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 occurs 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 probably 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, it is hardly a language that encourages 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) 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 in 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.

How Java Changed 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

An *applet* is a special kind of Java program that is designed to be transmitted over the Internet and automatically executed by a Java-compatible web browser. Furthermore, an applet is downloaded on demand, without further interaction with the user. If the user clicks a link that contains an applet, the applet will be automatically downloaded and run in the browser. Applets are intended to be small programs. They are 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 allows some functionality to be moved from the server to the client.

The creation of the applet changed Internet programming because it expanded the universe of objects that can 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.

As desirable as dynamic, networked programs are, they also present serious problems in the areas of security and portability. Obviously, a program that downloads and executes automatically 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.

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 applets to be downloaded and executed on the client computer safely, it was necessary to prevent an applet from launching such an attack.

Java achieved this protection by confining an applet to the Java execution environment and not allowing it access to other parts of the computer. (You will see how this is accomplished shortly.) The ability to download applets with confidence that no harm will be done and that no security will be breached 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. For example, in the case of an applet, the same applet must be able to be downloaded and executed by the wide variety of CPUs, operating systems, and browsers connected to the Internet. It is not practical to have different versions of the applet for different computers. The *same* 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 allows 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 the Java run-time system, which is called the *Java Virtual Machine (JVM)*. 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 the run-time package 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 can contain the program and prevent it from generating side effects outside of the system. As you will see, 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 it is not practical to compile an entire Java program into executable code all at once, because Java performs various run-time checks that can be done only at run time. 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.

Servlets: Java on the Server Side

As useful as applets can be, they are 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. The result was the *servlet*. A servlet is a small program that executes on the server. Just as applets dynamically extend the functionality of a web browser, servlets dynamically extend the functionality of a web server. Thus, with the advent of the servlet, Java spanned both sides of the client/server connection.

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 is available through mechanisms such as CGI (Common Gateway Interface), the servlet offers 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.

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 manages 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, are 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 all 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 elements, 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 runtime. 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 since 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

- Simplifying the creation and use of tasks that can execute concurrently
- Automatically making use of multiple processors

Therefore, by using the Fork/Join Framework, you can easily create scaleable 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.

Java SE 8

The newest release of Java is Java SE 8, with the developer's kit being called JDK 8. It has an internal version number of 1.8. JDK 8 represents a very 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 will 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 causes a new operator (the –>) and a new syntax element to be added to the language. Lambda expressions help ensure that Java will remain the vibrant, nimble language that users have come to expect.

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 is 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 very important new package is **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. JDK 8 also bundles support for JavaFX 8, the latest version of Java's new GUI application framework. JavaFX is expected to soon play an important part in nearly all Java applications, ultimately replacing Swing for most GUI-based projects. Part IV of this book provides an introduction to it.

In the final analysis, Java SE 8 is a major release that profoundly expands the capabilities of the language and changes the way that Java code is written. Its effects will be felt throughout the Java universe and for years to come. It truly is that important of a upgrade.

The material in this book has been updated to reflect Java SE 8, with many new features, updates, and additions indicated throughout.

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. The applet (and then the servlet) 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. Java SE 8 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 parts that form the car. 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 sound system consists of a radio, a CD player, and/or a tape or MP3 player. 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 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' 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).

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.

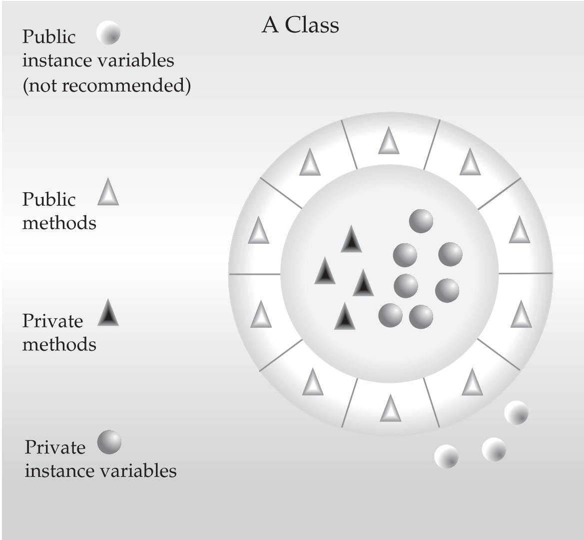
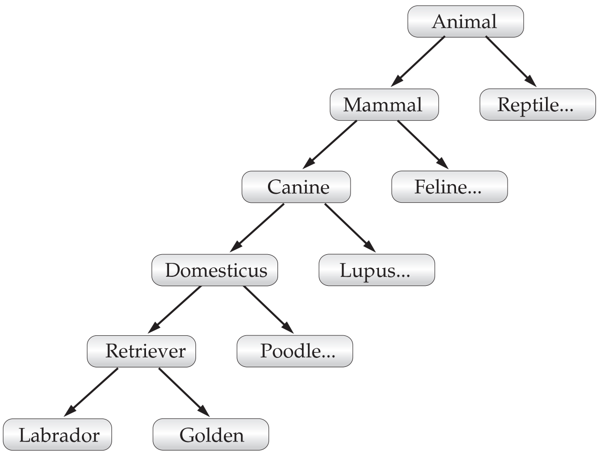


Figure 2-1 Encapsulation: public methods can be used to protect private data.

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.



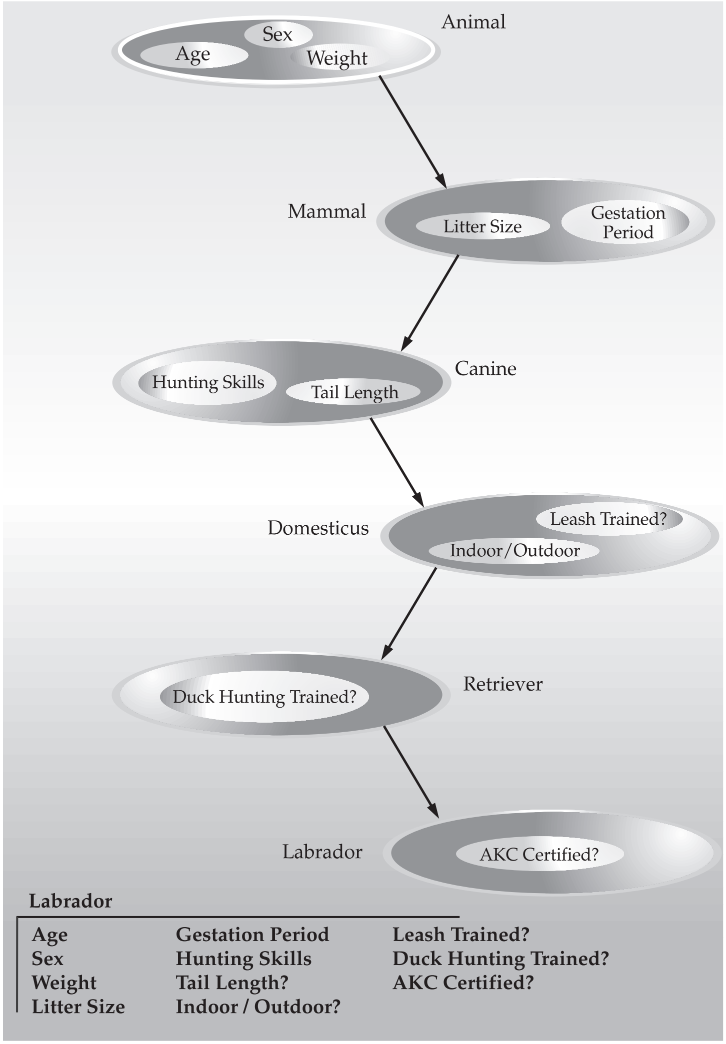


Figure 2-2 Labrador inherits the encapsulation of all its superclasses.

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 scaleable 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 4-, 6-, or 8-cylinder engines. 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 example 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 8 Development Kit (JDK 8), which is available from Oracle. 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 most 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.

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.

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. All Java applications begin 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 similar objects.) 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, in some cases, you won't need **main( )** at all. For example, when creating applets—Java programs that are embedded in web browsers—you won't use **main( )** since the web browser uses a different means of starting the execution of applets.

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 which 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 windowed and 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. All statements in Java end with a semicolon. The reason that the other lines in the program do not end in a semicolon is that they are not, technically, statements.

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 most 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 example 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. Further, it is syntactically identical to the **if** statements in C, C++, and C#. Its simplest form is shown here:

if*(condition) statement;*

Here, *condition* is a Boolean expression. 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 which 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

As you may know from your previous programming experience, loop statements are an important part of nearly any programming language. Java is no exception. In fact, 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, 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 which 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 is a space, tab, or newline.

