
This is y: 10
This is x: 6
This is y: 8
This is x: 7
This is y: 6
This is x: 8
This is y: 4
This is x: 9
This is y: 2
```

In this case, the target of the **for** loop is a block of code and not just a single statement. Thus, each time the loop iterates, the three statements inside the block will be executed. This fact is, of course, evidenced by the output generated by the program.

As you will see later in this book, blocks of code have additional properties and uses. However, the main reason for their existence is to create logically inseparable units of code.

Lexical Issues

Now that you have seen several short Java programs, it is time to more formally describe the atomic elements of Java. Java programs are a collection of whitespace, identifiers, literals, comments, operators, separators, and keywords. The operators are described in the next chapter. The others are described next.

Whitespace

Java is a free-form language. This means that you do not need to follow any special indentation rules. For instance, the **Example** program could have been written all on one line or in any other strange way you felt like typing it, as long as there was at least one whitespace character between each token that was not already delineated by an operator or separator. In Java, whitespace includes a space, tab, newline, or form feed.

Identifiers

Identifiers are used to name things, such as classes, variables, and methods. An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters. (The dollar-sign character is not intended for general use.) They must not begin with a number, lest they be confused with a numeric literal. Again, Java is case-sensitive, so **VALUE** is a different identifier than **Value**. Some examples of valid identifiers are

AvgTemp	count	a4	$test	this_is_ok

Invalid identifier names include these:

2count	high-temp	Not/ok

NOTE Beginning with JDK 9, the underscore cannot be used by itself as an identifier.

Literals

A constant value in Java is created by using a *literal* representation of it. For example, here are some literals:

100	98.6	'X'	"This is a test"

Left to right, the first literal specifies an integer, the next is a floating-point value, the third is a character constant, and the last is a string. A literal can be used anywhere a value of its type is allowed.

Comments

As mentioned, there are three types of comments defined by Java. You have already seen two: single-line and multiline. The third type is called a *documentation comment*. This type of comment is used to produce an HTML file that documents your program. The documentation comment begins with a /** and ends with a */. Documentation comments are explained in Appendix A.

Separators

In Java, there are a few characters that are used as separators. The most commonly used separator in Java is the semicolon. As you have seen, it is often used to terminate statements. The separators are shown in the following table:

Symbol	Name	Purpose
( )	Parentheses	Used to contain lists of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types.
{ }	Braces	Used to contain the values of automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes.
[ ]	Brackets	Used to declare array types. Also used when dereferencing array values.
;	Semicolon	Terminates statements.
,	Comma	Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside a **for** statement.
.	Period	Used to separate package names from subpackages and classes. Also used to separate a variable or method from a reference variable.
::	Colons	Used to create a method or constructor reference.
...	Ellipsis	Indicates a variable-arity parameter.
@	At-sign	Begins an annotation.

The Java Keywords

There are 68 keywords currently defined in the Java language (see Table 2-1). These keywords, combined with the syntax of the operators and separators, form the foundation of the Java language. The Java language limits the uses of its keywords and divides them into two categories: 51 *reserved words* and 17 *contextual keywords*. Reserved words cannot be used as identifiers, for example, variable, class, or method names. Contextual keywords have special meanings for the specific language feature that defines them but can be used elsewhere. Keywords support features added to Java over the past few years. Ten relate to modules: **exports**, **module**, **open**, **opens**, **provides**, **requires**, **to**, **transitive**, **uses**, and **with**. Records are declared by **record**; sealed classes and interfaces use **sealed**, **non-sealed**, and **permits**; **yield** and **when** are used by the enhanced **switch**; and **var** supports local variable type inference. Because keywords are context-sensitive, existing programs were unaffected by their addition. Also, beginning with JDK 9, an underscore by itself is considered a keyword in order to prevent its use as the name of something in your program. Since JDK 17, **strictfp** has been rendered obsolete because it has no effect.

abstract	assert	boolean	break	byte	case
catch	char	class	const	continue	default
do	double	else	enum	exports	extends
final	finally	float	for	goto	if
implements	import	instanceof	int	interface	long
module	native	new	non-sealed	open	opens
package	permits	private	protected	provides	public
record	requires	return	sealed	short	static
strictfp	super	switch	synchronized	this	throw
throws	to	transient	transitive	try	uses
var	void	volatile	when	while	with
yield	_				

Table 2-1 Java Keywords

The keywords **const** and **goto** are reserved but not used. In the early days of Java, several other keywords were reserved for possible future use. However, the current specification for Java defines only the keywords shown in Table 2-1.

In addition to the keywords, Java reserves three other names that have been part of Java from the start: **true**, **false**, and **null**. These are values defined by Java. You may not use these words for the names of variables, classes, and so on.

The Java Class Libraries

The sample programs shown in this chapter make use of two of Java's built-in methods: **println( )** and **print( )**. As mentioned, these methods are available through **System.out.System** is a class predefined by Java that is automatically included in your programs. In the larger view, the Java environment relies on several built-in class libraries that contain many built-in methods that provide support for such things as I/O, string handling, networking, and graphics. The standard classes also provide support for a graphical user interface (GUI). Thus, Java as a totality is a combination of the Java language itself plus its standard classes. As you will see, the class libraries provide much of the functionality that comes with Java. Indeed, part of becoming a Java programmer is learning to use the standard Java classes. Throughout Part I of this book, various elements of the standard library classes and methods are described as needed. In Part II, several class libraries are described in detail.

CHAPTER

3

Data Types, Variables, and Arrays

This chapter examines three of Java's most fundamental elements: data types, variables, and arrays. As with all modern programming languages, Java supports several types of data. You may use these types to declare variables and to create arrays. As you will see, Java's approach to these items is clean, efficient, and cohesive.

Java Is a Strongly Typed Language

It is important to state at the outset that Java is a strongly typed language. Indeed, part of Java's safety and robustness comes from this fact. Let's see what this means. First, every variable has a type, every expression has a type, and every type is strictly defined. Second, all assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility. There are no automatic coercions or conversions of conflicting types as in some languages. The Java compiler checks all expressions and parameters to ensure that the types are compatible. Any type mismatches are errors that must be corrected before the compiler will finish compiling the class.

The Primitive Types

Java defines eight *primitive* types of data: **byte**, **short**, **int**, **long**, **char**, **float**, **double**, and **boolean**. The primitive types are also commonly

referred to as *simple* types, and both terms will be used in this book. These can be put in four groups:

- **Integers** This group includes **byte**, **short**, **int**, and **long**, which are for whole-valued signed numbers.
- **Floating-point numbers** This group includes **float** and **double**, which represent numbers with fractional precision.
- **Characters** This group includes **char**, which represents symbols in a character set, like letters and numbers.
- **Boolean** This group includes **boolean**, which is a special type for representing true/false values.

You can use these types as-is or to construct arrays or your own class types. Thus, they form the basis for all other types of data that you can create.

The primitive types represent single values—not complex objects. Although Java is otherwise completely object-oriented, the primitive types are not. They are analogous to the simple types found in most other non-object-oriented languages. The reason for this is efficiency. Making the primitive types into objects would have degraded performance too much.

The primitive types are defined to have an explicit range and mathematical behavior. Languages such as C and C++ allow the size of an integer to vary based upon the dictates of the execution environment. However, Java is different. Because of Java's portability requirement, all data types have a strictly defined range. For example, an **int** is always 32 bits, regardless of the particular platform. This allows programs to be written that are guaranteed to run *without porting* on any machine architecture. While strictly specifying the size of an integer may cause a small loss of performance in some environments, it is necessary in order to achieve portability.

Let's look at each type of data in turn.

Integers

Java defines four integer types: **byte**, **short**, **int**, and **long**. All of these are signed, positive and negative values. Java does not support unsigned, positive-only integers. Many other computer languages support both signed

and unsigned integers. However, Java's designers felt that unsigned integers were unnecessary. Specifically, they felt that the concept of *unsigned* was used mostly to specify the behavior of the *high-order bit*, which defines the *sign* of an integer value. As you will see in Chapter 4, Java manages the meaning of the high-order bit differently, by adding a special "unsigned right shift" operator. Thus, the need for an unsigned integer type was eliminated.

The *width* of an integer type should not be thought of as the amount of storage it consumes, but rather as the *behavior* it defines for variables and expressions of that type. The Java run-time environment is free to use whatever size it wants, as long as the types behave as you declared them. The width and ranges of these integer types vary widely, as shown in this table:

Name	Width	Range
long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	–2,147,483,648 to 2,147,483,647
short	16	–32,768 to 32,767
byte	8	–128 to 127

Let's look at each type of integer.

byte

The smallest integer type is **byte**. This is a signed 8-bit type that has a range from –128 to 127. Variables of type **byte** are especially useful when you're working with a stream of data from a network or file. They are also useful when you're working with raw binary data that may not be directly compatible with Java's other built-in types.

Byte variables are declared by use of the **byte** keyword. For example, the following declares two **byte** variables called **b** and **c**:

```
byte b, c;
```

short

short is a signed 16-bit type. It has a range from –32,768 to 32,767. It is probably the least-used Java type. Here are some examples of **short** variable declarations:

```
short s;
short t;
```

int

The most commonly used integer type is **int**. It is a signed 32-bit type that has a range from –2,147,483,648 to 2,147,483,647. In addition to other uses, variables of type **int** are commonly employed to control loops and to index arrays. Although you might think that using a **byte** or **short** would be more efficient than using an **int** in situations in which the larger range of an **int** is not needed, this may not be the case. The reason is that when **byte** and **short** values are used in an expression, they are *promoted* to **int** when the expression is evaluated. (Type promotion is described later in this chapter.) Therefore, **int** is often the best choice when an integer is needed.

long

long is a signed 64-bit type and is useful for those occasions where an **int** type is not large enough to hold the desired value. The range of a **long** is quite large. This makes it useful when big, whole numbers are needed. For example, here is a program that computes the number of miles that light will travel in a specified number of days:

```
// Compute distance light travels using long variables.
class Light {
  public static void main(String[] args) {
    int lightspeed;
    long days;
    long seconds;
    long distance;

    // approximate speed of light in miles per second
    lightspeed = 186000;

    days = 1000; // specify number of days here

    seconds = days * 24 * 60 * 60; // convert to seconds

    distance = lightspeed * seconds; // compute distance

    System.out.print("In " + days);
    System.out.print(" days light will travel about ");
    System.out.println(distance + " miles.");
  }
}
```

This program generates the following output:

```
In 1000 days light will travel about 16070400000000 miles.
```

Clearly, the result could not have been held in an **int** variable.

Floating-Point Types

Floating-point numbers, also known as *real* numbers, are used when evaluating expressions that require fractional precision. For example, calculations such as square root, or transcendentals such as sine and cosine, result in a value whose precision requires a floating-point type. Java implements the standard (IEEE 754) set of floating-point types and operators. There are two kinds of floating-point types, **float** and **double**, which represent single- and double-precision numbers, respectively. Their width and ranges are shown here:

Name	Width in Bits	Approximate Range
double	64	4.9e–324 to 1.8e+308
float	32	1.4e–045 to 3.4e+038

Each of these floating-point types is examined next.

float

The type **float** specifies a *single-precision* value that uses 32 bits of storage. Single precision is faster on some processors and takes half as much space as double precision, but will become imprecise when the values are either very large or very small. Variables of type **float** are useful when you need a fractional component but don't require a large degree of precision. For example, **float** can be useful when representing dollars and cents.

Here are some sample **float** variable declarations:

```
float hightemp, lowtemp;
```

double

Double precision, as denoted by the **double** keyword, uses 64 bits to store a value. Double precision is actually faster than single precision on some modern processors that have been optimized for high-speed mathematical calculations. All transcendental math functions, such as **sin( )**, **cos( )**, and **sqrt( )**, return **double** values. When you need to maintain accuracy over many iterative calculations, or are manipulating large-valued numbers, **double** is the best choice.

Here is a short program that uses **double** variables to compute the area of a circle:

```
// Compute the area of a circle.
class Area {
  public static void main(String[] args) {
    double pi, r, a;
```

```
    r = 10.8; // radius of circle
    pi = 3.1416; // pi, approximately
    a = pi * r * r; // compute area

   System.out.println("Area of circle is " + a);
  }
}
```

Characters

In Java, the data type used to store characters is **char**. A key point to understand is that Java uses *Unicode* to represent characters. Unicode defines a fully international character set that can represent all of the characters found in all human languages. It is a unification of dozens of character sets, such as Latin, Greek, Arabic, Cyrillic, Hebrew, Katakana, Hangul, and many more. At the time of Java's creation, Unicode required 16 bits. Thus, in Java **char** is a 16-bit type. The range of a **char** is 0 to 65,535. There are no negative **char**s. The standard set of characters known as ASCII still ranges from 0 to 127 as always, and the extended 8-bit character set, ISO-Latin-1, ranges from 0 to 255. Since Java is designed to allow programs to be written for worldwide use, it makes sense that it would use Unicode to represent characters. Of course, the use of Unicode is somewhat inefficient for languages such as English, German, Spanish, and French, whose characters can easily be contained within 8 bits. But such is the price that must be paid for global portability.

NOTE More information about Unicode can be found at https://home.unicode.org.

Here is a program that demonstrates **char** variables:

```
// Demonstrate char data type.
class CharDemo {
  public static void main(String[] args) {
    char ch1, ch2;

    ch1 = 88; // code for X
    ch2 = 'Y';

    System.out.print("ch1 and ch2: ");
    System.out.println(ch1 + " " + ch2);
  }
}
```

This program displays the following output:

```
ch1 and ch2: X Y
```

Notice that **ch1** is assigned the value 88, which is the ASCII (and Unicode) value that corresponds to the letter *X*. As mentioned, the ASCII character set occupies the first 127 values in the Unicode character set. For this reason, all the "old tricks" that you may have used with characters in other languages will work in Java, too.

Although **char** is designed to hold Unicode characters, it can also be used as an integer type on which you can perform arithmetic operations. For example, you can add two characters together or increment the value of a character variable. Consider the following program:

```
// char variables behave like integers.
class CharDemo2 {
  public static void main(String[] args) {
    char ch1;

    ch1 = 'X';
    System.out.println("ch1 contains " + ch1);

    ch1++; // increment ch1
    System.out.println("ch1 is now " + ch1);
  }
}
```

The output generated by this program is shown here:

```
ch1 contains X
ch1 is now Y
```

In the program, **ch1** is first given the value *X*. Next, **ch1** is incremented. This results in **ch1** containing *Y*, the next character in the ASCII (and Unicode) sequence.

NOTE In the formal specification for Java, **char** is referred to as an *integral type*, which means that it is in the same general category as **int**, **short**, **long**, and **byte**. However, because its principal use is for representing Unicode characters, **char** is commonly considered to be in a category of its own.

Booleans

Java has a primitive type, called **boolean**, for logical values. It can have only one of two possible values, **true** or **false**. This is the type returned by all relational operators, as in the case of **a < b**. **boolean** is also the type *required* by the conditional expressions that govern the control statements such as **if** and **for**.

Here is a program that demonstrates the **boolean** type:

```
// Demonstrate boolean values.
class BoolTest {
  public static void main(String[] args) {
    boolean b;

    b = false;
    System.out.println("b is " + b);
    b = true;
    System.out.println("b is " + b);

    // a boolean value can control the if statement
    if(b) System.out.println("This is executed.");

    b = false;
    if(b) System.out.println("This is not executed.");
```

```
    // outcome of a relational operator is a boolean value
    System.out.println("10 > 9 is " + (10 > 9));
  }
}
```

The output generated by this program is shown here:

```
b is false
b is true
This is executed.
10 > 9 is true
```

There are three interesting things to notice about this program. First, as you can see, when a **boolean** value is output by **println( )**, "true" or "false" is displayed. Second, the value of a **boolean** variable is sufficient, by itself, to control the **if** statement. There is no need to write an **if** statement like this:

```
if(b == true) …
```

Third, the outcome of a relational operator, such as <, is a **boolean** value. This is why the expression **10>9** displays the value "true." Further, the extra set of parentheses around **10>9** is necessary because the + operator has a higher precedence than the >.

A Closer Look at Literals

Literals were mentioned briefly in Chapter 2. Now that the built-in types have been formally described, let's take a closer look at them.

Integer Literals

Integers are probably the most commonly used type in the typical program. Any whole number value is an integer literal. Examples are 1, 2, 3, and 42. These are all decimal values, meaning they are describing a base 10 number. Two other bases that can be used in integer literals are *octal* (base eight) and *hexadecimal* (base 16). Octal values are denoted in Java by a leading zero. Normal decimal numbers cannot have a leading zero. Thus, the seemingly valid value 09 will produce an error from the compiler, since

9 is outside of octal's 0 to 7 range. A more common base for numbers used by programmers is hexadecimal, which matches cleanly with modulo 8 word sizes, such as 8, 16, 32, and 64 bits. You signify a hexadecimal constant with a leading zero-x (**0x** or **0X**). The range of a hexadecimal digit is 0 to 15, so *A* through *F* (or *a* through *f*) are substituted for 10 through 15.

Integer literals create an **int** value, which in Java is a 32-bit integer value. Since Java is strongly typed, you might be wondering how it is possible to assign an integer literal to one of Java's other integer types, such as **byte** or **long**, without causing a type mismatch error. Fortunately, such situations are easily handled. When a literal value is assigned to a **byte** or **short** variable, no error is generated if the literal value is within the range of the target type. An integer literal can always be assigned to a **long** variable. However, to specify a **long** literal, you will need to explicitly tell the compiler that the literal value is of type **long**. You do this by appending an upper- or lowercase *L* to the literal. For example, 0x7fffffffffffffL or 9223372036854775807L, is the largest **long**. An integer can also be assigned to a **char** as long as it is within range.

You can also specify integer literals using binary. To do so, prefix the value with **0b** or **0B**. For example, this specifies the decimal value 10 using a binary literal:

```
int x = 0b1010;
```

Among other uses, the addition of binary literals makes it easier to enter values used as bitmasks. In such a case, the decimal (or hexadecimal) representation of the value does not visually convey its meaning relative to its use. The binary literal does.

You can embed one or more underscores in an integer literal. Doing so makes it easier to read large integer literals. When the literal is compiled, the underscores are discarded. For example, given

```
int x = 123_456_789;
```

the value given to **x** will be 123,456,789. The underscores will be ignored. Underscores can only be used to separate digits. They cannot come at the beginning or the end of a literal. It is, however, permissible for more than one underscore to be used between two digits. For example, this is valid:

```
int x = 123___456___789;
```

The use of underscores in an integer literal is especially useful when encoding such things as telephone numbers, customer ID numbers, part numbers, and so on. They are also useful for providing visual groupings when specifying binary literals. For example, binary values are often visually grouped in four-digits units, as shown here:

```
int x = 0b1101_0101_0001_1010;
```

Floating-Point Literals

Floating-point numbers represent decimal values with a fractional component. They can be expressed in either standard or scientific notation. *Standard notation* consists of a whole number component followed by a decimal point followed by a fractional component. For example, 2.0, 3.14159, and 0.6667 represent valid standard-notation floating-point numbers. *Scientific notation* uses a standard-notation floating-point number plus a suffix that specifies a power of 10 by which the number is to be multiplied. The exponent is indicated by an *E* or *e* followed by a decimal number, which can be positive or negative. Examples include 6.022E23, 314159E–05, and 2e+100.

Floating-point literals in Java default to **double** precision. To specify a **float** literal, you must append an *F* or *f* to the constant. You can also explicitly specify a **double** literal by appending a *D* or *d*. Doing so is, of course, redundant. The default **double** type consumes 64 bits of storage, while the smaller **float** type requires only 32 bits.

Hexadecimal floating-point literals are also supported, but they are rarely used. They must be in a form similar to scientific notation, but a **P** or **p**, rather than an **E** or **e**, is used. For example, 0x12.2P2 is a valid floating-point literal. The value following the **P**, called the *binary exponent*, indicates the power-of-two by which the number is multiplied. Therefore, **0x12.2P2** represents 72.5.

You can embed one or more underscores in a floating-point literal. This feature works the same as it does for integer literals, which were just described. Its purpose is to make it easier to read large floating-point

literals. When the literal is compiled, the underscores are discarded. For example, given

```
double num = 9_423_497_862.0;
```

the value given to **num** will be 9,423,497,862.0. The underscores will be ignored. As is the case with integer literals, underscores can only be used to separate digits. They cannot come at the beginning or the end of a literal. It is, however, permissible for more than one underscore to be used between two digits. It is also permissible to use underscores in the fractional portion of the number. For example,

```
double num = 9_423_497.1_0_9;
```

is legal. In this case, the fractional part is **.109**.

Boolean Literals

Boolean literals are simple. There are only two logical values that a **boolean** value can have, **true** and **false**. The values of **true** and **false** do not convert into any numerical representation. The **true** literal in Java does not equal 1, nor does the **false** literal equal 0. In Java, the Boolean literals can only be assigned to variables declared as **boolean** or used in expressions with Boolean operators.

Character Literals

Characters in Java are indices into the Unicode character set. They are 16-bit values that can be converted into integers and manipulated with the integer operators, such as the addition and subtraction operators. A literal character is represented inside a pair of single quotes. All of the visible ASCII characters can be directly entered inside the quotes, such as '*a*', '*z*', and '@'. For characters that are impossible to enter directly, there are several escape sequences that allow you to enter the character you need, such as ' \' ' for the single-quote character itself and ' **n**' for the newline character. There is also a mechanism for directly entering the value of a character in octal or hexadecimal. For octal notation, use the backslash followed by the three-digit number. For example, ' *141*' is the letter '*a*'. For

hexadecimal, you enter a backslash-u (**u**), then exactly four hexadecimal digits. For example, ' *u0061*' is the ISO-Latin-1 '*a*' because the top byte is zero. ' *ua432* ' is a Japanese Katakana character. Table 3-1 shows the character escape sequences.

Escape Sequence	Description
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal Unicode character (xxxx)
\'	Single quote
\"	Double quote
\\	Backslash
\r	Carriage return
\n	New line (also known as line feed)
\f	Form feed
\t	Tab
\b	Backspace
\s	Space (added by JDK 15)
\endofline	Continue line (applies only to text blocks; added by JDK 15)

Table 3-1 Character Escape Sequences

String Literals

String literals in Java are specified like they are in most other languages—by enclosing a sequence of characters between a pair of double quotes. Examples of string literals are

```
"Hello World"
"two\nlines"
" \"This is in quotes\""
```

The escape sequences and octal/hexadecimal notations that were defined for character literals work the same way inside of string literals. One

important thing to note about Java string literals is that they must begin and end on the same line, even if the line wraps. For string literals there is no line-continuation escape sequence as there is in some other languages. (It is useful to point out that beginning with JDK 15, Java added a feature called a *text block*, which gives you more control and flexibility when multiple lines of text are needed. See Chapter 17.)

NOTE As you may know, in some other languages strings are implemented as arrays of characters. However, this is not the case in Java. Strings are actually object types. As you will see later in this book, because Java implements strings as objects, Java includes extensive string-handling capabilities that are both powerful and easy to use.

Variables

The variable is the basic unit of storage in a Java program. A variable is defined by the combination of an identifier, a type, and an optional initializer. In addition, all variables have a scope, which defines their visibility, and a lifetime. These elements are examined next.

Declaring a Variable

In Java, all variables must be declared before they can be used. The basic form of a variable declaration is shown here:

type identifier [= *value*][, *identifier* [= *value*] …];

Here, *type* is one of Java's atomic types, or the name of a class or interface. (Class and interface types are discussed later in Part I of this book.) The *identifier* is the name of the variable. You can initialize the variable by specifying an equal sign and a value. Keep in mind that the initialization expression must result in a value of the same (or compatible) type as that specified for the variable. To declare more than one variable of the specified type, use a comma-separated list.

Here are several examples of variable declarations of various types. Note that some include an initialization.

```
int a, b, c;            // declares three ints, a, b, and c.
int d = 3, e, f = 5;    // declares three more ints, initializing
                        // d and f.
byte z = 22;            // initializes z.
double pi = 3.14159;    // declares an approximation of pi.
char x = 'x';           // the variable x has the value 'x'.
```

The identifiers that you choose have nothing intrinsic in their names that indicates their type. Java allows any properly formed identifier to have any declared type.

Dynamic Initialization

Although the preceding examples have used only constants as initializers, Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared.

For example, here is a short program that computes the length of the hypotenuse of a right triangle given the lengths of its two opposing sides:

```
// Demonstrate dynamic initialization.
class DynInit {
  public static void main(String[] args) {
    double a = 3.0, b = 4.0;

    // c is dynamically initialized
    double c = Math.sqrt(a * a + b * b);

    System.out.println("Hypotenuse is " + c);
  }
}
```

Here, three local variables—**a**, **b**, and **c**—are declared. The first two, **a** and **b**, are initialized by constants. However, **c** is initialized dynamically to the length of the hypotenuse (using the Pythagorean theorem). The program uses another of Java's built-in methods, **sqrt( )**, which is a member of the **Math** class, to compute the square root of its argument. The key point here is that the initialization expression may use any element valid at the time of the initialization, including calls to methods, other variables, or literals.

The Scope and Lifetime of Variables

So far, all of the variables used have been declared at the start of the **main( )** method. However, Java allows variables to be declared within any block. As explained in Chapter 2, a block is begun with an opening curly brace and ended by a closing curly brace. A block defines a *scope*. Thus, each time you start a new block, you are creating a new scope. A scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects.

It is not uncommon to think in terms of two general categories of scopes: global and local. However, these traditional scopes do not fit well with Java's strict, object-oriented model. While it is possible to create what amounts to being a global scope, it is by far the exception, not the rule. In Java, the two major scopes are those defined by a class and those defined by a method. Even this distinction is somewhat artificial. However, since the class scope has several unique properties and attributes that do not apply to the scope defined by a method, this distinction makes some sense. Because of the differences, a discussion of class scope (and variables declared within it) is deferred until Chapter 6, when classes are described. For now, we will only examine the scopes defined by or within a method.

The scope defined by a method begins with its opening curly brace. However, if that method has parameters, they too are included within the method's scope. A method's scope ends with its closing curly brace. This block of code is called the *method body*.

As a general rule, variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope. Thus, when you declare a variable within a scope, you are localizing that variable and protecting it from unauthorized access and/or modification. Indeed, the scope rules provide the foundation for encapsulation. A variable declared within a block is called a *local variable*.

Scopes can be nested. For example, each time you create a block of code, you are creating a new, nested scope. When this occurs, the outer scope encloses the inner scope. This means that objects declared in the outer scope will be visible to code within the inner scope. However, the reverse is not true. Objects declared within the inner scope will not be visible outside it.

To understand the effect of nested scopes, consider the following program:

```
// Demonstrate block scope.
class Scope {
  public static void main(String[] args) {
    int x; // known to all code within main

    x = 10;
    if(x == 10) { // start new scope
      int y = 20; // known only to this block

      // x and y both known here.
      System.out.println("x and y: " + x + " " + y);
      x = y * 2;
    }
    // y = 100; // Error! y not known here

    // x is still known here.
    System.out.println("x is " + x);
  }
}
```

As the comments indicate, the variable **x** is declared at the start of **main( )**'s scope and is accessible to all subsequent code within **main( )**. Within the **if** block, **y** is declared. Since a block defines a scope, **y** is only visible to other code within its block. This is why outside of its block, the line **y = 100**; is commented out. If you remove the leading comment symbol, a compile-time error will occur, because **y** is not visible outside of its block. Within the **if** block, **x** can be used because code within a block (that is, a nested scope) has access to variables declared by an enclosing scope.

Within a block, variables can be declared at any point but are valid only after they are declared. Thus, if you define a variable at the start of a method, it is available to all of the code within that method. Conversely, if you declare a variable at the end of a block, it is effectively useless, because no code will have access to it. For example, this fragment is invalid because **count** cannot be used prior to its declaration:

```
// This fragment is wrong!
count = 100; // oops! cannot use count before it is declared!
int count;
```

Here is another important point to remember: variables are created when their scope is entered, and destroyed when their scope is left. This means that a variable will not hold its value once it has gone out of scope. Therefore, variables declared within a method will not hold their values between calls to that method. Also, a variable declared within a block will lose its value when the block is left. Thus, the lifetime of a variable is confined to its scope.

If a variable declaration includes an initializer, then that variable will be reinitialized each time the block in which it is declared is entered. For example, consider the next program:

```
// Demonstrate lifetime of a variable.
class LifeTime {
  public static void main(String[] args) {
    int x;

    for(x = 0; x < 3; x++) {
      int y = -1; // y is initialized each time block is entered
      System.out.println("y is: " + y); // this always prints -1
      y = 100;
      System.out.println("y is now: " + y);
    }
  }
}
```

The output generated by this program is shown here:

```
y is: -1
y is now: 100
y is: -1
y is now: 100
y is: -1
y is now: 100
```

As you can see, **y** is reinitialized to –1 each time the inner **for** loop is entered. Even though it is subsequently assigned the value 100, this value is

lost.

One last point: Although blocks can be nested, you cannot declare a variable to have the same name as one in an outer scope. For example, the following program is illegal:

```
// This program will not compile
class ScopeErr {
  public static void main(String[] args) {
    int bar = 1;
    {                  // creates a new scope
      int bar = 2;  // Compile-time error - bar already defined!
    }
  }
}
```

Type Conversion and Casting

If you have previous programming experience, then you already know that it is fairly common to assign a value of one type to a variable of another type. If the two types are compatible, then Java will perform the conversion automatically. For example, it is always possible to assign an **int** value to a **long** variable. However, not all types are compatible, and thus, not all type conversions are implicitly allowed. For instance, there is no automatic conversion defined from **double** to **byte**. Fortunately, it is still possible to obtain a conversion between incompatible types. To do so, you must use a *cast*, which performs an explicit conversion between incompatible types. Let's look at both automatic type conversions and casting.

Java's Automatic Conversions

When one type of data is assigned to another type of variable, an *automatic type conversion* will take place if the following two conditions are met:

- The two types are compatible.
- The destination type is larger than the source type.

When these two conditions are met, a *widening conversion* takes place. For example, the **int** type is always large enough to hold all valid **byte**

values, so no explicit cast statement is required.

For widening conversions, the numeric types, including integer and floating-point types, are compatible with each other. However, there are no automatic conversions from the numeric types to **char** or **boolean**. Also, **char** and **boolean** are not compatible with each other.

As mentioned earlier, Java also performs an automatic type conversion when storing a literal integer constant into variables of type **byte**, **short**, **long**, or **char**.

Casting Incompatible Types

Although the automatic type conversions are helpful, they will not fulfill all needs. For example, what if you want to assign an **int** value to a **byte** variable? This conversion will not be performed automatically, because a **byte** is smaller than an **int**. This kind of conversion is sometimes called a *narrowing conversion*, since you are explicitly making the value narrower so that it will fit into the target type.

To create a conversion between two incompatible types, you must use a cast. A *cast* is simply an explicit type conversion. It has this general form:

(*target-type*) *value*

Here, *target-type* specifies the desired type to convert the specified value to. For example, the following fragment casts an **int** to a **byte**. If the integer's value is larger than the range of a **byte**, it will be reduced modulo (the remainder of an integer division by) the **byte**'s range.