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 8, the use of an underscore by itself as an identifier is not recommended.

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 the Appendix.

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 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. (Added by JDK 8.)

The Java Keywords

There are 50 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

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

Table 2-1 Java Keywords

of the Java language. These keywords cannot be used as identifiers. Thus, they cannot be used as names for a variable, class, or method.

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 the following: **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 example **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**. However, C/C++ programmers beware: **char** in Java is not the same as **char** in C or C++. In C/C++, **char** is 8 bits wide. This is *not* the case in Java. Instead, 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 65,536. 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, or 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 http://www.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, 0x7ffffffffffffffL or 9223372036854775807L is the largest **long**. An integer can also be assigned to a **char** as long as it is within range.

Beginning with JDK 7, 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.

Also beginning with JDK 7, 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.

Beginning with JDK 7, 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.

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

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

Table 3-1 Character Escape Sequences

```
"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 strings is that they must begin and end on the same line. There is no line-continuation escape sequence as there is in some other languages.

NOTE As you may know, in some other languages, including C/C++, 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.

Many other computer languages define 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. Although this book will look more closely at parameters in Chapter 6, for the sake of this discussion, they work the same as any other method variable.

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.

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.

NOTE If you are familiar with C/C++, be careful. Arrays in Java work differently than they do in those languages.

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 to be sure 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 actually arrays of arrays. These, as you might expect, look and act like regular multidimensional arrays. However, as you will see, there are a couple of subtle differences. 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.

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

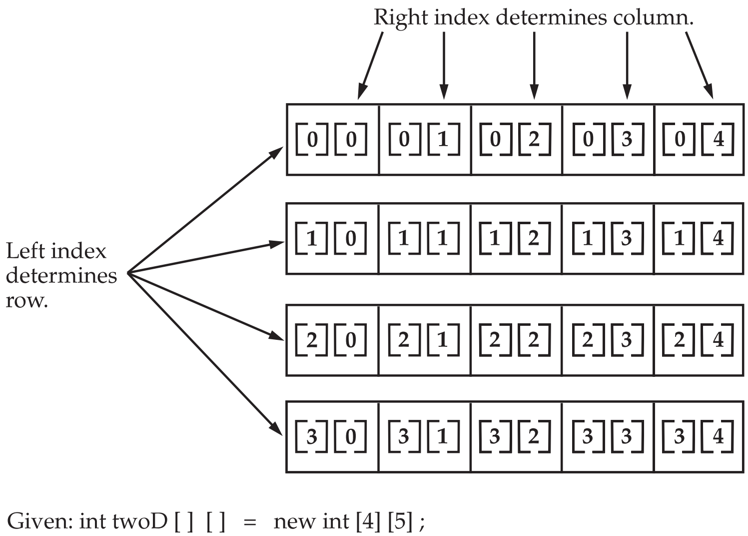


Figure 3-1 A conceptual view of a 4 by 5, two-dimensional array

separately. For example, this following code allocates memory for the first dimension of **twoD** when it is declared. It allocates the second dimension manually.

```
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 manually, 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++;
```

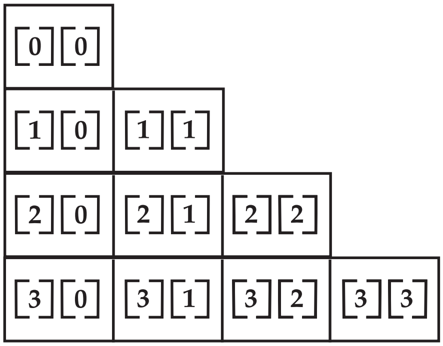
```
    }

    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 type specifier, and not the name of the array variable. For example, the following two declarations are equivalent:

```
int al[] = 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 declaring several arrays at the same time. For example,

```
int[] nums, nums2, nums3; // create three arrays
```

creates three array variables of type **int**. It is the same as writing

```
int nums[], nums2[], nums3[]; // create three arrays
```

The alternative declaration form is also useful when specifying an array as a return type for a method. Both forms are used in this book.

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 example 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.

A Note to C/C++ Programmers About Pointers

If you are an experienced C/C++ programmer, then you know that these languages provide support for pointers. However, no mention of pointers has been made in this chapter. The reason for this is simple: Java does not support or allow pointers. (Or more properly, Java does not support pointers that can be accessed and/or modified by the programmer.) Java cannot allow pointers, because doing so would allow Java programs to breach the firewall between the Java execution environment and the host computer. (Remember, a pointer can be given any address in memory—even addresses that might be outside the Java run-time system.) Since C/C++ make extensive use of pointers, you might be thinking that their loss is a significant disadvantage to Java. However, this is not true. Java is designed in such a way that as long as you stay within the confines of the execution environment, you will never need to use a pointer, nor would there be any benefit in using one.

This page has been intentionally left blank

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 new 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 example 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 example 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. 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
==	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 was added by JDK 8 and 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.

This page has been intentionally left blank

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.

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. Here is the general form of a **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 be of type **byte**, **short**, **int**, **char**, or an enumeration. (Enumerations are described in Chapter 12.) Beginning with JDK 7, *expression* can also be of type **String**. 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 **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, beginning with JDK 7, you can 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.

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;
      case '5':
        System.out.println("The for:\n");
        System.out.print("for(init; condition; iteration)");
        System.out.println(" statement;");
        break;
    }
  }
}
```

Here is a sample run produced by this program:

```
Help on:
  1. if
  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 be graphical and window-based. For these reasons, not much use of console input has been made 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.

Beginning with JDK 5, 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. 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
```

NOTE If you are familiar with C/C++, then you know that in those languages the comma is an operator that can be used in any valid expression. However, this is not the case with Java. In Java, the comma is a separator.

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

Beginning with JDK 5, a second form of **for** was defined that 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. Unlike some languages, such as C#, that implement a for-each loop by using the keyword **foreach**, Java adds the for-each capability by enhancing the **for** statement. The advantage of this approach is that no new keyword is required, and no preexisting code is broken. 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 18.

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 which 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 an example 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 an example 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 which 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.

CHAPTER

6 Introducing Classes

The class is at the core of Java. It is the logical construct upon which the entire Java language is built because it defines the shape and nature of an object. As such, the class forms the basis for object-oriented programming in Java. Any concept you wish to implement in a Java program must be encapsulated within a class.

Because the class is so fundamental to Java, this and the next few chapters will be devoted to it. Here, you will be introduced to the basic elements of a class and learn how a class can be used to create objects. You will also learn about methods, constructors, and the **this** keyword.

Class Fundamentals

Classes have been used since the beginning of this book. However, until now, only the most rudimentary form of a class has been shown. The classes created in the preceding chapters primarily exist simply to encapsulate the **main( )** method, which has been used to demonstrate the basics of the Java syntax. As you will see, classes are substantially more powerful than the limited ones presented so far.

Perhaps the most important thing to understand about a class is that it defines a new data type. Once defined, this new type can be used to create objects of that type. Thus, a class is a *template* for an object, and an object is an *instance* of a class. Because an object is an instance of a class, you will often see the two words *object* and *instance* used interchangeably.

The General Form of a Class

When you define a class, you declare its exact form and nature. You do this by specifying the data that it contains and the code that operates on that data. While very simple classes may contain only code or only data, most real-world classes contain both. As you will see, a class' code defines the interface to its data.

A class is declared by use of the **class** keyword. The classes that have been used up to this point are actually very limited examples of its complete form. Classes can (and usually do) get much more complex. A simplified general form of a **class** definition is shown here:

```
class classname {
    type instance-variable1;
```

```
    type instance-variable2;
    // ...
    type instance-variableN;

    type methodname1(parameter-list) {
      // body of method
    }
    type methodname2(parameter-list) {
      // body of method
    }
    // ...
    type methodnameN(parameter-list) {
      // body of method
    }
}
```

The data, or variables, defined within a **class** are called *instance variables*. The code is contained within *methods*. Collectively, the methods and variables defined within a class are called *members* of the class. In most classes, the instance variables are acted upon and accessed by the methods defined for that class. Thus, as a general rule, it is the methods that determine how a class' data can be used.

Variables defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables. Thus, the data for one object is separate and unique from the data for another. We will come back to this point shortly, but it is an important concept to learn early.

All methods have the same general form as **main( )**, which we have been using thus far. However, most methods will not be specified as **static** or **public**. Notice that the general form of a class does not specify a **main( )** method. Java classes do not need to have a **main( )** method. You only specify one if that class is the starting point for your program. Further, some kinds of Java applications, such as applets, don't require a **main( )** method at all.

A Simple Class

Let's begin our study of the class with a simple example. Here is a class called **Box** that defines three instance variables: **width**, **height**, and **depth**. Currently, **Box** does not contain any methods (but some will be added soon).

```
class Box {
  double width;
  double height;
  double depth;
}
```

As stated, a class defines a new type of data. In this case, the new data type is called **Box**. You will use this name to declare objects of type **Box**. It is important to remember that a class declaration only creates a template; it does not create an actual object. Thus, the preceding code does not cause any objects of type **Box** to come into existence.

To actually create a **Box** object, you will use a statement like the following:

```
Box mybox = new Box(); // create a Box object called mybox
```

After this statement executes, **mybox** will be an instance of **Box**. Thus, it will have "physical" reality. For the moment, don't worry about the details of this statement.

As mentioned earlier, each time you create an instance of a class, you are creating an object that contains its own copy of each instance variable defined by the class. Thus, every **Box** object will contain its own copies of the instance variables **width**, **height**, and **depth**. To access these variables, you will use the *dot* (.) operator. The dot operator links the name of the object with the name of an instance variable. For example, to assign the **width** variable of **mybox** the value 100, you would use the following statement:

```
mybox.width = 100;
```

This statement tells the compiler to assign the copy of **width** that is contained within the **mybox** object the value of 100. In general, you use the dot operator to access both the instance variables and the methods within an object. One other point: Although commonly referred to as the dot *operator*, the formal specification for Java categorizes the . as a separator. However, since the use of the term "dot operator" is widespread, it is used in this book.

Here is a complete program that uses the **Box** class:

```
/* A program that uses the Box class.

   Call this file BoxDemo.java
*/
class Box {
  double width;
  double height;
  double depth;
}

// This class declares an object of type Box.
class BoxDemo {
  public static void main(String args[]) {
    Box mybox = new Box();
    double vol;

    // assign values to mybox's instance variables
    mybox.width = 10;
    mybox.height = 20;
    mybox.depth = 15;

    // compute volume of box
    vol = mybox.width * mybox.height * mybox.depth;

    System.out.println("Volume is " + vol);
  }
}
```

You should call the file that contains this program **BoxDemo.java**, because the **main( )** method is in the class called **BoxDemo**, not the class called **Box**. When you compile this program, you will find that two **.class** files have been created, one for **Box** and one for **BoxDemo**. The Java compiler automatically puts each class into its own **.class** file. It is not necessary for both the **Box** and the **BoxDemo** class to actually be in the same source file. You could put each class in its own file, called **Box.java** and **BoxDemo.java**, respectively.

To run this program, you must execute **BoxDemo.class**. When you do, you will see the following output:

```
Volume is 3000.0
```

As stated earlier, each object has its own copies of the instance variables. This means that if you have two **Box** objects, each has its own copy of **depth**, **width**, and **height**. It is important to understand that changes to the instance variables of one object have no effect on the instance variables of another. For example, the following program declares two **Box** objects:

```
// This program declares two Box objects.

class Box {
  double width;
  double height;
  double depth;
}

class BoxDemo2 {
  public static void main(String args[]) {
    Box mybox1 = new Box();
    Box mybox2 = new Box();
    double vol;

    // assign values to mybox1's instance variables
    mybox1.width = 10;
    mybox1.height = 20;
    mybox1.depth = 15;

    /* assign different values to mybox2's
       instance variables */
    mybox2.width = 3;
    mybox2.height = 6;
    mybox2.depth = 9;

    // compute volume of first box
    vol = mybox1.width * mybox1.height * mybox1.depth;
    System.out.println("Volume is " + vol);

    // compute volume of second box
    vol = mybox2.width * mybox2.height * mybox2.depth;
    System.out.println("Volume is " + vol);
  }
}
```

The output produced by this program is shown here:

```
Volume is 3000.0
Volume is 162.0
```

As you can see, **mybox1**'s data is completely separate from the data contained in **mybox2**.

Declaring Objects

As just explained, when you create a class, you are creating a new data type. You can use this type to declare objects of that type. However, obtaining objects of a class is a two-step process. First, you must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can *refer* to an object. Second, you must acquire an actual, physical copy of the object and assign it to that variable. You can do this using the **new** operator. The **new** operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by **new**. This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated. Let's look at the details of this procedure.

In the preceding sample programs, a line similar to the following is used to declare an object of type **Box**:

```
Box mybox = new Box();
```

This statement combines the two steps just described. It can be rewritten like this to show each step more clearly:

```
Box mybox; // declare reference to object
mybox = new Box(); // allocate a Box object
```

The first line declares **mybox** as a reference to an object of type **Box**. At this point, **mybox** does not yet refer to an actual object. The next line allocates an object and assigns a reference to it to **mybox**. After the second line executes, you can use **mybox** as if it were a **Box** object. But in reality, **mybox** simply holds, in essence, the memory address of the actual **Box** object. The effect of these two lines of code is depicted in Figure 6-1.

NOTE Those readers familiar with C/C++ have probably noticed that object references appear to be similar to pointers. This suspicion is, essentially, correct. An object reference is similar to a memory pointer. The main difference—and the key to Java's safety—is that you cannot manipulate references as you can actual pointers. Thus, you cannot cause an object reference to point to an arbitrary memory location or manipulate it like an integer.

A Closer Look at new

As just explained, the **new** operator dynamically allocates memory for an object. It has this general form:

class-var = new *classname* ( );

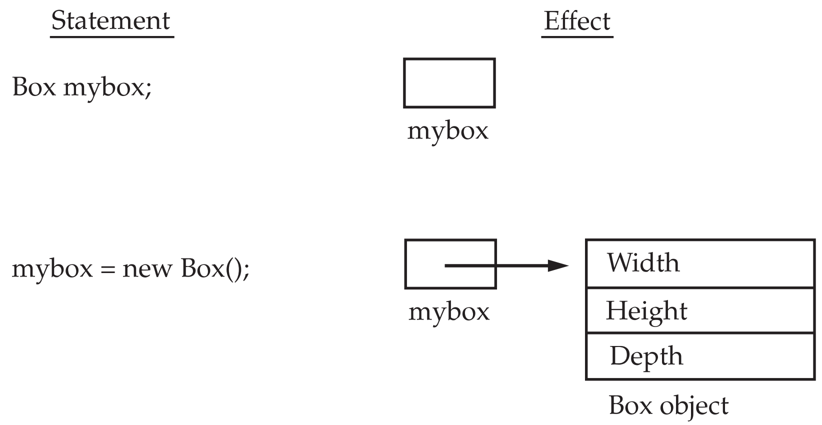


Figure 6-1 Declaring an object of type **Box**

Here, *class-var* is a variable of the class type being created. The *classname* is the name of the class that is being instantiated. The class name followed by parentheses specifies the *constructor* for the class. A constructor defines what occurs when an object of a class is created. Constructors are an important part of all classes and have many significant attributes. Most real-world classes explicitly define their own constructors within their class definition. However, if no explicit constructor is specified, then Java will automatically supply a default constructor. This is the case with **Box**. For now, we will use the default constructor. Soon, you will see how to define your own constructors.

At this point, you might be wondering why you do not need to use **new** for such things as integers or characters. The answer is that Java's primitive types are not implemented as objects. Rather, they are implemented as "normal" variables. This is done in the interest of efficiency. As you will see, objects have many features and attributes that require Java to treat them differently than it treats the primitive types. By not applying the same overhead to the primitive types that applies to objects, Java can implement the primitive types more efficiently. Later, you will see object versions of the primitive types that are available for your use in those situations in which complete objects of these types are needed.

It is important to understand that **new** allocates memory for an object during run time. The advantage of this approach is that your program can create as many or as few objects as it needs during the execution of your program. However, since memory is finite, it is possible that **new** will not be able to allocate memory for an object because insufficient memory exists. If this happens, a run-time exception will occur. (You will learn how to handle exceptions in Chapter 10.) For the sample programs in this book, you won't need to worry about running out of memory, but you will need to consider this possibility in real-world programs that you write.

Let's once again review the distinction between a class and an object. A class creates a new data type that can be used to create objects. That is, a class creates a logical framework that defines the relationship between its members. When you declare an object of a class, you are creating an instance of that class. Thus, a class is a logical construct. An object has physical reality. (That is, an object occupies space in memory.) It is important to keep this distinction clearly in mind.