```
int a;
byte b;
// …
b = (byte) a;
```

A different type of conversion will occur when a floating-point value is assigned to an integer type: *truncation*. As you know, integers do not have fractional components. Thus, when a floating-point value is assigned to an integer type, the fractional component is lost. For example, if the value 1.23 is assigned to an integer, the resulting value will simply be 1. The 0.23 will have been truncated. Of course, if the size of the whole number component

is too large to fit into the target integer type, then that value will be reduced modulo the target type's range.

The following program demonstrates some type conversions that require casts:

```
// Demonstrate casts.
class Conversion {
  public static void main(String[] args) {
    byte b;
    int i = 257;
    double d = 323.142;

    System.out.println("\nConversion of int to byte.");
    b = (byte) i;
    System.out.println("i and b " + i + " " + b);

    System.out.println("\nConversion of double to int.");
    i = (int) d;
    System.out.println("d and i " + d + " " + i);

    System.out.println("\nConversion of double to byte.");
    b = (byte) d;
    System.out.println("d and b " + d + " " + b);
  }
}
```

This program generates the following output:

```
Conversion of int to byte.
i and b 257 1

Conversion of double to int.
d and i 323.142 323

Conversion of double to byte.
d and b 323.142 67
```

Let's look at each conversion. When the value 257 is cast into a **byte** variable, the result is the remainder of the division of 257 by 256 (the range of a **byte**), which is 1 in this case. When the **d** is converted to an **int**, its

fractional component is lost. When **d** is converted to a **byte**, its fractional component is lost, *and* the value is reduced modulo 256, which in this case is 67.

Automatic Type Promotion in Expressions

In addition to assignments, there is another place where certain type conversions may occur: in expressions. To see why, consider the following. In an expression, the precision required of an intermediate value will sometimes exceed the range of either operand. For example, examine the following expression:

```
byte a = 40;
byte b = 50;
byte c = 100;
int d = a * b / c;
```

The result of the intermediate term **a * b** easily exceeds the range of either of its **byte** operands. To handle this kind of problem, Java automatically promotes each **byte**, **short**, or **char** operand to **int** when evaluating an expression. This means that the subexpression **a * b** is performed using integers—not bytes. Thus, 2,000, the result of the intermediate expression, **50 * 40**, is legal even though **a** and **b** are both specified as type **byte**.

As useful as the automatic promotions are, they can cause confusing compile-time errors. For example, this seemingly correct code causes a problem:

```
byte b = 50;
b = b * 2; // Error! Cannot assign an int to a byte!
```

The code is attempting to store 50 * 2, a perfectly valid **byte** value, back into a **byte** variable. However, because the operands were automatically promoted to **int** when the expression was evaluated, the result has also been promoted to **int**. Thus, the result of the expression is now of type **int**, which cannot be assigned to a **byte** without the use of a cast. This is true even if, as in this particular case, the value being assigned would still fit in the target type.

In cases where you understand the consequences of overflow, you should use an explicit cast, such as

```
byte b = 50;
b = (byte)(b * 2);
```

which yields the correct value of 100.

The Type Promotion Rules

Java defines several *type promotion* rules that apply to expressions. They are as follows: First, all **byte**, **short**, and **char** values are promoted to **int**, as just described. Then, if one operand is a **long**, the whole expression is promoted to **long**. If one operand is a **float**, the entire expression is promoted to **float**. If any of the operands are **double**, the result is **double**.

The following program demonstrates how each value in the expression gets promoted to match the second argument to each binary operator:

```
class Promote {
  public static void main(String[] args) {
    byte b = 42;
    char c = 'a';
    short s = 1024;
    int i = 50000;
    float f = 5.67f;
    double d = .1234;
    double result = (f * b) + (i / c) - (d * s);
    System.out.println((f * b) + " + " + (i / c) + " - " + (d * s));
    System.out.println("result = " + result);
  }
}
```

Let's look closely at the type promotions that occur in this line from the program:

```
double result = (f * b) + (i / c) - (d * s);
```

In the first subexpression, **f * b**, **b** is promoted to a **float** and the result of the subexpression is **float**. Next, in the subexpression **i/c**, **c** is promoted to

int, and the result is of type **int**. Then, in **d * s**, the value of **s** is promoted to **double**, and the type of the subexpression is **double**. Finally, these three intermediate values, **float**, **int**, and **double**, are considered. The outcome of **float** plus an **int** is a **float**. Then the resultant **float** minus the last **double** is promoted to **double**, which is the type for the final result of the expression.

Arrays

An *array* is a group of like-typed variables that are referred to by a common name. Arrays of any type can be created and may have one or more dimensions. A specific element in an array is accessed by its index. Arrays offer a convenient means of grouping related information.

One-Dimensional Arrays

A *one-dimensional array* is, essentially, a list of like-typed variables. To create an array, you first must create an array variable of the desired type. The general form of a one-dimensional array declaration is

type[ ] *var-name*;

Here, *type* declares the element type (also called the base type) of the array. The element type determines the data type of each element that comprises the array. Thus, the element type for the array determines what type of data the array will hold. For example, the following declares an array named **month_days** with the type "array of int":

```
int[] month_days;
```

Although this declaration establishes the fact that **month_days** is an array variable, no array actually exists. To link **month_days** with an actual, physical array of integers, you must allocate one using **new** and assign it to **month_days**. **new** is a special operator that allocates memory.

You will look more closely at **new** in a later chapter, but you need to use it now to allocate memory for arrays. The general form of **new** as it applies to one-dimensional arrays appears as follows:

array-var = new *type* [*size*];

Here, *type* specifies the type of data being allocated, *size* specifies the number of elements in the array, and *array-var* is the array variable that is linked to the array. That is, to use **new** to allocate an array, you must specify the type and number of elements to allocate. The elements in the array allocated by **new** will automatically be initialized to zero (for numeric types), **false** (for **boolean**), or **null** (for reference types, which are described in a later chapter). This example allocates a 12-element array of integers and links them to **month_days**:

```
month_days = new int[12];
```

After this statement executes, **month_days** will refer to an array of 12 integers. Further, all elements in the array will be initialized to zero.

Let's review: Obtaining an array is a two-step process. First, you must declare a variable of the desired array type. Second, you must allocate the memory that will hold the array, using **new**, and assign it to the array variable. Thus, in Java all arrays are dynamically allocated. If the concept of dynamic allocation is unfamiliar to you, don't worry. It will be described at length later in this book.

Once you have allocated an array, you can access a specific element in the array by specifying its index within square brackets. All array indexes start at zero. For example, this statement assigns the value 28 to the second element of **month_days**:

```
month_days[1] = 28;
```

The next line displays the value stored at index 3:

```
System.out.println(month_days[3]);
```

Putting together all the pieces, here is a program that creates an array of the number of days in each month:

```
// Demonstrate a one-dimensional array.
class Array {
  public static void main(String[] args) {
    int[] month_days;
    month_days = new int[12];
    month_days[0] = 31;
    month_days[1] = 28;
    month_days[2] = 31;
    month_days[3] = 30;
    month_days[4] = 31;
    month_days[5] = 30;
    month_days[6] = 31;
    month_days[7] = 31;
    month_days[8] = 30;
    month_days[9] = 31;
    month_days[10] = 30;
    month_days[11] = 31;
    System.out.println("April has " + month_days[3] + " days.");
  }
}
```

When you run this program, it prints the number of days in April. As mentioned, Java array indexes start with zero, so the number of days in April is **month_days[3]**, or 30.

It is possible to combine the declaration of the array variable with the allocation of the array itself, as shown here:

```
int[] month_days = new int[12];
```

This is the way that you will normally see it done in professionally written Java programs.

Arrays can be initialized when they are declared. The process is much the same as that used to initialize the simple types. An *array initializer* is a list of comma-separated expressions surrounded by curly braces. The commas separate the values of the array elements. The array will automatically be created large enough to hold the number of elements you specify in the array initializer. There is no need to use **new**. For example, to store the number of days in each month, the following code creates an initialized array of integers:

```
// An improved version of the previous program.
class AutoArray {
  public static void main(String[] args) {

    int[] month_days = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31,
                         30, 31 };
    System.out.println("April has " + month_days[3] + " days.");
  }
}
```

When you run this program, you see the same output as that generated by the previous version.

Java strictly checks to make sure you do not accidentally try to store or reference values outside of the range of the array. The Java run-time system will check that all array indexes are in the correct range. For example, the run-time system will check the value of each index into **month_days** to make sure that it is between 0 and 11, inclusive. If you try to access elements outside the range of the array (negative numbers or numbers greater than the length of the array), you will cause a run-time error.

Here is one more example that uses a one-dimensional array. It finds the average of a set of numbers.

```
// Average an array of values.
class Average {
  public static void main(String[] args) {
    double[] nums = {10.1, 11.2, 12.3, 13.4, 14.5};
    double result = 0;
    int i;

    for(i=0; i<5; i++)
      result = result + nums[i];
    System.out.println("Average is " + result / 5);
  }
}
```

Multidimensional Arrays

In Java, *multidimensional arrays* are implemented as arrays of arrays. To declare a multidimensional array variable, specify each additional index

using another set of square brackets. For example, the following declares a two-dimensional array variable called **twoD**:

```
int[][] twoD = new int[4][5];
```

This allocates a 4-by-5 array and assigns it to **twoD**. Internally, this matrix is implemented as an *array* of *arrays* of **int**. Conceptually, this array will look like the one shown in Figure 3-1.

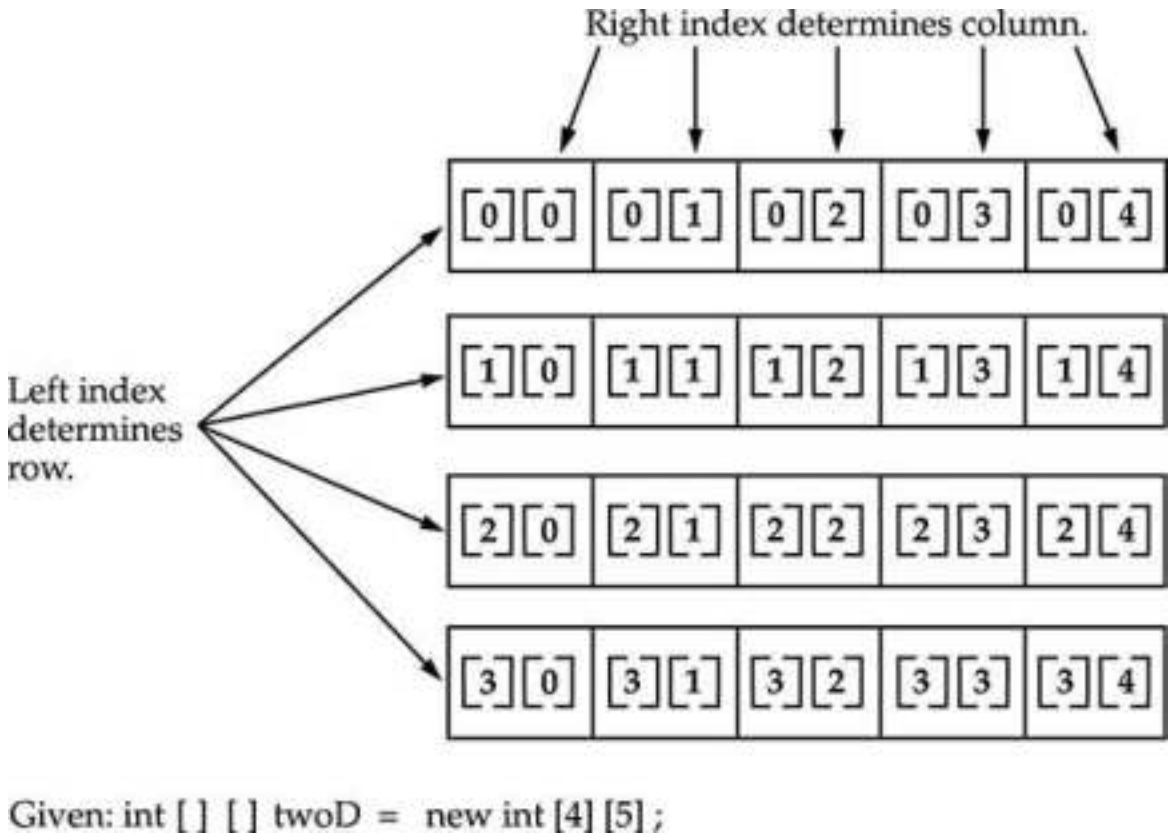


Figure 3-1 A conceptual view of a 4-by-5, two-dimensional array

The following program numbers each element in the array from left to right, top to bottom, and then displays these values:

```
// Demonstrate a two-dimensional array.
class TwoDArray {
  public static void main(String[] args) {
    int[][] twoD= new int[4][5];
    int i, j, k = 0;

    for(i=0; i<4; i++)
      for(j=0; j<5; j++) {
        twoD[i][j] = k;
        k++;
      }

    for(i=0; i<4; i++) {
      for(j=0; j<5; j++)
        System.out.print(twoD[i][j] + " ");
      System.out.println();
    }
  }
}
```

This program generates the following output:

```
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
```

When you allocate memory for a multidimensional array, you need only specify the memory for the first (leftmost) dimension. You can allocate the remaining dimensions separately. For example, this following code allocates memory for the first dimension of **twoD** when it is declared. It allocates the second dimension separately.

```
int[][] twoD = new int[4][];
twoD[0] = new int[5];
twoD[1] = new int[5];
twoD[2] = new int[5];
twoD[3] = new int[5];
```

While there is no advantage to individually allocating the second dimension arrays in this situation, there may be in others. For example,

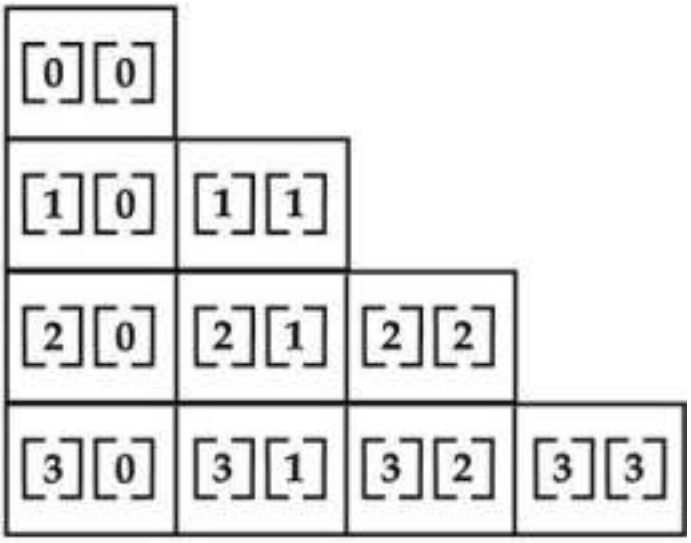
when you allocate dimensions individually, you do not need to allocate the same number of elements for each dimension. As stated earlier, since multidimensional arrays are actually arrays of arrays, the length of each array is under your control. For example, the following program creates a two-dimensional array in which the sizes of the second dimension are unequal:

```
// Manually allocate differing size second dimensions.
class TwoDAgain {
  public static void main(String[] args) {
    int[][] twoD = new int[4][];
    twoD[0] = new int[1];
    twoD[1] = new int[2];
    twoD[2] = new int[3];
    twoD[3] = new int[4];

    int i, j, k = 0;

    for(i=0; i<4; i++)
      for(j=0; j<i+1; j++) {
      twoD[i][j] = k;
      k++;
    }

    for(i=0; i<4; i++) {
      for(j=0; j<i+1; j++)
        System.out.print(twoD[i][j] + " ");
      System.out.println();
    }
  }
}
```

This program generates the following output:

```
0
1 2
3 4 5
6 7 8 9
```

The array created by this program looks like this:



The use of uneven (or irregular) multidimensional arrays may not be appropriate for many applications, because it runs contrary to what people expect to find when a multidimensional array is encountered. However, irregular arrays can be used effectively in some situations. For example, if you need a very large two-dimensional array that is sparsely populated (that is, one in which not all of the elements will be used), then an irregular array might be a perfect solution.

It is possible to initialize multidimensional arrays. To do so, simply enclose each dimension's initializer within its own set of curly braces. The following program creates a matrix where each element contains the product of the row and column indexes. Also notice that you can use expressions as well as literal values inside of array initializers.

```
// Initialize a two-dimensional array.
class Matrix {
  public static void main(String[] args) {
```

```
    double[][] m = {
      { 0*0, 1*0, 2*0, 3*0 },
      { 0*1, 1*1, 2*1, 3*1 },
      { 0*2, 1*2, 2*2, 3*2 },
      { 0*3, 1*3, 2*3, 3*3 }
    };
    int i, j;

    for(i=0; i<4; i++) {
      for(j=0; j<4; j++)
        System.out.print(m[i][j] + " ");
      System.out.println();
    }
  }
}
```

When you run this program, you will get the following output:

```
0.0  0.0  0.0  0.0
0.0  1.0  2.0  3.0
0.0  2.0  4.0  6.0
0.0  3.0  6.0  9.0
```

As you can see, each row in the array is initialized as specified in the initialization lists.

Let's look at one more example that uses a multidimensional array. The following program creates a 3-by-4-by-5, three-dimensional array. It then loads each element with the product of its indexes. Finally, it displays these products.