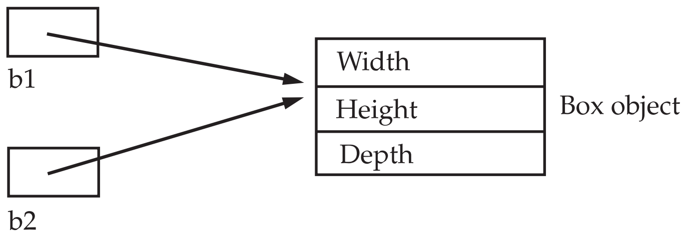
Assigning Object Reference Variables

Object reference variables act differently than you might expect when an assignment takes place. For example, what do you think the following fragment does?

```
Box b1 = new Box();
Box b2 = b1;
```

You might think that **b2** is being assigned a reference to a copy of the object referred to by **b1**. That is, you might think that **b1** and **b2** refer to separate and distinct objects. However, this would be wrong. Instead, after this fragment executes, **b1** and **b2** will both refer to the *same* object. The assignment of **b1** to **b2** did not allocate any memory or copy any part of the original object. It simply makes **b2** refer to the same object as does **b1**. Thus, any changes made to the object through **b2** will affect the object to which **b1** is referring, since they are the same object.

This situation is depicted here:



Although **b1** and **b2** both refer to the same object, they are not linked in any other way. For example, a subsequent assignment to **b1** will simply *unhook* **b1** from the original object without affecting the object or affecting **b2**. For example:

```
Box b1 = new Box();
Box b2 = b1;
// ...
b1 = null;
```

Here, **b1** has been set to **null**, but **b2** still points to the original object.

REMEMBER When you assign one object reference variable to another object reference variable, you are not creating a copy of the object, you are only making a copy of the reference.

Introducing Methods

As mentioned at the beginning of this chapter, classes usually consist of two things: instance variables and methods. The topic of methods is a large one because Java gives them so much power and flexibility. In fact, much of the next chapter is devoted to methods. However, there are some fundamentals that you need to learn now so that you can begin to add methods to your classes.

This is the general form of a method:

type name(*parameter-list*) {
    // body of method
}

Here, *type* specifies the type of data returned by the method. This can be any valid type, including class types that you create. If the method does not return a value, its return type must be **void**. The name of the method is specified by *name*. This can be any legal identifier other than those already used by other items within the current scope. The *parameter-list* is a sequence of type and identifier pairs separated by commas. Parameters are essentially variables that receive the value of the arguments passed to the method when it is called. If the method has no parameters, then the parameter list will be empty.

Methods that have a return type other than **void** return a value to the calling routine using the following form of the **return** statement:

return *value*;

Here, *value* is the value returned.

In the next few sections, you will see how to create various types of methods, including those that take parameters and those that return values.

Adding a Method to the Box Class

Although it is perfectly fine to create a class that contains only data, it rarely happens. Most of the time, you will use methods to access the instance variables defined by the class. In fact, methods define the interface to most classes. This allows the class implementor to hide the specific layout of internal data structures behind cleaner method abstractions. In addition to defining methods that provide access to data, you can also define methods that are used internally by the class itself.

Let's begin by adding a method to the **Box** class. It may have occurred to you while looking at the preceding programs that the computation of a box's volume was something that was best handled by the **Box** class rather than the **BoxDemo** class. After all, since the volume of a box is dependent upon the size of the box, it makes sense to have the **Box** class compute it. To do this, you must add a method to **Box**, as shown here:

```
// This program includes a method inside the box class.

class Box {
  double width;
  double height;
  double depth;

  // display volume of a box
  void volume() {
    System.out.print("Volume is ");
    System.out.println(width * height * depth);
  }
}

class BoxDemo3 {
  public static void main(String args[]) {
```

```
    Box mybox1 = new Box();
    Box mybox2 = new Box();

    // assign values to mybox1's instance variables
    mybox1.width = 10;
    mybox1.height = 20;
    mybox1.depth = 15;

    /* assign different values to mybox2's
       instance variables */
    mybox2.width = 3;
    mybox2.height = 6;
    mybox2.depth = 9;

    // display volume of first box
    mybox1.volume();

    // display volume of second box
    mybox2.volume();
  }
}
```

This program generates the following output, which is the same as the previous version.

```
Volume is 3000.0
Volume is 162.0
```

Look closely at the following two lines of code:

```
mybox1.volume();
mybox2.volume();
```

The first line here invokes the **volume( )** method on **mybox1**. That is, it calls **volume( )** relative to the **mybox1** object, using the object's name followed by the dot operator. Thus, the call to **mybox1.volume( )** displays the volume of the box defined by **mybox1**, and the call to **mybox2.volume( )** displays the volume of the box defined by **mybox2**. Each time **volume( )** is invoked, it displays the volume for the specified box.

If you are unfamiliar with the concept of calling a method, the following discussion will help clear things up. When **mybox1.volume( )** is executed, the Java run-time system transfers control to the code defined inside **volume( )**. After the statements inside **volume( )** have executed, control is returned to the calling routine, and execution resumes with the line of code following the call. In the most general sense, a method is Java's way of implementing subroutines.

There is something very important to notice inside the **volume( )** method: the instance variables **width**, **height**, and **depth** are referred to directly, without preceding them with an object name or the dot operator. When a method uses an instance variable that is defined by its class, it does so directly, without explicit reference to an object and without use of the dot operator. This is easy to understand if you think about it. A method is always invoked relative to some object of its class. Once this invocation has occurred, the object is known. Thus, within a method, there is no need to specify the object a second time. This means that **width**, **height**, and **depth** inside **volume( )** implicitly refer to the copies of those variables found in the object that invokes **volume( )**.

Let's review: When an instance variable is accessed by code that is not part of the class in which that instance variable is defined, it must be done through an object, by use of the dot operator. However, when an instance variable is accessed by code that is part of the same class as the instance variable, that variable can be referred to directly. The same thing applies to methods.

Returning a Value

While the implementation of **volume( )** does move the computation of a box's volume inside the **Box** class where it belongs, it is not the best way to do it. For example, what if another part of your program wanted to know the volume of a box, but not display its value? A better way to implement **volume( )** is to have it compute the volume of the box and return the result to the caller. The following example, an improved version of the preceding program, does just that:

```
// Now, volume() returns the volume of a box.

class Box {
  double width;
  double height;
  double depth;

  // compute and return volume
  double volume() {
    return width * height * depth;
  }
}

class BoxDemo4 {
  public static void main(String args[]) {
    Box mybox1 = new Box();
    Box mybox2 = new Box();
    double vol;

    // assign values to mybox1's instance variables
    mybox1.width = 10;
    mybox1.height = 20;
    mybox1.depth = 15;

    /* assign different values to mybox2's
       instance variables */
    mybox2.width = 3;
    mybox2.height = 6;
    mybox2.depth = 9;

    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume is " + vol);

    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume is " + vol);
  }
}
```

As you can see, when **volume( )** is called, it is put on the right side of an assignment statement. On the left is a variable, in this case **vol**, that will receive the value returned by **volume( )**. Thus, after

```
vol = mybox1.volume();
```

executes, the value of **mybox1.volume( )** is 3,000 and this value then is stored in **vol**.

There are two important things to understand about returning values:

- The type of data returned by a method must be compatible with the return type specified by the method. For example, if the return type of some method is **boolean**, you could not return an integer.
- The variable receiving the value returned by a method (such as **vol**, in this case) must also be compatible with the return type specified for the method.

One more point: The preceding program can be written a bit more efficiently because there is actually no need for the **vol** variable. The call to **volume( )** could have been used in the **println( )** statement directly, as shown here:

```
System.out.println("Volume is" + mybox1.volume());
```

In this case, when **println( )** is executed, **mybox1.volume( )** will be called automatically and its value will be passed to **println( )**.

Adding a Method That Takes Parameters

While some methods don't need parameters, most do. Parameters allow a method to be generalized. That is, a parameterized method can operate on a variety of data and/or be used in a number of slightly different situations. To illustrate this point, let's use a very simple example. Here is a method that returns the square of the number 10:

```
int square()
{
  return 10 * 10;
}
```

While this method does, indeed, return the value of 10 squared, its use is very limited. However, if you modify the method so that it takes a parameter, as shown next, then you can make **square( )** much more useful.

```
int square(int i)
{
  return i * i;
}
```

Now, **square( )** will return the square of whatever value it is called with. That is, **square( )** is now a general-purpose method that can compute the square of any integer value, rather than just 10.

Here is an example:

```
int x, y;
x = square(5); // x equals 25
x = square(9); // x equals 81
```

```
y = 2;
x = square(y); // x equals 4
```

In the first call to **square( )**, the value 5 will be passed into parameter **i**. In the second call, **i** will receive the value 9. The third invocation passes the value of **y**, which is 2 in this example. As these examples show, **square( )** is able to return the square of whatever data it is passed.

It is important to keep the two terms *parameter* and *argument* straight. A *parameter* is a variable defined by a method that receives a value when the method is called. For example, in **square( )**, **i** is a parameter. An *argument* is a value that is passed to a method when it is invoked. For example, **square(100)** passes 100 as an argument. Inside **square( )**, the parameter **i** receives that value.

You can use a parameterized method to improve the **Box** class. In the preceding examples, the dimensions of each box had to be set separately by use of a sequence of statements, such as:

```
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
```

While this code works, it is troubling for two reasons. First, it is clumsy and error prone. For example, it would be easy to forget to set a dimension. Second, in well-designed Java programs, instance variables should be accessed only through methods defined by their class. In the future, you can change the behavior of a method, but you can't change the behavior of an exposed instance variable.

Thus, a better approach to setting the dimensions of a box is to create a method that takes the dimensions of a box in its parameters and sets each instance variable appropriately. This concept is implemented by the following program:

```
// This program uses a parameterized method.

class Box {
  double width;
  double height;
  double depth;

  // compute and return volume
  double volume() {
    return width * height * depth;
  }

  // sets dimensions of box
  void setDim(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }
}

class BoxDemo5 {
```

```
  public static void main(String args[]) {
    Box mybox1 = new Box();
    Box mybox2 = new Box();
    double vol;

    // initialize each box
    mybox1.setDim(10, 20, 15);
    mybox2.setDim(3, 6, 9);

    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume is " + vol);

    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume is " + vol);
  }
}
```

As you can see, the **setDim( )** method is used to set the dimensions of each box. For example, when

```
mybox1.setDim(10, 20, 15);
```

is executed, 10 is copied into parameter **w**, 20 is copied into **h**, and 15 is copied into **d**. Inside **setDim( )** the values of **w**, **h**, and **d** are then assigned to **width**, **height**, and **depth**, respectively.

For many readers, the concepts presented in the preceding sections will be familiar. However, if such things as method calls, arguments, and parameters are new to you, then you might want to take some time to experiment before moving on. The concepts of the method invocation, parameters, and return values are fundamental to Java programming.

Constructors

It can be tedious to initialize all of the variables in a class each time an instance is created. Even when you add convenience functions like **setDim( )**, it would be simpler and more concise to have all of the setup done at the time the object is first created. Because the requirement for initialization is so common, Java allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor.

A *constructor* initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method. Once defined, the constructor is automatically called when the object is created, before the **new** operator completes. Constructors look a little strange because they have no return type, not even **void**. This is because the implicit return type of a class' constructor is the class type itself. It is the constructor's job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately.

You can rework the **Box** example so that the dimensions of a box are automatically initialized when an object is constructed. To do so, replace **setDim( )** with a constructor.

Let's begin by defining a simple constructor that simply sets the dimensions of each box to the same values. This version is shown here:

```
/* Here, Box uses a constructor to initialize the
   dimensions of a box.
*/
class Box {
  double width;
  double height;
  double depth;

  // This is the constructor for Box.
  Box() {
    System.out.println("Constructing Box");
    width = 10;
    height = 10;
    depth = 10;
  }

  // compute and return volume
  double volume() {
    return width * height * depth;
  }
}

class BoxDemo6 {
  public static void main(String args[]) {
    // declare, allocate, and initialize Box objects
    Box mybox1 = new Box();
    Box mybox2 = new Box();

    double vol;

    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume is " + vol);

    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume is " + vol);
  }
}
```

When this program is run, it generates the following results:

```
Constructing Box
Constructing Box
Volume is 1000.0
Volume is 1000.0
```

As you can see, both **mybox1** and **mybox2** were initialized by the **Box( )** constructor when they were created. Since the constructor gives all boxes the same dimensions, 10 by 10 by 10, both **mybox1** and **mybox2** will have the same volume. The **println( )** statement inside **Box( )** is for the sake of illustration only. Most constructors will not display anything. They will simply initialize an object.

Before moving on, let's reexamine the **new** operator. As you know, when you allocate an object, you use the following general form:

class-var = new *classname* ( );

Now you can understand why the parentheses are needed after the class name. What is actually happening is that the constructor for the class is being called. Thus, in the line

```
Box mybox1 = new Box();
```

new Box( ) is calling the **Box( )** constructor. When you do not explicitly define a constructor for a class, then Java creates a default constructor for the class. This is why the preceding line of code worked in earlier versions of **Box** that did not define a constructor. The default constructor automatically initializes all instance variables to their default values, which are zero, **null**, and **false**, for numeric types, reference types, and **boolean**, respectively. The default constructor is often sufficient for simple classes, but it usually won't do for more sophisticated ones. Once you define your own constructor, the default constructor is no longer used.

Parameterized Constructors

While the **Box( )** constructor in the preceding example does initialize a **Box** object, it is not very useful—all boxes have the same dimensions. What is needed is a way to construct **Box** objects of various dimensions. The easy solution is to add parameters to the constructor. As you can probably guess, this makes it much more useful. For example, the following version of **Box** defines a parameterized constructor that sets the dimensions of a box as specified by those parameters. Pay special attention to how **Box** objects are created.

```
/* Here, Box uses a parameterized constructor to
   initialize the dimensions of a box.
*/
class Box {
  double width;
  double height;
  double depth;

  // This is the constructor for Box.
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  // compute and return volume
  double volume() {
    return width * height * depth;
  }
}
```

```
class BoxDemo7 {
  public static void main(String args[]) {
    // declare, allocate, and initialize Box objects
    Box mybox1 = new Box(10, 20, 15);
    Box mybox2 = new Box(3, 6, 9);

    double vol;

    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume is " + vol);

    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume is " + vol);
  }
}
```

The output from this program is shown here:

```
Volume is 3000.0
Volume is 162.0
```

As you can see, each object is initialized as specified in the parameters to its constructor. For example, in the following line,

```
Box mybox1 = new Box(10, 20, 15);
```

the values 10, 20, and 15 are passed to the **Box( )** constructor when **new** creates the object. Thus, **mybox1**'s copy of **width**, **height**, and **depth** will contain the values 10, 20, and 15, respectively.

The this Keyword

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the **this** keyword. **this** can be used inside any method to refer to the *current* object. That is, **this** is always a reference to the object on which the method was invoked. You can use **this** anywhere a reference to an object of the current class' type is permitted.

To better understand what **this** refers to, consider the following version of **Box( )**:

```
// A redundant use of this.
Box(double w, double h, double d) {
  this.width = w;
  this.height = h;
  this.depth = d;
}
```

This version of **Box( )** operates exactly like the earlier version. The use of **this** is redundant, but perfectly correct. Inside **Box( )**, **this** will always refer to the invoking object. While it is redundant in this case, **this** is useful in other contexts, one of which is explained in the next section.

Instance Variable Hiding

As you know, it is illegal in Java to declare two local variables with the same name inside the same or enclosing scopes. Interestingly, you can have local variables, including formal parameters to methods, which overlap with the names of the class' instance variables. However, when a local variable has the same name as an instance variable, the local variable *hides* the instance variable. This is why **width**, **height**, and **depth** were not used as the names of the parameters to the **Box( )** constructor inside the **Box** class. If they had been, then **width,** for example, would have referred to the formal parameter, hiding the instance variable **width**. While it is usually easier to simply use different names, there is another way around this situation. Because **this** lets you refer directly to the object, you can use it to resolve any namespace collisions that might occur between instance variables and local variables. For example, here is another version of **Box( )**, which uses **width**, **height**, and **depth** for parameter names and then uses **this** to access the instance variables by the same name:

```
// Use this to resolve name-space collisions.
Box(double width, double height, double depth) {
  this.width = width;
  this.height = height;
  this.depth = depth;
}
```

A word of caution: The use of **this** in such a context can sometimes be confusing, and some programmers are careful not to use local variables and formal parameter names that hide instance variables. Of course, other programmers believe the contrary—that it is a good convention to use the same names for clarity, and use **this** to overcome the instance variable hiding. It is a matter of taste which approach you adopt.

Garbage Collection

Since objects are dynamically allocated by using the **new** operator, you might be wondering how such objects are destroyed and their memory released for later reallocation. In some languages, such as C++, dynamically allocated objects must be manually released by use of a **delete** operator. Java takes a different approach; it handles deallocation for you automatically. The technique that accomplishes this is called *garbage collection*. It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed. There is no explicit need to destroy objects as in C++. Garbage collection only occurs sporadically (if at all) during the execution of your program. It will not occur simply because one or more objects exist that are no longer used. Furthermore, different Java run-time implementations will take varying approaches to garbage collection, but for the most part, you should not have to think about it while writing your programs.