```
// Demonstrate a three-dimensional array.
class ThreeDMatrix {
  public static void main(String[] args) {
    int[][][] threeD = new int[3][4][5];
    int i, j, k;

    for(i=0; i<3; i++)
      for(j=0; j<4; j++)
        for(k=0; k<5; k++)
          threeD[i][j][k] = i * j * k;

    for(i=0; i<3; i++) {
      for(j=0; j<4; j++) {
        for(k=0; k<5; k++)
          System.out.print(threeD[i][j][k] + " ");
        System.out.println();
      }
      System.out.println();
    }
  }
}
```

This program generates the following output:

```
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0

0 0 0 0 0
0 1 2 3 4
0 2 4 6 8
0 3 6 9 12

0 0 0 0 0
0 2 4 6 8
0 4 8 12 16
0 6 12 18 24
```

Alternative Array Declaration Syntax

There is a second form that may be used to declare an array:

type var-name[ ];

Here, the square brackets follow the array variable name, and not the type specifier. For example, the following two declarations are equivalent:

```
int a1[] = new int[3];
int[] a2 = new int[3];
```

The following declarations are also equivalent:

```
char twod1[][] = new char[3][4];
char[][] twod2 = new char[3][4];
```

This alternative declaration form offers convenience when converting code from C/C++ to Java. It also lets you declare both array and non-array variables in a single declaration statement. Today, the alternative form of array declaration is less commonly used, but it is still important that you are familiar with it because both forms of array declaration are legal in Java.

Introducing Type Inference with Local Variables

Not long ago, a new feature called *local variable type inference* was added to the Java language. To begin, let's review two important aspects of variables. First, all variables in Java must be declared prior to their use. Second, a variable can be initialized with a value when it is declared. Furthermore, when a variable is initialized, the type of the initializer must be the same as (or convertible to) the declared type of the variable. Thus, in principle, it would not be necessary to specify an explicit type for an initialized variable because it could be inferred by the type of its initializer. Of course, in the past, such inference was not supported, and all variables required an explicitly declared type, whether they were initialized or not. Today, that situation has changed.

Beginning with JDK 10, it is now possible to let the compiler infer the type of a local variable based on the type of its initializer, thus avoiding the need to explicitly specify the type. Local variable type inference offers a number of advantages. For example, it can streamline code by eliminating

the need to redundantly specify a variable's type when it can be inferred from its initializer. It can simplify declarations in cases in which the type name is quite lengthy, such as can be the case with some class names. It can also be helpful when a type is difficult to discern or cannot be denoted. (An example of a type that cannot be denoted is the type of an anonymous class, discussed in Chapter 25.) Furthermore, local variable type inference has become a common part of the contemporary programming environment. Its inclusion in Java helps keep Java up to date with evolving trends in language design. To support local variable type inference, the context-sensitive keyword **var** was added.

To use local variable type inference, the variable must be declared with **var** as the type name and it must include an initializer. For example, in the past you would declare a local **double** variable called **avg** that is initialized with the value 10.0 as shown here:

```
double avg = 10.0;
```

Using type inference, this declaration can now also be written like this:

```
var avg = 10.0;
```

In both cases, **avg** will be of type **double**. In the first case, its type is explicitly specified. In the second, its type is inferred as **double** because the initializer 10.0 is of type **double**.

As mentioned, **var** is context-sensitive. When it is used as the type name in the context of a local variable declaration, it tells the compiler to use type inference to determine the type of the variable being declared based on the type of the initializer. Thus, in a local variable declaration, **var** is a placeholder for the actual, inferred type. However, when used in most other places, **var** is simply a user-defined identifier with no special meaning. For example, the following declaration is still valid:

```
int var = 1; // In this case, var is simply a user-defined
identifier.
```

In this case, the type is explicitly specified as **int** and **var** is the name of the variable being declared. Even though it is context-sensitive, there are a few

places in which the use of **var** is illegal. It cannot be used as the name of a class, for example.

The following program puts the preceding discussion into action:

```
// A simple demonstration of local variable type inference.
class VarDemo {
  public static void main(String[] args) {

    // Use type inference to determine the type of the
    // variable named avg. In this case, double is inferred.
    var avg = 10.0;
    System.out.println("Value of avg: " + avg);

    // In the following context, var is not a predefined identifier.
    // It is simply a user-defined variable name.
    int var = 1;
    System.out.println("Value of var: " + var);

    // Interestingly, in the following sequence, var is used
    // as both the type of the declaration and as a variable name
    // in the initializer.
    var k = -var;
    System.out.println("Value of k: " + k);
  }
}
```

Here is the output:

```
Value of avg: 10.0
Value of var: 1
Value of k: -1
```

The preceding example uses **var** to declare only simple variables, but you can also use **var** to declare an array. For example:

```
var myArray = new int[10]; // This is valid.
```

Notice that neither **var** nor **myArray** has brackets. Instead, the type of **myArray** is inferred to be **int[ ]**. Furthermore, you *cannot* use brackets on

the left side of a **var** declaration. Thus, both of these declarations are invalid:

```
var[] myArray = new int[10]; // Wrong
var myArray[] = new int[10]; // Wrong
```

In the first line, an attempt is made to bracket **var**. In the second, an attempt is made to bracket **myArray**. In both cases, the use of the brackets is wrong because the type is inferred from the type of the initializer.

It is important to emphasize that **var** can be used to declare a variable only when that variable is initialized. For example, the following statement is incorrect:

```
var counter; // Wrong! Initializer required.
```

Also, remember that **var** can be used only to declare local variables. It cannot be used when declaring instance variables, parameters, or return types, for example.

Although the preceding discussion and examples have introduced the basics of local variable type inference, they haven't shown its full power. As you will see in Chapter 7, local variable type inference is especially effective in shortening declarations that involve long class names. It can also be used with generic types (see Chapter 14), in a **try**-with-resources statement (see Chapter 13), and with a **for** loop (see Chapter 5).

Some var Restrictions

In addition to those mentioned in the preceding discussion, several other restrictions apply to the use of **var**. Only one variable can be declared at a time; a variable cannot use **null** as an initializer; and the variable being declared cannot be used by the initializer expression. Although you can declare an array type using **var**, you cannot use **var** with an array initializer. For example, this is valid:

```
var myArray = new int[10]; // This is valid.
```

However, this is not:

```
var myArray = { 1, 2, 3 }; // Wrong
```

As mentioned earlier, **var** cannot be used as the name of a class. It also cannot be used as the name of other reference types, including an interface, enumeration, or annotation, or as the name of a generic type parameter, all of which are described later in this book. Here are two other restrictions that relate to Java features described in subsequent chapters but mentioned here in the interest of completeness. Local variable type inference cannot be used to declare the exception type caught by a **catch** statement. Also, neither lambda expressions nor method references can be used as initializers.

NOTE At the time of this writing, a number of readers of this book will be using Java environments that don't support local variable type inference. So that as many of the code examples as possible will compile and run for all readers, local variable type inference will not be used by most of the programs in the remainder of this edition of the book. Using the full declaration syntax also makes it very clear at a glance what type of variable is being created, which is important for the sample code. Of course, going forward, you should consider the use of local variable type inference where appropriate in your own code.

A Few Words About Strings

As you may have noticed, in the preceding discussion of data types and arrays there has been no mention of strings or a string data type. This is not because Java does not support such a type—it does. It is just that Java's string type, called **String**, is not a primitive type. Nor is it simply an array of characters. Rather, **String** defines an object, and a full description of it requires an understanding of several object-related features. As such, it will be covered later in this book, after objects are described. However, so that you can use simple strings in sample programs, the following brief introduction is in order.

The **String** type is used to declare string variables. You can also declare arrays of strings. A quoted string constant can be assigned to a **String** variable. A variable of type **String** can be assigned to another variable of type **String**. You can use an object of type **String** as an argument to **println( )**. For example, consider the following fragment:

```
String str = "this is a test";
System.out.println(str);
```

Here, **str** is an object of type **String**. It is assigned the string "this is a test". This string is displayed by the **println( )** statement.

As you will see later, **String** objects have many special features and attributes that make them quite powerful and easy to use. However, for the next few chapters, you will be using them only in their simplest form.

CHAPTER

4

Operators

Java provides a rich operator environment. Most of its operators can be divided into the following four groups: arithmetic, bitwise, relational, and logical. Java also defines some additional operators that handle certain special situations. This chapter describes all of Java's operators except for the type comparison operator **instanceof**, which is examined in Chapter 13, and the arrow operator (−>), which is described in Chapter 15.

Arithmetic Operators

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:

Operator	Result
+	Addition (also unary plus)
-	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
-=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
--	Decrement

The operands of the arithmetic operators must be of a numeric type. You cannot use them on **boolean** types, but you can use them on **char** types, since the **char** type in Java is, essentially, a subset of **int**.

The Basic Arithmetic Operators

The basic arithmetic operations—addition, subtraction, multiplication, and division—all behave as you would expect for all numeric types. The unary minus operator negates its single operand. The unary plus operator simply returns the value of its operand. Remember that when the division operator is applied to an integer type, there will be no fractional component attached to the result.

The following simple sample program demonstrates the arithmetic operators. It also illustrates the difference between floating-point division and integer division.

```
// Demonstrate the basic arithmetic operators.
class BasicMath {
  public static void main(String[] args) {
    // arithmetic using integers
    System.out.println("Integer Arithmetic");
    int a = 1 + 1;
    int b = a * 3;
    int c = b / 4;
    int d = c - a;
    int e = -d;
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    System.out.println("c = " + c);
    System.out.println("d = " + d);
    System.out.println("e = " + e);

    // arithmetic using doubles
    System.out.println("\nFloating Point Arithmetic");
    double da = 1 + 1;
    double db = da * 3;
    double dc = db / 4;
    double dd = dc - a;
    double de = -dd;
    System.out.println("da = " + da);
    System.out.println("db = " + db);
    System.out.println("dc = " + dc);
    System.out.println("dd = " + dd);
    System.out.println("de = " + de);
  }
}
```

When you run this program, you will see the following output:

```
Integer Arithmetic
a = 2
b = 6
c = 1
d = -1
e = 1

Floating Point Arithmetic
da = 2.0
db = 6.0
dc = 1.5
dd = -0.5
de = 0.5
```

The Modulus Operator

The modulus operator, **%**, returns the remainder of a division operation. It can be applied to floating-point types as well as integer types. The following sample program demonstrates the **%**:

```
// Demonstrate the % operator.
class Modulus {
  public static void main(String[] args) {
    int x = 42;
    double y = 42.25;

    System.out.println("x mod 10 = " + x % 10);
    System.out.println("y mod 10 = " + y % 10);
  }
}
```

When you run this program, you will get the following output:

```
x mod 10 = 2
y mod 10 = 2.25
```

Arithmetic Compound Assignment Operators

Java provides special operators that can be used to combine an arithmetic operation with an assignment. As you probably know, statements like the

following are quite common in programming:

```
a = a + 4;
```

In Java, you can rewrite this statement as shown here:

```
a += 4;
```

This version uses the += *compound assignment operator*. Both statements perform the same action: they increase the value of **a** by 4.

Here is another example,

```
a = a % 2;
```

which can be expressed as

```
a %= 2;
```

In this case, the **%=** obtains the remainder of **a** /2 and puts that result back into **a**.

There are compound assignment operators for all of the arithmetic, binary operators. Thus, any statement of the form

var = var op expression;

can be rewritten as

var op= expression;

The compound assignment operators provide two benefits. First, they save you a bit of typing, because they are "shorthand" for their equivalent long forms. Second, in some cases they are more efficient than are their equivalent long forms. For these reasons, you will often see the compound assignment operators used in professionally written Java programs.

Here is a sample program that shows several *op*= assignments in action:

```
// Demonstrate several assignment operators.
class OpEquals {
  public static void main(String[] args) {
    int a = 1;
    int b = 2;
    int c = 3;

    a += 5;
    b *= 4;
    c += a * b;
    c %= 6;
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    System.out.println("c = " + c);
  }
}
```

The output of this program is shown here:

```
a = 6
b = 8
c = 3
```

Increment and Decrement

The ++ and the – – are Java's increment and decrement operators, respectively. They were introduced in Chapter 2. Here they will be discussed in detail. As you will see, they have some special properties that make them quite interesting. Let's begin by reviewing precisely what the increment and decrement operators do.

The increment operator increases its operand by one. The decrement operator decreases its operand by one. For example, this statement:

```
x = x + 1;
```

can be rewritten like this by use of the increment operator:

```
x++;
```

Similarly, this statement:

```
x = x - 1;
```

is equivalent to

```
x--;
```

These operators are unique in that they can appear both in *postfix* form, where they follow the operand as just shown, and *prefix* form, where they precede the operand. In the foregoing examples, there is no difference between the prefix and postfix forms. However, when the increment and/or decrement operators are part of a larger expression, then a subtle, yet powerful, difference between these two forms appears. In the prefix form, the operand is incremented or decremented before the value is obtained for use in the expression. In postfix form, the previous value is obtained for use in the expression, and then the operand is modified. For example:

```
x = 42;
y = ++x;
```

In this case, **y** is set to 43 as you would expect, because the increment occurs *before* **x** is assigned to **y**. Thus, the line **y = ++x**; is the equivalent of these two statements:

```
x = x + 1;
y = x;
```

However, when written like this,

```
x = 42;
y = x++;
```

the value of **x** is obtained before the increment operator is executed, so the value of **y** is 42. Of course, in both cases **x** is set to 43. Here, the line **y = x++**; is the equivalent of these two statements:

```
y = x;
x = x + 1;
```

The following program demonstrates the increment operator:

```
// Demonstrate ++.
class IncDec {
  public static void main(String[] args) {
    int a = 1;
    int b = 2;
    int c;
    int d;
    c = ++b;
    d = a++;
    c++;
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    System.out.println("c = " + c);
    System.out.println("d = " + d);
  }
}
```

The output of this program follows:

```
a = 2
b = 3
c = 4
d = 1
```

The Bitwise Operators

Java defines several *bitwise operators* that can be applied to the integer types: **long**, **int**, **short**, **char**, and **byte**. These operators act upon the individual bits of their operands. They are summarized in the following table:

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

Since the bitwise operators manipulate the bits within an integer, it is important to understand what effects such manipulations may have on a value. Specifically, it is useful to know how Java stores integer values and how it represents negative numbers. So, before continuing, let's briefly review these two topics.

All of the integer types are represented by binary numbers of varying bit widths. For example, the **byte** value for 42 in binary is 00101010, where each position represents a power of two, starting with 2^0 at the rightmost bit. The next bit position to the left would be 2^1, or 2, continuing toward the left with 2^2, or 4, then 8, 16, 32, and so on. So 42 has 1 bits set at positions 1, 3, and 5 (counting from 0 at the right); thus, 42 is the sum of $2^1 + 2^3 + 2^5$, which is 2 + 8 + 32.

All of the integer types (except **char**) are signed integers. This means that they can represent negative values as well as positive ones. Java uses an encoding known as *two's complement*, which means that negative numbers are represented by inverting (changing 1's to 0's and vice versa) all of the bits in a value, then adding 1 to the result. For example, –42 is

represented by inverting all of the bits in 42, or 00101010, which yields 11010101, then adding 1, which results in 11010110, or –42. To decode a negative number, first invert all of the bits, then add 1. For example, –42, or 11010110 inverted, yields 00101001, or 41, so when you add 1 you get 42.

The reason Java (and most other computer languages) uses two's complement is easy to see when you consider the issue of *zero crossing*. Assuming a **byte** value, zero is represented by 00000000. In one's complement, simply inverting all of the bits creates 11111111, which creates negative zero. The trouble is that negative zero is invalid in integer math. This problem is solved by using two's complement to represent negative values. When using two's complement, 1 is added to the complement, producing 100000000. This produces a 1 bit too far to the left to fit back into the **byte** value, resulting in the desired behavior, where –0 is the same as 0, and 11111111 is the encoding for –1. Although we used a **byte** value in the preceding example, the same basic principle applies to all of Java's integer types.

Because Java uses two's complement to store negative numbers—and because all integers are signed values in Java—applying the bitwise operators can easily produce unexpected results. For example, turning on the high-order bit will cause the resulting value to be interpreted as a negative number, whether this is what you intended or not. To avoid unpleasant surprises, just remember that the high-order bit determines the sign of an integer no matter how that high-order bit gets set.

The Bitwise Logical Operators

The bitwise logical operators are **&**, |, ^, and ~. The following table shows the outcome of each operation. In the discussion that follows, keep in mind that the bitwise operators are applied to each individual bit within each operand.

A	B	A|B	A&B	A^B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

The Bitwise NOT

Also called the *bitwise complement*, the unary NOT operator, ~, inverts all of the bits of its operand. For example, the number 42, which has the following bit pattern:

00101010

becomes

11010101

after the NOT operator is applied.

The Bitwise AND

The AND operator, **&**, produces a 1 bit if both operands are also 1. A zero is produced in all other cases. Here is an example:

```
 00101010  42
&00001111  15
----------
 00001010  10
```

The Bitwise OR

The OR operator, |, combines bits such that if either of the bits in the operands is a 1, then the resultant bit is a 1, as shown here:

```
 00101010   42
|00001111   15
----------
 00101111   47
```

The Bitwise XOR

The XOR operator, ^, combines bits such that if exactly one operand is 1, then the result is 1. Otherwise, the result is zero. The following example shows the effect of the ^. This example also demonstrates a useful attribute of the XOR operation. Notice how the bit pattern of 42 is inverted wherever the second operand has a 1 bit. Wherever the second operand has a 0 bit, the first operand is unchanged. You will find this property useful when performing some types of bit manipulations.