The finalize( ) Method

Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or character font, then you might want to make sure these resources are freed before an object is destroyed. To handle such situations, Java provides a mechanism called *finalization*. By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.

To add a finalizer to a class, you simply define the **finalize( )** method. The Java run time calls that method whenever it is about to recycle an object of that class. Inside the **finalize( )** method, you will specify those actions that must be performed before an object is destroyed. The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects. Right before an asset is freed, the Java run time calls the **finalize( )** method on the object.

The **finalize( )** method has this general form:

```
protected void finalize( )
{
// finalization code here
}
```

Here, the keyword **protected** is a specifier that limits access to **finalize( )**. This and the other access modifiers are explained in Chapter 7.

It is important to understand that **finalize( )** is only called just prior to garbage collection. It is not called when an object goes out-of-scope, for example. This means that you cannot know when—or even if—**finalize( )** will be executed. Therefore, your program should provide other means of releasing system resources, etc., used by the object. It must not rely on **finalize( )** for normal program operation.

NOTE If you are familiar with C++, then you know that C++ allows you to define a destructor for a class, which is called when an object goes out-of-scope. Java does not support this idea or provide for destructors. The **finalize( )** method only approximates the function of a destructor. As you get more experienced with Java, you will see that the need for destructor functions is minimal because of Java's garbage collection subsystem.

A Stack Class

While the **Box** class is useful to illustrate the essential elements of a class, it is of little practical value. To show the real power of classes, this chapter will conclude with a more sophisticated example. As you recall from the discussion of object-oriented programming (OOP) presented in Chapter 2, one of OOP's most important benefits is the encapsulation of data and the code that manipulates that data. As you have seen, the class is the mechanism by which encapsulation is achieved in Java. By creating a class, you are creating a new data type that defines both the nature of the data being manipulated and the routines used to manipulate it. Further, the methods define a consistent and controlled interface to the class' data. Thus, you can use the class through its methods without having to worry about the details of its implementation or how the data is actually managed within the class. In a sense, a class is like a "data engine." No knowledge of what goes on inside the engine is required to use the engine through its controls. In fact, since the details are hidden, its inner workings can be changed as needed. As long as your code uses the class through its methods, internal details can change without causing side effects outside the class.

To see a practical application of the preceding discussion, let's develop one of the archetypal examples of encapsulation: the stack. A *stack* stores data using first-in, last-out ordering. That is, a stack is like a stack of plates on a table—the first plate put down on the table is the last plate to be used. Stacks are controlled through two operations traditionally called *push* and *pop*. To put an item on top of the stack, you will use push. To take an item off the stack, you will use pop. As you will see, it is easy to encapsulate the entire stack mechanism.

Here is a class called **Stack** that implements a stack for up to ten integers:

```
// This class defines an integer stack that can hold 10 values
class Stack {
  int stck[] = new int[10];
  int tos;

  // Initialize top-of-stack
  Stack() {
    tos = -1;
  }

  // Push an item onto the stack
  void push(int item) {
    if(tos==9)
      System.out.println("Stack is full.");
    else
      stck[++tos] = item;
  }

  // Pop an item from the stack
  int pop() {
    if(tos < 0) {
       System.out.println("Stack underflow.");
       return 0;
    }
    else
      return stck[tos--];
  }
}
```

As you can see, the **Stack** class defines two data items and three methods. The stack of integers is held by the array **stck**. This array is indexed by the variable **tos**, which always contains the index of the top of the stack. The **Stack( )** constructor initializes **tos** to –1, which indicates an empty stack. The method **push( )** puts an item on the stack. To retrieve an item, call **pop( )**. Since access to the stack is through **push( )** and **pop( )**, the fact that the stack is held in an array is actually not relevant to using the stack. For example, the stack could be held in a more complicated data structure, such as a linked list, yet the interface defined by **push( )** and **pop( )** would remain the same.

The class **TestStack**, shown here, demonstrates the **Stack** class. It creates two integer stacks, pushes some values onto each, and then pops them off.

```
class TestStack {
  public static void main(String args[]) {
    Stack mystack1 = new Stack();
    Stack mystack2 = new Stack();

    // push some numbers onto the stack
    for(int i=0; i<10; i++) mystack1.push(i);
    for(int i=10; i<20; i++) mystack2.push(i);

    // pop those numbers off the stack
    System.out.println("Stack in mystack1:");
    for(int i=0; i<10; i++)
       System.out.println(mystack1.pop());

    System.out.println("Stack in mystack2:");
    for(int i=0; i<10; i++)
       System.out.println(mystack2.pop());
  }
}
```

This program generates the following output:

```
Stack in mystack1:
9
8
7
6
5
4
3
2
1
0
Stack in mystack2:
19
18
17
16
15
14
13
12
11
10
```

As you can see, the contents of each stack are separate.

One last point about the **Stack** class. As it is currently implemented, it is possible for the array that holds the stack, **stck**, to be altered by code outside of the **Stack** class. This leaves **Stack** open to misuse or mischief. In the next chapter, you will see how to remedy this situation.

CHAPTER

7 A Closer Look at Methods and Classes

This chapter continues the discussion of methods and classes begun in the preceding chapter. It examines several topics relating to methods, including overloading, parameter passing, and recursion. The chapter then returns to the class, discussing access control, the use of the keyword **static**, and one of Java's most important built-in classes: **String**.

Overloading Methods

In Java, it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be overloaded, and the process is referred to as *method overloading*. Method overloading is one of the ways that Java supports polymorphism. If you have never used a language that allows the overloading of methods, then the concept may seem strange at first. But as you will see, method overloading is one of Java's most exciting and useful features.

When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call. Thus, overloaded methods must differ in the type and/or number of their parameters. While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method. When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

Here is a simple example that illustrates method overloading:

```
// Demonstrate method overloading.
class OverloadDemo {
  void test() {
    System.out.println("No parameters");
  }

  // Overload test for one integer parameter.
  void test(int a) {
    System.out.println("a: " + a);
  }
```

```
  // Overload test for two integer parameters.
  void test(int a, int b) {
    System.out.println("a and b: " + a + " " + b);
  }

  // Overload test for a double parameter
  double test(double a) {
    System.out.println("double a: " + a);
    return a*a;
  }
}

class Overload {
  public static void main(String args[]) {
    OverloadDemo ob = new OverloadDemo();
    double result;

    // call all versions of test()
    ob.test();
    ob.test(10);
    ob.test(10, 20);
    result = ob.test(123.25);
    System.out.println("Result of ob.test(123.25): " + result);
  }
}
```

This program generates the following output:

```
No parameters
a: 10
a and b: 10 20
double a: 123.25
Result of ob.test(123.25): 15190.5625
```

As you can see, **test( )** is overloaded four times. The first version takes no parameters, the second takes one integer parameter, the third takes two integer parameters, and the fourth takes one **double** parameter. The fact that the fourth version of **test( )** also returns a value is of no consequence relative to overloading, since return types do not play a role in overload resolution.

When an overloaded method is called, Java looks for a match between the arguments used to call the method and the method's parameters. However, this match need not always be exact. In some cases, Java's automatic type conversions can play a role in overload resolution. For example, consider the following program:

```
// Automatic type conversions apply to overloading.
class OverloadDemo {
  void test() {
    System.out.println("No parameters");
  }

  // Overload test for two integer parameters.
  void test(int a, int b) {
    System.out.println("a and b: " + a + " " + b);
```

```
  }

  // Overload test for a double parameter
  void test(double a) {
    System.out.println("Inside test(double) a: " + a);
  }
}

class Overload {
  public static void main(String args[]) {
    OverloadDemo ob = new OverloadDemo();
    int i = 88;

    ob.test();
    ob.test(10, 20);

    ob.test(i); // this will invoke test(double)
    ob.test(123.2); // this will invoke test(double)
  }
}
```

This program generates the following output:

```
No parameters
a and b: 10 20
Inside test(double) a: 88
Inside test(double) a: 123.2
```

As you can see, this version of **OverloadDemo** does not define **test(int)**. Therefore, when **test( )** is called with an integer argument inside **Overload**, no matching method is found. However, Java can automatically convert an integer into a **double**, and this conversion can be used to resolve the call. Therefore, after **test(int)** is not found, Java elevates **i** to **double** and then calls **test(double)**. Of course, if **test(int)** had been defined, it would have been called instead. Java will employ its automatic type conversions only if no exact match is found.

Method overloading supports polymorphism because it is one way that Java implements the "one interface, multiple methods" paradigm. To understand how, consider the following. In languages that do not support method overloading, each method must be given a unique name. However, frequently you will want to implement essentially the same method for different types of data. Consider the absolute value function. In languages that do not support overloading, there are usually three or more versions of this function, each with a slightly different name. For instance, in C, the function **abs( )** returns the absolute value of an integer, **labs( )** returns the absolute value of a long integer, and **fabs( )** returns the absolute value of a floating-point value. Since C does not support overloading, each function has its own name, even though all three functions do essentially the same thing. This makes the situation more complex, conceptually, than it actually is. Although the underlying concept of each function is the same, you still have three names to remember. This situation does not occur in Java, because each absolute value method can use the same name. Indeed, Java's standard class library includes an absolute value method, called **abs( )**. This method is overloaded by Java's **Math** class to handle all numeric types. Java determines which version of **abs( )** to call based upon the type of argument.

The value of overloading is that it allows related methods to be accessed by use of a common name. Thus, the name **abs** represents the *general action* that is being performed. It is left to the compiler to choose the right *specific* version for a particular circumstance. You, the programmer, need only remember the general operation being performed. Through the application of polymorphism, several names have been reduced to one. Although this example is fairly simple, if you expand the concept, you can see how overloading can help you manage greater complexity.

When you overload a method, each version of that method can perform any activity you desire. There is no rule stating that overloaded methods must relate to one another. However, from a stylistic point of view, method overloading implies a relationship. Thus, while you can use the same name to overload unrelated methods, you should not. For example, you could use the name **sqr** to create methods that return the *square* of an integer and the *square root* of a floating-point value. But these two operations are fundamentally different. Applying method overloading in this manner defeats its original purpose. In practice, you should only overload closely related operations.

Overloading Constructors

In addition to overloading normal methods, you can also overload constructor methods. In fact, for most real-world classes that you create, overloaded constructors will be the norm, not the exception. To understand why, let's return to the **Box** class developed in the preceding chapter. Following is the latest version of **Box**:

```
class Box {
  double width;
  double height;
  double depth;

  // This is the constructor for Box.
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  // compute and return volume
  double volume() {
    return width * height * depth;
  }
}
```

As you can see, the **Box( )** constructor requires three parameters. This means that all declarations of **Box** objects must pass three arguments to the **Box( )** constructor. For example, the following statement is currently invalid:

```
Box ob = new Box();
```

Since **Box( )** requires three arguments, it's an error to call it without them. This raises some important questions. What if you simply wanted a box and did not care (or know) what its initial dimensions were? Or, what if you want to be able to initialize a cube by specifying only one value that would be used for all three dimensions? As the **Box** class is currently written, these other options are not available to you.

Fortunately, the solution to these problems is quite easy: simply overload the **Box** constructor so that it handles the situations just described. Here is a program that contains an improved version of **Box** that does just that:

```
/* Here, Box defines three constructors to initialize
   the dimensions of a box various ways.
*/
class Box {
  double width;
  double height;
  double depth;

  // constructor used when all dimensions specified
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  // constructor used when no dimensions specified
  Box() {
    width = -1;  // use -1 to indicate
    height = -1; // an uninitialized
    depth = -1;  // box
  }

  // constructor used when cube is created
  Box(double len) {
    width = height = depth = len;
  }

  // compute and return volume
  double volume() {
    return width * height * depth;
  }
}

class OverloadCons {
  public static void main(String args[]) {
    // create boxes using the various constructors
    Box mybox1 = new Box(10, 20, 15);
    Box mybox2 = new Box();
    Box mycube = new Box(7);

    double vol;

    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume of mybox1 is " + vol);

    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume of mybox2 is " + vol);
```

```
    // get volume of cube
    vol = mycube.volume();
    System.out.println("Volume of mycube is " + vol);
  }
}
```

The output produced by this program is shown here:

```
Volume of mybox1 is 3000.0
Volume of mybox2 is -1.0
Volume of mycube is 343.0
```

As you can see, the proper overloaded constructor is called based upon the parameters specified when **new** is executed.

Using Objects as Parameters

So far, we have only been using simple types as parameters to methods. However, it is both correct and common to pass objects to methods. For example, consider the following short program:

```
// Objects may be passed to methods.
class Test {
  int a, b;

  Test(int i, int j) {
    a = i;
    b = j;
  }

  // return true if o is equal to the invoking object
  boolean equalTo(Test o) {
    if(o.a == a && o.b == b) return true;
    else return false;
  }
}

class PassOb {
  public static void main(String args[]) {
    Test ob1 = new Test(100, 22);
    Test ob2 = new Test(100, 22);
    Test ob3 = new Test(-1, -1);

    System.out.println("ob1 == ob2: " + ob1.equalTo(ob2));
    System.out.println("ob1 == ob3: " + ob1.equalTo(ob3));
  }
}
```

This program generates the following output:

```
ob1 == ob2: true
ob1 == ob3: false
```

As you can see, the **equalTo( )** method inside **Test** compares two objects for equality and returns the result. That is, it compares the invoking object with the one that it is passed. If they contain the same values, then the method returns **true**. Otherwise, it returns **false**. Notice that the parameter **o** in **equalTo( )** specifies **Test** as its type. Although **Test** is a class type created by the program, it is used in just the same way as Java's built-in types.

One of the most common uses of object parameters involves constructors. Frequently, you will want to construct a new object so that it is initially the same as some existing object. To do this, you must define a constructor that takes an object of its class as a parameter. For example, the following version of **Box** allows one object to initialize another:

```
// Here, Box allows one object to initialize another.

class Box {
  double width;
  double height;
  double depth;

  // Notice this constructor. It takes an object of type Box.
  Box(Box ob) { // pass object to constructor
    width = ob.width;
    height = ob.height;
    depth = ob.depth;
  }

  // constructor used when all dimensions specified
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  // constructor used when no dimensions specified
  Box() {
    width = -1;  // use -1 to indicate
    height = -1; // an uninitialized
    depth = -1;  // box
  }

  // constructor used when cube is created
  Box(double len) {
    width = height = depth = len;
  }

  // compute and return volume
  double volume() {
    return width * height * depth;
  }
}

class OverloadCons2 {
  public static void main(String args[]) {
    // create boxes using the various constructors
```

```
    Box mybox1 = new Box(10, 20, 15);
    Box mybox2 = new Box();
    Box mycube = new Box(7);

    Box myclone = new Box(mybox1); // create copy of mybox1

    double vol;

    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume of mybox1 is " + vol);

    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume of mybox2 is " + vol);

    // get volume of cube
    vol = mycube.volume();
    System.out.println("Volume of cube is " + vol);

   // get volume of clone
    vol = myclone.volume();
    System.out.println("Volume of clone is " + vol);
  }
}
```

As you will see when you begin to create your own classes, providing many forms of constructors is usually required to allow objects to be constructed in a convenient and efficient manner.

A Closer Look at Argument Passing

In general, there are two ways that a computer language can pass an argument to a subroutine. The first way is *call-by-value*. This approach copies the *value* of an argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument. The second way an argument can be passed is *call-by-reference*. In this approach, a reference to an argument (not the value of the argument) is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter will affect the argument used to call the subroutine. As you will see, although Java uses call-by-value to pass all arguments, the precise effect differs between whether a primitive type or a reference type is passed.

When you pass a primitive type to a method, it is passed by value. Thus, a copy of the argument is made, and what occurs to the parameter that receives the argument has no effect outside the method. For example, consider the following program:

```
// Primitive types are passed by value.
class Test {
  void meth(int i, int j) {
    i *= 2;
    j /= 2;
  }
}
```

```
class CallByValue {
  public static void main(String args[]) {
    Test ob = new Test();

    int a = 15, b = 20;

    System.out.println("a and b before call: " +
                       a + " " + b);

    ob.meth(a, b);

    System.out.println("a and b after call: " +
                       a + " " + b);
  }
}
```

The output from this program is shown here:

```
a and b before call: 15 20
a and b after call: 15 20
```

As you can see, the operations that occur inside **meth( )** have no effect on the values of **a** and **b** used in the call; their values here did not change to 30 and 10.