```
  00101010   42
^ 00001111   15
----------
  00100101   37
```

Using the Bitwise Logical Operators

The following program demonstrates the bitwise logical operators:

```
// Demonstrate the bitwise logical operators.
class BitLogic {
  public static void main(String[] args) {
    String[] binary = {
      "0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111",
      "1000", "1001", "1010", "1011", "1100", "1101", "1110", "1111"
    };
    int a = 3; // 0 + 2 + 1 or 0011 in binary
    int b = 6; // 4 + 2 + 0 or 0110 in binary
    int c = a | b;
    int d = a & b;
    int e = a ^ b;
    int f = (~a & b)|(a & ~b);
    int g = ~a & 0x0f;
```

```
    System.out.println("        a = " + binary[a]);
    System.out.println("        b = " + binary[b]);
    System.out.println("      a|b = " + binary[c]);
    System.out.println("      a&b = " + binary[d]);
    System.out.println("      a^b = " + binary[e]);
    System.out.println("~a&b|a&~b = " + binary[f]);
    System.out.println("       ~a = " + binary[g]);
  }
}
```

In this example, **a** and **b** have bit patterns that present all four possibilities for two binary digits: 0-0, 0-1, 1-0, and 1-1. You can see how the | and **&** operate on each bit by the results in **c** and **d**. The values assigned to **e** and **f** are the same and illustrate how the ^ works. The string array named **binary** holds the human-readable, binary representation of the numbers 0 through 15. In this example, the array is indexed to show the binary representation of each result. The array is constructed such that the correct string representation of a binary value **n** is stored in **binary[n]**. The value of **~a** is ANDed with **0x0f** (0000 1111 in binary) in order to reduce its value to less than 16, so it can be printed by use of the **binary** array. Here is the output from this program:

```
        a = 0011
        b = 0110
      a|b = 0111
      a&b = 0010
      a^b = 0101
~a&b|a&~b = 0101
       ~a = 1100
```

The Left Shift

The left shift operator, **<<,** shifts all of the bits in a value to the left a specified number of times. It has this general form:

value << *num*

Here, *num* specifies the number of positions to left-shift the value in *value*. That is, the **<<** moves all of the bits in the specified value to the left by the

number of bit positions specified by *num*. For each shift left, the high-order bit is shifted out (and lost), and a zero is brought in on the right. This means that when a left shift is applied to an **int** operand, bits are lost once they are shifted past bit position 31. If the operand is a **long**, then bits are lost after bit position 63.

Java's automatic type promotions produce unexpected results when you are shifting **byte** and **short** values. As you know, **byte** and **short** values are promoted to **int** when an expression is evaluated. Furthermore, the result of such an expression is also an **int**. This means that the outcome of a left shift on a **byte** or **short** value will be an **int**, and the bits shifted left will not be lost until they shift past bit position 31. Furthermore, a negative **byte** or **short** value will be sign-extended when it is promoted to **int**. Thus, the high-order bits will be filled with 1's. For these reasons, to perform a left shift on a **byte** or **short** implies that you must discard the high-order bytes of the **int** result. For example, if you left-shift a **byte** value, that value will first be promoted to **int** and then shifted. This means that you must discard the top three bytes of the result if what you want is the result of a shifted **byte** value. The easiest way to do this is to simply cast the result back into a **byte**. The following program demonstrates this concept:

```
// Left shifting a byte value.
class ByteShift {
  public static void main(String[] args) {
    byte a = 64, b;
    int i;

    i = a << 2;
    b = (byte) (a << 2);

    System.out.println("Original value of a: " + a);
    System.out.println("i and b: " + i + " " + b);
  }
}
```

The output generated by this program is shown here:

```
Original value of a: 64
i and b: 256 0
```

Since **a** is promoted to **int** for the purposes of evaluation, left-shifting the value 64 (0100 0000) twice results in **i** containing the value 256 (1 0000 0000). However, the value in **b** contains 0 because after the shift, the low-order byte is now zero. Its only 1 bit has been shifted out.

Since each left shift has the effect of doubling the original value, programmers frequently use this fact as an efficient alternative to multiplying by 2. But you need to watch out. If you shift a 1 bit into the high-order position (bit 31 or 63), the value will become negative. The following program illustrates this point:

```
// Left shifting as a quick way to multiply by 2.
class MultByTwo {
  public static void main(String[] args) {
    int i;
    int num = 0xFFFFFFE;

    for(i=0; i<4; i++) {
      num = num << 1;
      System.out.println(num);
    }
  }
}
```

The program generates the following output:

```
536870908
1073741816
2147483632
-32
```

The starting value was carefully chosen so that after being shifted left 4 bit positions, it would produce –32. As you can see, when a 1 bit is shifted into bit 31, the number is interpreted as negative.

The Right Shift

The right shift operator, >>, shifts all of the bits in a value to the right a specified number of times. Its general form is shown here:

value >> *num*

Here, *num* specifies the number of positions to right-shift the value in *value*. That is, the >> moves all of the bits in the specified value to the right the number of bit positions specified by *num*.

The following code fragment shifts the value 32 to the right by two positions, resulting in **a** being set to **8**:

```
int a = 32;
a = a >> 2; // a now contains 8
```

When a value has bits that are "shifted off," those bits are lost. For example, the next code fragment shifts the value 35 to the right two positions, which causes the two low-order bits to be lost, resulting again in **a** being set to 8:

```
int a = 35;
a = a >> 2; // a contains 8
```

Looking at the same operation in binary shows more clearly how this happens:

```
00100011  35
>> 2
00001000   8
```

Each time you shift a value to the right, it divides that value by two—and discards any remainder. In some cases, you can take advantage of this for high-performance integer division by 2.

When you are shifting right, the top (leftmost) bits exposed by the right shift are filled in with the previous contents of the top bit. This is called *sign extension* and serves to preserve the sign of negative numbers when you shift them right. For example, –8 >> 1 is –4, which, in binary, is

```
11111000  –8
>> 1
11111100  –4
```

It is interesting to note that if you shift –1 right, the result always remains –1, since sign extension keeps bringing in more ones in the high-order bits.

Sometimes it is not desirable to sign-extend values when you are shifting them to the right. For example, the following program converts a **byte** value to its hexadecimal string representation. Notice that the shifted value is masked by ANDing it with **0x0f** to discard any sign-extended bits so that the value can be used as an index into the array of hexadecimal characters.

```
// Masking sign extension.
class HexByte {
  static public void main(String[] args) {
    char[] hex = {
      '0', '1', '2', '3', '4', '5', '6', '7',
      '8', '9', 'a', 'b', 'c', 'd', 'e', 'f'
    };

    byte b = (byte) 0xf1;

   System.out.println("b = 0x" + hex[(b >> 4) & 0x0f] + hex[b & 0x0f]);
  }
}
```

Here is the output of this program:

```
b = 0xf1
```

The Unsigned Right Shift

As you have just seen, the >> operator automatically fills the high-order bit with its previous contents each time a shift occurs. This preserves the sign of the value. However, sometimes this is undesirable. For example, if you are shifting something that does not represent a numeric value, you may not want sign extension to take place. This situation is common when you are working with pixel-based values and graphics. In these cases, you will generally want to shift a zero into the high-order bit no matter what its initial value was. This is known as an *unsigned shift*. To accomplish this,

you will use Java's unsigned, shift-right operator, **>>>,** which always shifts zeros into the high-order bit.

The following code fragment demonstrates the **>>>.** Here, **a** is set to –1, which sets all 32 bits to 1 in binary. This value is then shifted right 24 bits, filling the top 24 bits with zeros, ignoring normal sign extension. This sets **a** to 255.

```
int a = -1;
a = a >>> 24;
```

Here is the same operation in binary form to further illustrate what is happening:

```
11111111 11111111 11111111 11111111   –1 in binary as an int
>>>24
00000000 00000000 00000000 11111111   255 in binary as an int
```

The >>> operator is often not as useful as you might like, since it is only meaningful for 32- and 64-bit values. Remember, smaller values are automatically promoted to **int** in expressions. This means that sign extension occurs and that the shift will take place on a 32-bit rather than on an 8- or 16-bit value. That is, one might expect an unsigned right shift on a **byte** value to zero-fill beginning at bit 7. But this is not the case, since it is a 32-bit value that is actually being shifted. The following program demonstrates this effect:

```
// Unsigned shifting a byte value.
class ByteUShift {
  static public void main(String[] args) {
    char[] hex = {
      '0', '1', '2', '3', '4', '5', '6', '7',
      '8', '9', 'a', 'b', 'c', 'd', 'e', 'f'
    };
    byte b = (byte) 0xf1;
    byte c = (byte) (b >> 4);
    byte d = (byte) (b >>> 4);
    byte e = (byte) ((b & 0xff) >> 4);

    System.out.println("               b = 0x"
      + hex[(b >> 4) & 0x0f] + hex[b & 0x0f]);
    System.out.println("          b >> 4 = 0x"
      + hex[(c >> 4) & 0x0f] + hex[c & 0x0f]);
    System.out.println("         b >>> 4 = 0x"
      + hex[(d >> 4) & 0x0f] + hex[d & 0x0f]);
    System.out.println("(b & 0xff) >> 4 = 0x"
      + hex[(e >> 4) & 0x0f] + hex[e & 0x0f]);
  }
}
```

The following output of this program shows how the >>> operator appears to do nothing when dealing with bytes. The variable **b** is set to an arbitrary negative **byte** value for this demonstration. Then **c** is assigned the **byte** value of **b** shifted right by four, which is 0xff because of the expected sign extension. Then **d** is assigned the **byte** value of **b** unsigned shifted right by four, which you might have expected to be 0x0f but is actually 0xff because of the sign extension that happened when **b** was promoted to **int** before the shift. The last expression sets **e** to the **byte** value of **b** masked to 8 bits using the AND operator, then shifted right by four, which produces the expected value of 0x0f. Notice that the unsigned shift right operator was not used for **d**, since the state of the sign bit after the AND was known.

```
               b = 0xf1
          b >> 4 = 0xff
         b >>> 4 = 0xff
(b & 0xff) >> 4 = 0x0f
```

Bitwise Operator Compound Assignments

All of the binary bitwise operators have a compound form similar to that of the algebraic operators, which combines the assignment with the bitwise operation. For example, the following two statements, which shift the value in **a** right by four bits, are equivalent:

```
a = a >> 4;
a >>= 4;
```

Likewise, the following two statements, which result in **a** being assigned the bitwise expression **a** OR **b**, are equivalent:

```
a = a | b;
a |= b;
```

The following program creates a few integer variables and then uses compound bitwise operator assignments to manipulate the variables:

```
class OpBitEquals {
  public static void main(String[] args) {
    int a = 1;
    int b = 2;
    int c = 3;

    a |= 4;
    b >>= 1;
    c <<= 1;
    a ^= c;
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    System.out.println("c = " + c);
  }
}
```

The output of this program is shown here:

```
a = 3
b = 1
c = 6
```

Relational Operators

The *relational operators* determine the relationship that one operand has to the other. Specifically, they determine equality and ordering. The relational operators are shown here:

Operator	Result
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

The outcome of these operations is a **boolean** value. The relational operators are most frequently used in the expressions that control the **if** statement and the various loop statements.

Any type in Java, including integers, floating-point numbers, characters, and Booleans can be compared using the equality test, ==, and the inequality test, **!=**. Notice that in Java equality is denoted with two equal signs, not one. (Remember: a single equal sign is the assignment operator.) Only numeric types can be compared using the ordering operators. That is, only integer, floating-point, and character operands may be compared to see which is greater or less than the other.

As stated, the result produced by a relational operator is a **boolean** value. For example, the following code fragment is perfectly valid:

```
int a = 4;
int b = 1;
boolean c = a < b;
```

In this case, the result of **a<b** (which is **false**) is stored in **c**.

If you are coming from a C/C++ background, please note the following. In C/C++, these types of statements are very common:

```
int done;
//...
if(!done)... // Valid in C/C++
if(done)...  // but not in Java.
```

In Java, these statements must be written like this:

```
if(done == 0)... // This is Java-style.
if(done != 0)...
```

The reason is that Java does not define true and false in the same way as C/C++. In C/C++, true is any nonzero value and false is zero. In Java, **true** and **false** are nonnumeric values that do not relate to zero or nonzero. Therefore, to test for zero or nonzero, you must explicitly employ one or more of the relational operators.

Boolean Logical Operators

The Boolean logical operators shown here operate only on **boolean** operands. All of the binary logical operators combine two **boolean** values to form a resultant **boolean** value.

Operator	Result
&	Logical AND
|	Logical OR
^	Logical XOR (exclusive OR)
||	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
|=	OR assignment
^=	XOR assignment

Operator	Result
==	Equal to
!=	Not equal to
?:	Ternary if-then-else

The logical Boolean operators, **&**, |, and ^, operate on **boolean** values in the same way that they operate on the bits of an integer. The logical **!** operator inverts the Boolean state: **!true == false** and **!false == true**. The following table shows the effect of each logical operation:

A	B	A | B	A & B	A ^ B	!A
False	False	False	False	False	True
True	False	True	False	True	False
False	True	True	False	True	True
True	True	True	True	False	False

Here is a program that is almost the same as the **BitLogic** example shown earlier, but it operates on **boolean** logical values instead of binary bits:

```
// Demonstrate the boolean logical operators.
class BoolLogic {
  public static void main(String[] args) {
    boolean a = true;
    boolean b = false;
    boolean c = a | b;
    boolean d = a & b;
    boolean e = a ^ b;
    boolean f = (!a & b) | (a & !b);
    boolean g = !a;
    System.out.println("        a = " + a);
    System.out.println("        b = " + b);
    System.out.println("      a|b = " + c);
    System.out.println("      a&b = " + d);
    System.out.println("      a^b = " + e);
    System.out.println("!a&b|a&!b = " + f);
    System.out.println("       !a = " + g);
  }
}
```

After running this program, you will see that the same logical rules apply to **boolean** values as they did to bits. As you can see from the following output, the string representation of a Java **boolean** value is one of the literal values **true** or **false**:

```
        a = true
        b = false
      a|b = true
      a&b = false
      a^b = true
!a&b|a&!b = true
       !a = false
```

Short-Circuit Logical Operators

Java provides two interesting Boolean operators not found in some other computer languages. These are secondary versions of the Boolean AND and OR operators, and are commonly known as *short-circuit* logical operators. As you can see from the preceding table, the OR operator results in **true** when **A** is **true**, no matter what **B** is. Similarly, the AND operator

results in **false** when **A** is **false**, no matter what **B** is. If you use the || and **&&** forms rather than the | and **&** forms of these operators, Java will not bother to evaluate the right-hand operand when the outcome of the expression can be determined by the left operand alone. This is very useful when the right-hand operand depends on the value of the left one in order to function properly. For example, the following code fragment shows how you can take advantage of short-circuit logical evaluation to be sure that a division operation will be valid before evaluating it:

```
if (denom != 0 && num / denom > 10)
```

Since the short-circuit form of AND (**&&**) is used, there is no risk of causing a run-time exception when **denom** is zero. If this line of code were written using the single **&** version of AND, both sides would be evaluated, causing a run-time exception when **denom** is zero.

It is standard practice to use the short-circuit forms of AND and OR in cases involving Boolean logic, leaving the single-character versions exclusively for bitwise operations. However, there are exceptions to this rule. For example, consider the following statement:

```
if(c==1 & e++ < 100) d = 100;
```

Here, using a single **&** ensures that the increment operation will be applied to **e** whether **c** is equal to 1 or not.

NOTE The formal specification for Java refers to the short-circuit operators as the *conditional-and* and the *conditional-or*.

The Assignment Operator

You have been using the assignment operator since Chapter 2. Now it is time to take a formal look at it. The *assignment operator* is the single equal sign, =. The assignment operator works in Java much as it does in any other computer language. It has this general form:

var = *expression*;

Here, the type of *var* must be compatible with the type of *expression*.

The assignment operator does have one interesting attribute that you may not be familiar with: it allows you to create a chain of assignments. For example, consider this fragment:

```
int x, y, z;

x = y = z = 100; // set x, y, and z to 100
```

This fragment sets the variables **x**, **y**, and **z** to 100 using a single statement. This works because the = is an operator that yields the value of the right-hand expression. Thus, the value of **z = 100** is 100, which is then assigned to **y**, which in turn is assigned to **x**. Using a "chain of assignment" is an easy way to set a group of variables to a common value.

The ? Operator

Java includes a special *ternary* (three-way) *operator* that can replace certain types of if-then-else statements. This operator is the **?**. It can seem somewhat confusing at first, but the **?** can be used very effectively once mastered. The **?** has this general form:

expression1 ? *expression2* : *expression3*

Here, *expression1* can be any expression that evaluates to a **boolean** value. If *expression1* is **true**, then *expression2* is evaluated; otherwise, *expression3* is evaluated. The result of the **?** operation is that of the expression evaluated. Both *expression2* and *expression3* are required to return the same (or compatible) type, which can't be **void**.

Here is an example of the way that the **?** is employed:

```
ratio = denom == 0 ? 0 : num / denom;
```

When Java evaluates this assignment expression, it first looks at the expression to the *left* of the question mark. If **denom** equals zero, then the expression *between* the question mark and the colon is evaluated and used as the value of the entire **?** expression. If **denom** does not equal zero, then the expression *after* the colon is evaluated and used for the value of the

entire **?** expression. The result produced by the **?** operator is then assigned to **ratio**.

Here is a program that demonstrates the **?** operator. It uses it to obtain the absolute value of a variable.

```
// Demonstrate ?.
class Ternary {
  public static void main(String[] args) {
    int i, k;

    i = 10;
    k = i < 0 ? -i : i; // get absolute value of i
    System.out.print("Absolute value of ");
    System.out.println(i + " is " + k);

    i = -10;
    k = i < 0 ? -i : i; // get absolute value of i
    System.out.print("Absolute value of ");
    System.out.println(i + " is " + k);
  }
}
```

The output generated by the program is shown here:

```
Absolute value of 10 is 10
Absolute value of -10 is 10
```

Operator Precedence

Table 4-1 shows the order of precedence for Java operators, from highest to lowest. Operators in the same row are equal in precedence. In binary operations, the order of evaluation is left to right (except for assignment, which evaluates right to left). Although they are technically separators, the **[ ]**, **( )**, and **.** can also act like operators. In that capacity, they would have the highest precedence. Also, notice the arrow operator (->). It is used in lambda expressions.

Highest						
++ (postfix)	– – (postfix)					
++ (prefix)	– – (prefix)	~	!	+ (unary)	– (unary)	(*type-cast*)
*	/	%				
+	–					
>>	>>>	<<				
>	>=	<	<=	instanceof		
==	!=					
&						
^						
|						
&&						
||						
?:						
->						
=	op=					
Lowest						

Table 4-1 The Precedence of the Java Operators

Using Parentheses

Parentheses raise the precedence of the operations that are inside them. This is often necessary to obtain the result you desire. For example, consider the following expression:

```
a >> b + 3
```

This expression first adds 3 to **b** and then shifts **a** right by that result. That is, this expression can be rewritten using redundant parentheses like this:

```
a >> (b + 3)
```

However, if you want to first shift **a** right by **b** positions and then add 3 to that result, you will need to parenthesize the expression like this:

```
(a >> b) + 3
```

In addition to altering the normal precedence of an operator, parentheses can sometimes be used to help clarify the meaning of an expression. For anyone reading your code, a complicated expression can be difficult to understand. Adding redundant but clarifying parentheses to complex expressions can help prevent confusion later. For example, which of the following expressions is easier to read?

```
a | 4 + c >> b & 7
(a | (((4 + c) >> b) & 7))
```

One other point: parentheses (redundant or not) do not degrade the performance of your program. Therefore, adding parentheses to reduce ambiguity does not negatively affect your program.

CHAPTER

5

Control Statements

A programming language uses *control* statements to cause the flow of execution to advance and branch based on changes to the state of a program. Java's program control statements can be put into the following categories: selection, iteration, and jump. *Selection* statements allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable. *Iteration* statements enable program execution to repeat one or more statements (that is, iteration statements form loops). *Jump* statements allow your program to execute in a nonlinear fashion. All of Java's control statements are examined here.

Java's Selection Statements

Java supports two selection statements: **if** and **switch**. These statements allow you to control the flow of your program's execution based upon conditions known only during run time. You will be pleasantly surprised by the power and flexibility contained in these two statements.

if

The **if** statement was introduced in Chapter 2. It is examined in detail here. The **if** statement is Java's conditional branch statement. It can be used to route program execution through two different paths. Here is the general form of the **if** statement:

```
if (condition) statement1;
else statement2;
```

Here, each *statement* may be a single statement or a compound statement enclosed in curly braces (that is, a *block*). The *condition* is any expression that returns a **boolean** value. The **else** clause is optional.

The **if** works like this: If the *condition* is true, then *statement1* is executed. Otherwise, *statement2* (if it exists) is executed. In no case will both statements be executed. For example, consider the following:

```
int a, b;
//...
if(a < b) a = 0;
else b = 0;
```

Here, if **a** is less than **b**, then **a** is set to zero. Otherwise, **b** is set to zero. In no case are they both set to zero.

Most often, the expression used to control the **if** will involve the relational operators. However, this is not technically necessary. It is possible to control the **if** using a single **boolean** variable, as shown in this code fragment:

```
boolean dataAvailable;
//...
if (dataAvailable)
  ProcessData();
else
  waitForMoreData();
```

Remember, only one statement can appear directly after the **if** or the **else**. If you want to include more statements, you'll need to create a block, as in this fragment:

```
int bytesAvailable;
// ...
if (bytesAvailable > 0) {
  ProcessData();
  bytesAvailable -= n;
} else
  waitForMoreData();
```

Here, both statements within the **if** block will execute if **bytesAvailable** is greater than zero.

Some programmers find it convenient to include the curly braces when using the **if**, even when there is only one statement in each clause. This makes it easy to add another statement at a later date, and you don't have to worry about forgetting the braces. In fact, forgetting to define a block when one is needed is a common cause of errors. For example, consider the following code fragment:

```
int bytesAvailable;
// ...
if (bytesAvailable > 0) {
  ProcessData();
  bytesAvailable -= n;
} else
  waitForMoreData();
  bytesAvailable = n;
```

It seems clear that the statement **bytesAvailable = n**; was intended to be executed inside the **else** clause, because of the indentation level. However, as you recall, whitespace is insignificant to Java, and there is no way for the compiler to know what was intended. This code will compile without complaint, but it will behave incorrectly when run. The preceding example is fixed in the code that follows:

```
int bytesAvailable;
// ...
if (bytesAvailable > 0) {
  ProcessData();
  bytesAvailable -= n;
} else {
```

```
  waitForMoreData();
  bytesAvailable = n;
}
```

Nested ifs

A *nested* **if** is an **if** statement that is the target of another **if** or **else**. Nested **if**s are very common in programming. When you nest **if**s, the main thing to remember is that an **else** statement always refers to the nearest **if** statement that is within the same block as the **else** and that is not already associated with an **else**. Here is an example:

```
if(i == 10) {
  if(j < 20) a = b;
  if(k > 100) c = d; // this if is
  else a = c;        // associated with this else
}
else a = d;          // this else refers to if(i == 10)
```

As the comments indicate, the final **else** is not associated with **if(j<20)** because it is not in the same block (even though it is the nearest **if** without an **else**). Rather, the final **else** is associated with **if(i==10)**. The inner **else** refers to **if(k>100)** because it is the closest **if** within the same block.

The if-else-if Ladder

A common programming construct that is based upon a sequence of nested **if**s is the *if-else-if* ladder. It looks like this:

```
if(condition)
  statement;
else if(condition)
  statement;
else if(condition)
  statement;
.
.
.
else
  statement;
```

The **if** statements are executed from the top down. As soon as one of the conditions controlling the **if** is **true**, the statement associated with that **if** is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final **else** statement will be executed. The final **else** acts as a default condition; that is, if all other conditional tests fail, then the last **else** statement is performed. If there is no final **else** and all other conditions are **false**, then no action will take place.

Here is a program that uses an **if-else-if** ladder to determine which season a particular month is in:

```
// Demonstrate if-else-if statements.
class IfElse {
  public static void main(String[] args) {
```

```
    int month = 4; // April
    String season;

    if(month == 12 || month == 1 || month == 2)
      season = "Winter";
    else if(month == 3 || month == 4 || month == 5)
      season = "Spring";
    else if(month == 6 || month == 7 || month == 8)
      season = "Summer";
    else if(month == 9 || month == 10 || month == 11)
      season = "Autumn";
    else
      season = "Bogus Month";

    System.out.println("April is in the " + season + ".");
  }
}
```

Here is the output produced by the program:

```
April is in the Spring.
```

You might want to experiment with this program before moving on. As you will find, no matter what value you give **month**, one and only one assignment statement within the ladder will be executed.

The Traditional switch

The **switch** statement is Java's multiway branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression. As such, it often provides a better alternative than a large series of **if-else-if** statements.

At the outset, it is necessary to state that beginning with JDK 14, the **switch** has been significantly enhanced and expanded with several new features that go far beyond its traditional form. The traditional form of **switch** has been part of Java from the beginning and is, therefore, in widespread use. Furthermore, it is the form that will work in all Java development environments and for all readers. Because of the substantial nature of the recent **switch** enhancements, they are described in Chapter 17,

in the context of other recent additions to Java. Here, the traditional form of the **switch** is examined. Here is the general form of a traditional **switch** statement:

```
switch (expression) {
  case value1:
    // statement sequence
    break;
  case value2:
    // statement sequence
    break;
.
.
  case valueN:
    // statement sequence
    break;
  default:
    // default statement sequence
}
```

For versions of Java prior to JDK 7, *expression* must resolve to type **byte**, **short**, **int**, **char**, or an enumeration. (Enumerations are described in Chapter 12.) Today, *expression* can also be of type **String**, and beginning with JDK 21, in some situations, it can be an object reference (see Chapter 17). Each value specified in the **case** statements must be a unique constant expression (such as a literal value). Duplicate **case** values are not allowed. The type of each value must be compatible with the type of *expression*.

The traditional **switch** statement works like this: The value of the expression is compared with each of the values in the **case** statements. If a match is found, the code sequence following that **case** statement is executed. If none of the constants matches the value of the expression, then the **default** statement is executed. However, the **default** statement is optional. If no **case** matches and no **default** is present, then no further action is taken.

The **break** statement is used inside the **switch** to terminate a statement sequence. When a **break** statement is encountered, execution branches to the first line of code that follows the entire **switch** statement. This has the effect of "jumping out" of the **switch**.

Here is a simple example that uses a **switch** statement:

```
// A simple example of the switch.
class SampleSwitch {
  public static void main(String[] args) {
    for(int i=0; i<6; i++)
      switch(i) {
        case 0:
          System.out.println("i is zero.");
          break;
        case 1:
          System.out.println("i is one.");
          break;
        case 2:
          System.out.println("i is two.");
          break;
        case 3:
          System.out.println("i is three.");
          break;
        default:
          System.out.println("i is greater than 3.");
      }
  }
}
```

The output produced by this program is shown here:

```
i is zero.
i is one.
i is two.

i is three.
i is greater than 3.
i is greater than 3.
```

As you can see, each time through the loop, the statements associated with the **case** constant that matches **i** are executed. All others are bypassed. After **i** is greater than 3, no **case** statements match, so the **default** statement is executed.

The **break** statement is optional. If you omit the **break**, execution will continue on into the next **case**. It is sometimes desirable to have multiple **case**s without **break** statements between them. For example, consider the following program:

```
// In a switch, break statements are optional.
class MissingBreak {
  public static void main(String[] args) {
    for(int i=0; i<12; i++)
      switch(i) {
        case 0:
        case 1:
        case 2:
        case 3:
        case 4:
          System.out.println("i is less than 5");
          break;
        case 5:
        case 6:
        case 7:
        case 8:
        case 9:
          System.out.println("i is less than 10");
          break;
        default:
          System.out.println("i is 10 or more");
      }
  }
}
```

This program generates the following output:

```
i is less than 5
i is less than 5
i is less than 5
i is less than 5
i is less than 5
i is less than 10
i is less than 10
i is less than 10
i is less than 10
i is less than 10
i is 10 or more
i is 10 or more
```

As you can see, execution falls through each **case** until a **break** statement (or the end of the **switch**) is reached.

While the preceding example is, of course, contrived for the sake of illustration, omitting the **break** statement has many practical applications in real programs. To sample its more realistic usage, consider the following rewrite of the season example shown earlier. This version uses a **switch** to provide a more efficient implementation.

```
// An improved version of the season program.
class Switch {
  public static void main(String[] args) {
    int month = 4;
    String season;

    switch (month) {
      case 12:
      case 1:
      case 2:
        season = "Winter";
        break;
      case 3:
      case 4:
      case 5:
        season = "Spring";
        break;
      case 6:
      case 7:
      case 8:
        season = "Summer";
        break;
      case 9:
      case 10:
      case 11:
        season = "Autumn";
        break;
      default:
        season = "Bogus Month";
    }
    System.out.println("April is in the " + season + ".");
  }
}
```

As mentioned, you can also use a string to control a **switch** statement. For example:

```
// Use a string to control a switch statement.

class StringSwitch {
  public static void main(String[] args) {

    String str = "two";

    switch(str) {
      case "one":
        System.out.println("one");
        break;
      case "two":
        System.out.println("two");
        break;
      case "three":

        System.out.println("three");
        break;
      default:
        System.out.println("no match");
        break;
    }
  }
}
```

As you would expect, the output from the program is

```
two
```

The string contained in **str** (which is "two" in this program) is tested against the **case** constants. When a match is found (as it is in the second **case**), the code sequence associated with that sequence is executed.

Being able to use strings in a **switch** statement streamlines many situations. For example, using a string-based **switch** is an improvement over using the equivalent sequence of **if/else** statements. However, switching on strings can be more expensive than switching on integers. Therefore, it is best to switch on strings only in cases in which the controlling data is already in string form. In other words, don’t use strings in a **switch** unnecessarily.

Nested switch Statements

You can use a **switch** as part of the statement sequence of an outer **switch**. This is called a *nested* **switch**. Since a **switch** statement defines its own block, no conflicts arise between the **case** constants in the inner **switch** and those in the outer **switch**. For example, the following fragment is perfectly valid:

```
switch(count) {
  case 1:
    switch(target) { // nested switch
      case 0:
        System.out.println("target is zero");
        break;
      case 1: // no conflicts with outer switch
        System.out.println("target is one");
        break;
    }
    break;
  case 2: // ...
```

Here, the **case 1:** statement in the inner switch does not conflict with the **case 1:** statement in the outer switch. The **count** variable is compared only with the list of cases at the outer level. If **count** is 1, then **target** is compared with the inner list cases.

In summary, there are three important features of the **switch** statement to note:

- The **switch** differs from the **if** in that **switch** can only test for equality, whereas **if** can evaluate any type of Boolean expression. That is, the **switch** looks only for a match between the value of the expression and one of its **case** constants.
- No two **case** constants in the same **switch** can have identical values. Of course, a **switch** statement and an enclosing outer **switch** can have **case** constants in common.
- A **switch** statement is usually more efficient than a set of nested **if**s.

The last point is particularly interesting because it gives insight into how the Java compiler works. When it compiles a **switch** statement, the Java

compiler will inspect each of the **case** constants and create a "jump table" that it will use for selecting the path of execution depending on the value of the expression. Therefore, if you need to select among a large group of values, a **switch** statement will run much faster than the equivalent logic coded using a sequence of **if-else**s. The compiler can do this because it knows that the **case** constants are all the same type and simply must be compared for equality with the **switch** expression. The compiler has no such knowledge of a long list of **if** expressions.

REMEMBER Recently, the capabilities and features of **switch** have been substantially expanded beyond those offered by the traditional **switch** just described. Refer to Chapter 17 for details on the enhanced **switch**.

Iteration Statements

Java's iteration statements are **for**, **while**, and **do-while**. These statements create what we commonly call *loops*. As you probably know, a loop repeatedly executes the same set of instructions until a termination condition is met. As you will see, Java has a loop to fit any programming need.

while

The **while** loop is Java's most fundamental loop statement. It repeats a statement or block while its controlling expression is true. Here is its general form:

```
while(condition) {
  // body of loop
}
```

The *condition* can be any Boolean expression. The body of the loop will be executed as long as the conditional expression is true. When *condition* becomes false, control passes to the next line of code immediately following the loop. The curly braces are unnecessary if only a single statement is being repeated.

Here is a **while** loop that counts down from 10, printing exactly ten lines of "tick":

```
// Demonstrate the while loop.
class While {
  public static void main(String[] args) {
    int n = 10;

    while(n > 0) {
      System.out.println("tick " + n);
      n--;
    }
  }
}
```

When you run this program, it will "tick" ten times:

```
tick 10
tick 9
tick 8
tick 7
tick 6
tick 5
tick 4
tick 3
tick 2
tick 1
```

Since the **while** loop evaluates its conditional expression at the top of the loop, the body of the loop will not execute even once if the condition is false to begin with. For example, in the following fragment, the call to **println()** is never executed:

```
int a = 10, b = 20;

while(a > b)
  System.out.println("This will not be displayed");
```

The body of the **while** (or any other of Java's loops) can be empty. This is because a *null statement* (one that consists only of a semicolon) is syntactically valid in Java. For example, consider the following program:

```
// The target of a loop can be empty.
class NoBody {
  public static void main(String[] args) {
    int i, j;

    i = 100;
    j = 200;

    // find midpoint between i and j
    while(++i < --j); // no body in this loop

    System.out.println("Midpoint is " + i);
  }
}
```

This program finds the midpoint between **i** and **j**. It generates the following output:

```
Midpoint is 150
```

Here is how this **while** loop works. The value of **i** is incremented, and the value of **j** is decremented. These values are then compared with one another. If the new value of **i** is still less than the new value of **j**, then the loop repeats. If **i** is equal to or greater than **j**, the loop stops. Upon exit from the loop, **i** will hold a value that is midway between the original values of **i** and **j**. (Of course, this procedure only works when **i** is less than **j** to begin with.) As you can see, there is no need for a loop body; all of the action occurs within the conditional expression itself. In professionally written Java code, short loops are frequently coded without bodies when the controlling expression can handle all of the details itself.

do-while

As you just saw, if the conditional expression controlling a **while** loop is initially false, then the body of the loop will not be executed at all. However, sometimes it is desirable to execute the body of a loop at least once, even if the conditional expression is false to begin with. In other words, there are times when you would like to test the termination expression at the end of the loop rather than at the beginning. Fortunately,

Java supplies a loop that does just that: the **do-while**. The **do-while** loop always executes its body at least once, because its conditional expression is at the bottom of the loop. Its general form is

```
do {
  // body of loop
} while (condition);
```

Each iteration of the **do-while** loop first executes the body of the loop and then evaluates the conditional expression. If this expression is true, the loop will repeat. Otherwise, the loop terminates. As with all of Java's loops, *condition* must be a Boolean expression.

Here is a reworked version of the "tick" program that demonstrates the **do-while** loop. It generates the same output as before.

```
// Demonstrate the do-while loop.
class DoWhile {
  public static void main(String[] args) {
    int n = 10;

    do {
      System.out.println("tick " + n);
      n--;
    } while(n > 0);
  }
}
```

The loop in the preceding program, while technically correct, can be written more efficiently as follows:

```
do {
  System.out.println("tick " + n);
} while(--n > 0);
```

In this example, the expression **(– –n > 0)** combines the decrement of **n** and the test for zero into one expression. Here is how it works. First, the **– –n** statement executes, decrementing **n** and returning the new value of **n**. This value is then compared with zero. If it is greater than zero, the loop continues; otherwise, it terminates.

The **do-while** loop is especially useful when you process a menu selection, because you will usually want the body of a menu loop to execute at least once. Consider the following program, which implements a very simple help system for Java's selection and iteration statements:

```
// Using a do-while to process a menu selection
class Menu {
  public static void main(String[] args)
    throws java.io.IOException {
    char choice;

    do {
      System.out.println("Help on: ");
      System.out.println("  1. if");
      System.out.println("  2. switch");
      System.out.println("  3. while");
      System.out.println("  4. do-while");
      System.out.println("  5. for\n");
      System.out.println("Choose one:");
      choice = (char) System.in.read();
    } while( choice < '1' || choice > '5');

    System.out.println("\n");

    switch(choice) {
      case '1':
        System.out.println("The if:\n");
        System.out.println("if(condition) statement;");
        System.out.println("else statement;");
        break;
      case '2':
        System.out.println("The switch:\n");
        System.out.println("switch(expression) {");
        System.out.println("  case constant:");
        System.out.println("    statement sequence");
        System.out.println("    break;");
        System.out.println("  //...");
        System.out.println("}");
        break;
      case '3':
        System.out.println("The while:\n");
        System.out.println("while(condition) statement;");
        break;
      case '4':
        System.out.println("The do-while:\n");
        System.out.println("do {");
        System.out.println("  statement;");
        System.out.println("} while (condition);");
        break;
```

Here is a sample run produced by this program:

```
        System.out.println("The for:\n");
    Help on:              :("for(init; condition; iteration)");
      1. if               :ln(" statement;");
      2. switch
      3. while
    } 4. do-while
}     5. for
    Choose one:
    4
    The do-while:
    do {
      statement;
    } while (condition);
```

In the program, the **do-while** loop is used to verify that the user has entered a valid choice. If not, then the user is reprompted. Since the menu must be displayed at least once, the **do-while** is the perfect loop to accomplish this.

A few other points about this example: Notice that characters are read from the keyboard by calling **System.in.read( )**. This is one of Java's console input functions. Although Java's console I/O methods won't be discussed in detail until Chapter 13, **System.in.read( )** is used here to obtain the user's choice. It reads characters from standard input (returned as integers, which is why the return value was cast to **char**). By default, standard input is line buffered, so you must press ENTER before any characters that you type will be sent to your program.

Java's console input can be a bit awkward to work with. Further, most real-world Java programs will use a graphical user interface (GUI). For these reasons, console input is not used much in this book. However, it is useful in this context. One other point to consider: Because **System.in.read( )** is being used, the program must specify the **throws java.io.IOException** clause. This line is necessary to handle input errors. It is part of Java's exception handling features, which are discussed in Chapter 10.

for

You were introduced to a simple form of the **for** loop in Chapter 2. As you will see, it is a powerful and versatile construct.

There are two forms of the **for** loop. The first is the traditional form that has been in use since the original version of Java. The second is the newer "for-each" form, added by JDK 5. Both types of **for** loops are discussed here, beginning with the traditional form.

Here is the general form of the traditional **for** statement:

```
for(initialization; condition; iteration) {
  // body
}
```

If only one statement is being repeated, there is no need for the curly braces.

The **for** loop operates as follows. When the loop first starts, the *initialization* portion of the loop is executed. Generally, this is an expression that sets the value of the *loop control variable*, which acts as a counter that controls the loop. It is important to understand that the initialization expression is executed only once. Next, *condition* is evaluated. This must be a Boolean expression. It usually tests the loop control variable against a target value. If this expression is true, then the body of the loop is executed. If it is false, the loop terminates. Next, the *iteration* portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable. The loop then iterates, first evaluating the conditional expression, then executing the body of the loop, and then executing the iteration expression with each pass. This process repeats until the controlling expression is false.

Here is a version of the "tick" program that uses a **for** loop:

```
// Demonstrate the for loop.
class ForTick {
  public static void main(String[] args) {
    int n;

    for(n=10; n>0; n--)
      System.out.println("tick " + n);
  }
}
```

Declaring Loop Control Variables Inside the for Loop

Often the variable that controls a **for** loop is needed only for the purposes of the loop and is not used elsewhere. When this is the case, it is possible to declare the variable inside the initialization portion of the **for**. For example, here is the preceding program recoded so that the loop control variable **n** is declared as an **int** inside the **for**:

```
// Declare a loop control variable inside the for.
class ForTick {
  public static void main(String[] args) {

    // here, n is declared inside of the for loop
    for(int n=10; n>0; n--)
      System.out.println("tick " + n);
  }
}
```

When you declare a variable inside a **for** loop, there is one important point to remember: the scope of that variable ends when the **for** statement does. (That is, the scope of the variable is limited to the **for** loop.) Outside the **for** loop, the variable will cease to exist. If you need to use the loop control variable elsewhere in your program, you will not be able to declare it inside the **for** loop.

When the loop control variable will not be needed elsewhere, most Java programmers declare it inside the **for**. For example, here is a simple program that tests for prime numbers. Notice that the loop control variable, **i**, is declared inside the **for** since it is not needed elsewhere.

```
// Test for primes.
class FindPrime {
  public static void main(String[] args) {
    int num;
    boolean isPrime;

    num = 14;
```

```
    if(num < 2) isPrime = false;
    else isPrime = true;

    for(int i=2; i <= num/i; i++) {
      if((num % i) == 0) {
        isPrime = false;
        break;
      }
    }

    if(isPrime) System.out.println("Prime");
    else System.out.println("Not Prime");
  }
}
```

Using the Comma

There will be times when you will want to include more than one statement in the initialization and iteration portions of the **for** loop. For example, consider the loop in the following program:

```
class Sample {
  public static void main(String[] args) {
    int a, b;

    b = 4;
    for(a=1; a<b; a++) {
      System.out.println("a = " + a);
      System.out.println("b = " + b);
      b--;
    }
  }
}
```

As you can see, the loop is controlled by the interaction of two variables. Since the loop is governed by two variables, it would be useful if both could be included in the **for** statement itself, instead of **b** being handled manually. Fortunately, Java provides a way to accomplish this. To allow two or more variables to control a **for** loop, Java permits you to include

multiple statements in both the initialization and iteration portions of the **for**. Each statement is separated from the next by a comma.

Using the comma, the preceding **for** loop can be more efficiently coded, as shown here:

```
// Using the comma.
class Comma {
  public static void main(String[] args) {
    int a, b;

    for(a=1, b=4; a<b; a++, b--) {
      System.out.println("a = " + a);
      System.out.println("b = " + b);
    }
  }
}
```

In this example, the initialization portion sets the values of both **a** and **b**. The two comma-separated statements in the iteration portion are executed each time the loop repeats. The program generates the following output:

```
a = 1
b = 4
a = 2
b = 3
```

Some for Loop Variations

The **for** loop supports a number of variations that increase its power and applicability. The reason it is so flexible is that its three parts—the initialization, the conditional test, and the iteration—do not need to be used for only those purposes. In fact, the three sections of the **for** can be used for any purpose you desire. Let's look at some examples.

One of the most common variations involves the conditional expression. Specifically, this expression does not need to test the loop control variable against some target value. In fact, the condition controlling the **for** can be any Boolean expression. For example, consider the following fragment:

```
boolean done = false;

for(int i=1; !done; i++) {
  // ...
  if(interrupted()) done = true;
}
```

In this example, the **for** loop continues to run until the **boolean** variable **done** is set to **true**. It does not test the value of **i**.

Here is another interesting **for** loop variation. Either the initialization or the iteration expression or both may be absent, as in this next program:

```
// Parts of the for loop can be empty.
class ForVar {
  public static void main(String[] args) {
    int i;
    boolean done = false;

    i = 0;
    for( ; !done; ) {
      System.out.println("i is " + i);
      if(i == 10) done = true;
      i++;
    }
  }
}
```

Here, the initialization and iteration expressions have been moved out of the **for**. Thus, parts of the **for** are empty. While this is of no value in this simple example—indeed, it would be considered quite poor style—there can be times when this type of approach makes sense. For example, if the initial condition is set through a complex expression elsewhere in the program or if the loop control variable changes in a nonsequential manner determined by actions that occur within the body of the loop, it may be appropriate to leave these parts of the **for** empty.

Here is one more **for** loop variation. You can intentionally create an infinite loop (a loop that never terminates) if you leave all three parts of the **for** empty. For example:

```
for( ; ; ) {
  // ...
}
```

This loop will run forever because there is no condition under which it will terminate. Although there are some programs, such as operating system command processors, that require an infinite loop, most "infinite loops" are really just loops with special termination requirements. As you will soon see, there is a way to terminate a loop—even an infinite loop like the one shown—that does not make use of the normal loop conditional expression.

The For-Each Version of the for Loop

A second form of **for** implements a "for-each" style loop. As you may know, contemporary language theory has embraced the for-each concept, and it has become a standard feature that programmers have come to expect. A for-each style loop is designed to cycle through a collection of objects, such as an array, in strictly sequential fashion, from start to finish. In Java, the for-each style of **for** is also referred to as the *enhanced* **for** loop.

The general form of the for-each version of the **for** is shown here:

for(*type itr-var* : *collection*) *statement-block*

Here, *type* specifies the type and *itr-var* specifies the name of an *iteration variable* that will receive the elements from a collection, one at a time, from beginning to end. The collection being cycled through is specified by *collection*. There are various types of collections that can be used with the **for**, but the only type used in this chapter is the array. (Other types of collections that can be used with the **for**, such as those defined by the Collections Framework, are discussed later in this book.) With each iteration of the loop, the next element in the collection is retrieved and stored in *itr-var*. The loop repeats until all elements in the collection have been obtained.

Because the iteration variable receives values from the collection, *type* must be the same as (or compatible with) the elements stored in the collection. Thus, when iterating over arrays, *type* must be compatible with the element type of the array.

To understand the motivation behind a for-each style loop, consider the type of **for** loop that it is designed to replace. The following fragment uses a traditional **for** loop to compute the sum of the values in an array:

```
int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;

for(int i=0; i < 10; i++) sum += nums[i];
```

To compute the sum, each element in **nums** is read, in order, from start to finish. Thus, the entire array is read in strictly sequential order. This is accomplished by manually indexing the **nums** array by **i**, the loop control variable.

The for-each style **for** automates the preceding loop. Specifically, it eliminates the need to establish a loop counter, specify a starting and ending value, and manually index the array. Instead, it automatically cycles through the entire array, obtaining one element at a time, in sequence, from beginning to end. For example, here is the preceding fragment rewritten using a for-each version of the **for**:

```
int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;

for(int x: nums) sum += x;
```

With each pass through the loop, **x** is automatically given a value equal to the next element in **nums**. Thus, on the first iteration, **x** contains 1; on the second iteration, **x** contains 2; and so on. Not only is the syntax streamlined, but it also prevents boundary errors.

Here is an entire program that demonstrates the for-each version of the **for** just described:

```
// Use a for-each style for loop.
class ForEach {
  public static void main(String[] args) {
    int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int sum = 0;

    // use for-each style for to display and sum the values
    for(int x : nums) {
      System.out.println("Value is: " + x);
      sum += x;
    }

    System.out.println("Summation: " + sum);
  }
}
```

The output from the program is shown here:

```
Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Value is: 6
Value is: 7
Value is: 8
Value is: 9
Value is: 10
Summation: 55
```

As this output shows, the for-each style **for** automatically cycles through an array in sequence from the lowest index to the highest.

Although the for-each **for** loop iterates until all elements in an array have been examined, it is possible to terminate the loop early by using a **break** statement. For example, this program sums only the first five elements of **nums**:

```
// Use break with a for-each style for.
class ForEach2 {
  public static void main(String[] args) {
    int sum = 0;
    int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    // use for to display and sum the values
    for(int x : nums) {
      System.out.println("Value is: " + x);
      sum += x;
      if(x == 5) break; // stop the loop when 5 is obtained
    }
    System.out.println("Summation of first 5 elements: " + sum);
  }
}
```

This is the output produced:

```
Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Summation of first 5 elements: 15
```

As is evident, the **for** loop stops after the fifth element has been obtained. The **break** statement can also be used with Java's other loops, and it is discussed in detail later in this chapter.

There is one important point to understand about the for-each style loop. Its iteration variable is "read-only" as it relates to the underlying array. An assignment to the iteration variable has no effect on the underlying array. In other words, you can't change the contents of the array by assigning the iteration variable a new value. For example, consider this program:

```
// The for-each loop is essentially read-only.
class NoChange {
  public static void main(String[] args) {
    int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    for(int x: nums) {
      System.out.print(x + " ");
      x = x * 10; // no effect on nums
    }

    System.out.println();

    for(int x : nums)
      System.out.print(x + " ");

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

The first **for** loop increases the value of the iteration variable by a factor of 10. However, this assignment has no effect on the underlying array **nums**, as the second **for** loop illustrates. The output, shown here, proves this point:

```
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
```

Iterating Over Multidimensional Arrays

The enhanced version of the **for** also works on multidimensional arrays. Remember, however, that in Java, multidimensional arrays consist of *arrays of arrays*. (For example, a two-dimensional array is an array of one-dimensional arrays.) This is important when iterating over a multidimensional array, because each iteration obtains the *next array*, not an individual element. Furthermore, the iteration variable in the **for** loop must be compatible with the type of array being obtained. For example, in the case of a two-dimensional array, the iteration variable must be a reference to a one-dimensional array. In general, when using the for-each **for** to iterate over an array of *N* dimensions, the objects obtained will be arrays of *N*–1 dimensions. To understand the implications of this, consider

the following program. It uses nested **for** loops to obtain the elements of a two-dimensional array in row-order, from first to last.

```
// Use for-each style for on a two-dimensional array.
class ForEach3 {
  public static void main(String[] args) {
    int sum = 0;
    int[][] nums = new int[3][5];

    // give nums some values
    for(int i = 0; i < 3; i++)
      for(int j = 0; j < 5; j++)
        nums[i][j] = (i+1)*(j+1);

    // use for-each for to display and sum the values
    for(int[] x : nums) {
      for(int y : x) {
        System.out.println("Value is: " + y);
        sum += y;
      }
    }
    System.out.println("Summation: " + sum);
  }
}
```

The output from this program is shown here:

```
Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Value is: 2
Value is: 4
Value is: 6
Value is: 8
Value is: 10
Value is: 3
Value is: 6
Value is: 9
Value is: 12
Value is: 15
Summation: 90
```

In the program, pay special attention to this line:

```
for(int[] x: nums) {
```

Notice how **x** is declared. It is a reference to a one-dimensional array of integers. This is necessary because each iteration of the **for** obtains the next *array* in **nums**, beginning with the array specified by **nums[0]**. The inner **for** loop then cycles through each of these arrays, displaying the values of each element.

Applying the Enhanced for

Since the for-each style **for** can only cycle through an array sequentially, from start to finish, you might think that its use is limited, but this is not true. A large number of algorithms require exactly this mechanism. One of the most common is searching. For example, the following program uses a **for** loop to search an unsorted array for a value. It stops if the value is found.

```
// Search an array using for-each style for.
class Search {
  public static void main(String[] args) {
    int[] nums = { 6, 8, 3, 7, 5, 6, 1, 4 };
    int val = 5;
    boolean found = false;

    // use for-each style for to search nums for val
    for(int x : nums) {
      if(x == val) {
        found = true;
        break;
      }
    }

    if(found)
      System.out.println("Value found!");
  }
}
```

The for-each style **for** is an excellent choice in this application because searching an unsorted array involves examining each element in sequence. (Of course, if the array were sorted, a binary search could be used, which would require a different style loop.) Other types of applications that benefit from for-each style loops include computing an average, finding the minimum or maximum of a set, looking for duplicates, and so on.

Although we have been using arrays in the examples in this chapter, the for-each style **for** is especially useful when operating on collections defined by the Collections Framework, which is described in Part II. More generally, the **for** can cycle through the elements of any collection of objects, as long as that collection satisfies a certain set of constraints, which are described in Chapter 20.

Local Variable Type Inference in a for Loop

As explained in Chapter 3, JDK 10 introduced a feature called *local variable type inference*, which allows the type of a local variable to be inferred from the type of its initializer. To use local variable type inference, the type of the variable is specified as **var** and the variable must be

initialized. Local variable type inference can be used in a **for** loop when declaring and initializing the loop control variable inside a traditional **for** loop, or when specifying the iteration variable in a for-each **for**. The following program shows an example of each case:

```
// Use type inference in a for loop.
class TypeInferenceInFor {
  public static void main(String[] args) {

    // Use type inference with the loop control variable.
    System.out.print("Values of x: ");
    for(var x = 2.5; x < 100.0; x = x * 2)
      System.out.print(x + " ");

    System.out.println();

    // Use type inference with the iteration variable.
    int[] nums = { 1, 2, 3, 4, 5, 6};
    System.out.print("Values in nums array: ");
    for(var v : nums)
      System.out.print(v + " ");

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

The output is shown here:

```
Values of x: 2.5 5.0 10.0 20.0 40.0 80.0
Values in nums array: 1 2 3 4 5 6
```

In this example, loop control variable **x** in this line:

```
for(var x = 2.5; x < 100.0; x = x * 2)
```

is inferred to be type **double** because that is the type of its initializer. Iteration variable **v** is this line:

```
for(var v : nums)
```

inferred to be of type **int** because that is the element type of the array **nums**.

One last point: Because a number of readers will be working in environments that predate JDK 10, local variable type inference will not be used by most of the **for** loops in the remainder of this edition of this book. You should, of course, consider it for new code that you write.

Nested Loops

Like all other programming languages, Java allows loops to be nested. That is, one loop may be inside another. For example, here is a program that nests **for** loops:

```
// Loops may be nested.
class Nested {
  public static void main(String[] args) {
    int i, j;

    for(i=0; i<10; i++) {
      for(j=i; j<10; j++)
        System.out.print(".");
      System.out.println();
    }
  }
}
```

The output produced by this program is shown here:

```
..........
.........
........
.......
......
.....
....
...
..
.
```

Jump Statements

Java supports three jump statements: **break**, **continue**, and **return**. These statements transfer control to another part of your program. Each is examined here.

NOTE In addition to the jump statements discussed here, Java supports one other way that you can change your program's flow of execution: through exception handling. Exception handling provides a structured method by which run-time errors can be trapped and handled by your program. It is supported by the keywords **try**, **catch**, **throw**, **throws**, and **finally**. In essence, the exception handling mechanism allows your program to perform a nonlocal branch. Since exception handling is a large topic, it is discussed in its own chapter, Chapter 10.

Using break

In Java, the **break** statement has three uses. First, as you have seen, it terminates a statement sequence in a **switch** statement. Second, it can be used to exit a loop. Third, it can be used as a "civilized" form of goto. The last two uses are explained here.

Using break to Exit a Loop

By using **break**, you can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop. When a **break** statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop. Here is a simple example:

```
// Using break to exit a loop.
class BreakLoop {
  public static void main(String[] args) {
    for(int i=0; i<100; i++) {
      if(i == 10) break; // terminate loop if i is 10
      System.out.println("i: " + i);
    }
    System.out.println("Loop complete.");
  }
}
```

This program generates the following output:

```
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
Loop complete.
```

As you can see, although the **for** loop is designed to run from 0 to 99, the **break** statement causes it to terminate early, when **i** equals 10.

The **break** statement can be used with any of Java's loops, including intentionally infinite loops. For example, here is the preceding program coded by use of a **while** loop. The output from this program is the same as just shown.

```
// Using break to exit a while loop.
class BreakLoop2 {
  public static void main(String[] args) {
    int i = 0;

    while(i < 100) {
      if(i == 10) break; // terminate loop if i is 10
      System.out.println("i: " + i);
      i++;
    }
    System.out.println("Loop complete.");
  }
}
```

When used inside a set of nested loops, the **break** statement will only break out of the innermost loop. For example:

```
// Using break with nested loops.
class BreakLoop3 {
  public static void main(String[] args) {
    for(int i=0; i<3; i++) {
      System.out.print("Pass " + i + ": ");
      for(int j=0; j<100; j++) {
        if(j == 10) break; // terminate loop if j is 10
        System.out.print(j + " ");
      }
      System.out.println();
    }
    System.out.println("Loops complete.");
  }
}
```

This program generates the following output:

```
Pass 0: 0 1 2 3 4 5 6 7 8 9
Pass 1: 0 1 2 3 4 5 6 7 8 9
Pass 2: 0 1 2 3 4 5 6 7 8 9
Loops complete.
```

As you can see, the **break** statement in the inner loop only causes termination of that loop. The outer loop is unaffected.

Here are two other points to remember about **break**. First, more than one **break** statement may appear in a loop. However, be careful. Too many **break** statements have the tendency to destructure your code. Second, the **break** that terminates a **switch** statement affects only that **switch** statement and not any enclosing loops.

REMEMBER break was not designed to provide the normal means by which a loop is terminated. The loop's conditional expression serves this purpose. The **break** statement should be used to cancel a loop only when some sort of special situation occurs.

Using break as a Form of Goto

In addition to its uses with the **switch** statement and loops, the **break** statement can also be employed by itself to provide a "civilized" form of the goto statement. Java does not have a goto statement because it provides a way to branch in an arbitrary and unstructured manner. This usually makes goto-ridden code hard to understand and hard to maintain. It also

prohibits certain compiler optimizations. There are, however, a few places where the goto is a valuable and legitimate construct for flow control. For example, the goto can be useful when you are exiting from a deeply nested set of loops. To handle such situations, Java defines an expanded form of the **break** statement. By using this form of **break**, you can, for example, break out of one or more blocks of code. These blocks need not be part of a loop or a **switch**. They can be any block. Further, you can specify precisely where execution will resume, because this form of **break** works with a label. As you will see, **break** gives you the benefits of a goto without its problems.

The general form of the labeled **break** statement is shown here:

break *label*;

Most often, *label* is the name of a label that identifies a block of code. This can be a stand-alone block of code but it can also be a block that is the target of another statement. When this form of **break** executes, control is transferred out of the named block. The labeled block must enclose the **break** statement, but it does not need to be the immediately enclosing block. This means, for example, that you can use a labeled **break** statement to exit from a set of nested blocks. But you cannot use **break** to transfer control out of a block that does not enclose the **break** statement.

To name a block, put a label at the start of it. A *label* is any valid Java identifier followed by a colon. Once you have labeled a block, you can then use this label as the target of a **break** statement. Doing so causes execution to resume at the *end* of the labeled block. For example, the following program shows three nested blocks, each with its own label. The **break** statement causes execution to jump forward, past the end of the block labeled **second**, skipping the two **println( )** statements.

```
// Using break as a civilized form of goto.
class Break {
  public static void main(String[] args) {
    boolean t = true;

    first: {
      second: {
        third: {
          System.out.println("Before the break.");
          if(t) break second; // break out of second block
          System.out.println("This won't execute");
        }
        System.out.println("This won't execute");
      }
      System.out.println("This is after second block.");
    }
  }
}
```

Running this program generates the following output:

```
Before the break.
This is after second block.
```

One of the most common uses for a labeled **break** statement is to exit from nested loops. For example, in the following program, the outer loop executes only once:

```
// Using break to exit from nested loops
class BreakLoop4 {
  public static void main(String[] args) {
    outer: for(int i=0; i<3; i++) {
      System.out.print("Pass " + i + ": ");
      for(int j=0; j<100; j++) {
        if(j == 10) break outer; // exit both loops
        System.out.print(j + " ");
```

```
      }
      System.out.println("This will not print");
    }
    System.out.println("Loops complete.");
  }
}
```

This program generates the following output:

```
Pass 0: 0 1 2 3 4 5 6 7 8 9 Loops complete.
```

As you can see, when the inner loop breaks to the outer loop, both loops have been terminated. Notice that this example labels the **for** statement, which has a block of code as its target.

Keep in mind that you cannot break to any label that is not defined for an enclosing block. For example, the following program is invalid and will not compile:

```
// This program contains an error.
class BreakErr {
  public static void main(String[] args) {

    one: for(int i=0; i<3; i++) {
      System.out.print("Pass " + i + ": ");
    }

    for(int j=0; j<100; j++) {
      if(j == 10) break one; // WRONG
      System.out.print(j + " ");
    }
  }
}
```

Since the loop labeled **one** does not enclose the **break** statement, it is not possible to transfer control out of that block.

Using continue

Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration. This is, in effect, a goto just past the body of the loop, to the loop's end. The **continue** statement performs such an action. In **while** and **do-while** loops, a **continue** statement causes control to be transferred directly to the conditional expression that controls the loop. In a **for** loop, control goes first to the iteration portion of the **for** statement and then to the conditional expression. For all three loops, any intermediate code is bypassed.

Here is a sample program that uses **continue** to cause two numbers to be printed on each line:

```
// Demonstrate continue.
class Continue {
  public static void main(String[] args) {
    for(int i=0; i<10; i++) {
      System.out.print(i + " ");

      if (i%2 == 0) continue;
      System.out.println("");
    }
  }
}
```

This code uses the **%** operator to check if **i** is even. If it is, the loop continues without printing a newline. Here is the output from this program:

```
0 1
2 3
4 5
6 7
8 9
```

As with the **break** statement, **continue** may specify a label to describe which enclosing loop to continue. Here is a sample program that uses **continue** to print a triangular multiplication table for 0 through 9:

```
// Using continue with a label.
class ContinueLabel {
  public static void main(String[] args) {
outer: for (int i=0; i<10; i++) {
         for(int j=0; j<10; j++) {
           if(j > i) {
             System.out.println();
             continue outer;
           }
           System.out.print(" " + (i * j));
         }
       }
       System.out.println();
  }
}
```

The **continue** statement in this example terminates the loop counting **j** and continues with the next iteration of the loop counting **i**. Here is the output of this program:

```
0
0 1
0 2 4
0 3 6 9
0 4 8 12 16
0 5 10 15 20 25
0 6 12 18 24 30 36
0 7 14 21 28 35 42 49
0 8 16 24 32 40 48 56 64
0 9 18 27 36 45 54 63 72 81
```

Good uses of **continue** are rare. One reason is that Java provides a rich set of loop statements that fit most applications. However, for those special circumstances in which early iteration is needed, the **continue** statement provides a structured way to accomplish it.

return

The last control statement is **return**. The **return** statement is used to explicitly return from a method. That is, it causes program control to

transfer back to the caller of the method. As such, it is categorized as a jump statement. Although a full discussion of **return** must wait until methods are discussed in Chapter 6, a brief look at **return** is presented here.

At any time in a method, the **return** statement can be used to cause execution to branch back to the caller of the method. Thus, the **return** statement immediately terminates the method in which it is executed. The following example illustrates this point. Here, **return** causes execution to return to the Java run-time system, since it is the run-time system that calls **main( )**:

```
// Demonstrate return.
class Return {
  public static void main(String[] args) {
    boolean t = true;

    System.out.println("Before the return.");

    if(t) return; // return to caller

    System.out.println("This won't execute.");
  }
}
```

The output from this program is shown here:

```
Before the return.
```

As you can see, the final **println( )** statement is not executed. As soon as **return** is executed, control passes back to the caller.

One last point: In the preceding program, the **if(t)** statement is necessary. Without it, the Java compiler would flag an "unreachable code" error because the compiler would know that the last **println( )** statement would never be executed. To prevent this error, the **if** statement is used here to trick the compiler for the sake of this demonstration.