When you pass an object to a method, the situation changes dramatically, because objects are passed by what is effectively call-by-reference. Keep in mind that when you create a variable of a class type, you are only creating a reference to an object. Thus, when you pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument. This effectively means that objects act as if they are passed to methods by use of call-by-reference. Changes to the object inside the method *do* affect the object used as an argument. For example, consider the following program:

```
// Objects are passed through their references.

class Test {
  int a, b;

  Test(int i, int j) {
    a = i;
    b = j;
  }

  // pass an object
  void meth(Test o) {
    o.a *= 2;
    o.b /= 2;
  }
}

class PassObjRef {
  public static void main(String args[]) {
    Test ob = new Test(15, 20);
```

```
    System.out.println("ob.a and ob.b before call: " +
                       ob.a + " " + ob.b);

    ob.meth(ob);

    System.out.println("ob.a and ob.b after call: " +
                       ob.a + " " + ob.b);
  }
}
```

This program generates the following output:

```
ob.a and ob.b before call: 15 20
ob.a and ob.b after call: 30 10
```

As you can see, in this case, the actions inside **meth( )** have affected the object used as an argument.

REMEMBER When an object reference is passed to a method, the reference itself is passed by use of call-by-value. However, since the value being passed refers to an object, the copy of that value will still refer to the same object that its corresponding argument does.

Returning Objects

A method can return any type of data, including class types that you create. For example, in the following program, the **incrByTen( )** method returns an object in which the value of **a** is ten greater than it is in the invoking object.

```
// Returning an object.
class Test {
  int a;

  Test(int i) {
    a = i;
  }

  Test incrByTen() {
    Test temp = new Test(a+10);
    return temp;
  }
}

class RetOb {
  public static void main(String args[]) {
    Test ob1 = new Test(2);
    Test ob2;

    ob2 = ob1.incrByTen();
    System.out.println("ob1.a: " + ob1.a);
    System.out.println("ob2.a: " + ob2.a);
```

```
    ob2 = ob2.incrByTen();
    System.out.println("ob2.a after second increase: "
                       + ob2.a);
  }
}
```

The output generated by this program is shown here:

```
ob1.a: 2
ob2.a: 12
ob2.a after second increase: 22
```

As you can see, each time **incrByTen( )** is invoked, a new object is created, and a reference to it is returned to the calling routine.

The preceding program makes another important point: Since all objects are dynamically allocated using **new**, you don't need to worry about an object going out-of-scope because the method in which it was created terminates. The object will continue to exist as long as there is a reference to it somewhere in your program. When there are no references to it, the object will be reclaimed the next time garbage collection takes place.

Recursion

Java supports *recursion*. Recursion is the process of defining something in terms of itself. As it relates to Java programming, recursion is the attribute that allows a method to call itself. A method that calls itself is said to be *recursive*.

The classic example of recursion is the computation of the factorial of a number. The factorial of a number *N* is the product of all the whole numbers between 1 and *N*. For example, 3 factorial is 1 × 2 × 3 ×, or 6. Here is how a factorial can be computed by use of a recursive method:

```
// A simple example of recursion.
class Factorial {
  // this is a recursive method
  int fact(int n) {
    int result;

    if(n==1) return 1;
    result = fact(n-1) * n;
    return result;
  }
}

class Recursion {
  public static void main(String args[]) {
    Factorial f = new Factorial();

    System.out.println("Factorial of 3 is " + f.fact(3));
    System.out.println("Factorial of 4 is " + f.fact(4));
    System.out.println("Factorial of 5 is " + f.fact(5));
  }
}
```

The output from this program is shown here:

```
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120
```

If you are unfamiliar with recursive methods, then the operation of **fact( )** may seem a bit confusing. Here is how it works. When **fact( )** is called with an argument of 1, the function returns 1; otherwise, it returns the product of **fact(n–1)*n**. To evaluate this expression, **fact( )** is called with **n–1**. This process repeats until **n** equals 1 and the calls to the method begin returning.

To better understand how the **fact( )** method works, let's go through a short example. When you compute the factorial of 3, the first call to **fact( )** will cause a second call to be made with an argument of 2. This invocation will cause **fact( )** to be called a third time with an argument of 1. This call will return 1, which is then multiplied by 2 (the value of **n** in the second invocation). This result (which is 2) is then returned to the original invocation of **fact( )** and multiplied by 3 (the original value of **n**). This yields the answer, 6. You might find it interesting to insert **println( )** statements into **fact( )**, which will show at what level each call is and what the intermediate answers are.

When a method calls itself, new local variables and parameters are allocated storage on the stack, and the method code is executed with these new variables from the start. As each recursive call returns, the old local variables and parameters are removed from the stack, and execution resumes at the point of the call inside the method. Recursive methods could be said to "telescope" out and back.

Recursive versions of many routines may execute a bit more slowly than the iterative equivalent because of the added overhead of the additional method calls. Many recursive calls to a method could cause a stack overrun. Because storage for parameters and local variables is on the stack and each new call creates a new copy of these variables, it is possible that the stack could be exhausted. If this occurs, the Java run-time system will cause an exception. However, you probably will not have to worry about this unless a recursive routine runs wild.

The main advantage to recursive methods is that they can be used to create clearer and simpler versions of several algorithms than can their iterative relatives. For example, the QuickSort sorting algorithm is quite difficult to implement in an iterative way. Also, some types of AI-related algorithms are most easily implemented using recursive solutions.

When writing recursive methods, you must have an **if** statement somewhere to force the method to return without the recursive call being executed. If you don't do this, once you call the method, it will never return. This is a very common error in working with recursion. Use **println( )** statements liberally during development so that you can watch what is going on and abort execution if you see that you have made a mistake.

Here is one more example of recursion. The recursive method **printArray( )** prints the first **i** elements in the array **values**.

```
// Another example that uses recursion.

class RecTest {
  int values[];
```

```
  RecTest(int i) {
    values = new int[i];
  }

  // display array -- recursively
  void printArray(int i) {
    if(i==0) return;
    else printArray(i-1);
    System.out.println("[" + (i-1) + "] " + values[i-1]);
  }
}

class Recursion2 {
  public static void main(String args[]) {
    RecTest ob = new RecTest(10);
    int i;

    for(i=0; i<10; i++) ob.values[i] = i;

    ob.printArray(10);
  }
}
```

This program generates the following output:

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

Introducing Access Control

As you know, encapsulation links data with the code that manipulates it. However, encapsulation provides another important attribute: *access control*. Through encapsulation, you can control what parts of a program can access the members of a class. By controlling access, you can prevent misuse. For example, allowing access to data only through a well-defined set of methods, you can prevent the misuse of that data. Thus, when correctly implemented, a class creates a "black box" which may be used, but the inner workings of which are not open to tampering. However, the classes that were presented earlier do not completely meet this goal. For example, consider the **Stack** class shown at the end of Chapter 6. While it is true that the methods **push( )** and **pop( )** do provide a controlled interface to the stack, this interface is not enforced. That is, it is possible for another part of the program to bypass these methods and access the stack directly. Of course, in the wrong hands, this could lead to trouble. In this section, you will be introduced to the mechanism by which you can precisely control access to the various members of a class.

How a member can be accessed is determined by the *access modifier* attached to its declaration. Java supplies a rich set of access modifiers. Some aspects of access control are related mostly to inheritance or packages. (A *package* is, essentially, a grouping of classes.) These parts of Java's access control mechanism will be discussed later. Here, let's begin by examining access control as it applies to a single class. Once you understand the fundamentals of access control, the rest will be easy.

Java's access modifiers are **public**, **private**, and **protected**. Java also defines a default access level. **protected** applies only when inheritance is involved. The other access modifiers are described next.

Let's begin by defining **public** and **private**. When a member of a class is modified by **public**, then that member can be accessed by any other code. When a member of a class is specified as **private**, then that member can only be accessed by other members of its class. Now you can understand why **main( )** has always been preceded by the **public** modifier. It is called by code that is outside the program—that is, by the Java run-time system. When no access modifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package. (Packages are discussed in the following chapter.)

In the classes developed so far, all members of a class have used the default access mode. However, this is not what you will typically want to be the case. Usually, you will want to restrict access to the data members of a class—allowing access only through methods. Also, there will be times when you will want to define methods that are private to a class.

An access modifier precedes the rest of a member's type specification. That is, it must begin a member's declaration statement. Here is an example:

```
public int i;
private double j;

private int myMethod(int a, char b) { //...
```

To understand the effects of public and private access, consider the following program:

```
/* This program demonstrates the difference between
   public and private.
*/
class Test {
  int a; // default access
  public int b; // public access
  private int c; // private access

  // methods to access c
  void setc(int i) { // set c's value
    c = i;
  }
  int getc() { // get c's value
    return c;
  }
}
```

```
class AccessTest {
  public static void main(String args[]) {
    Test ob = new Test();

    // These are OK, a and b may be accessed directly
    ob.a = 10;
    ob.b = 20;

    // This is not OK and will cause an error
//  ob.c = 100; // Error!

    // You must access c through its methods
    ob.setc(100); // OK
    System.out.println("a, b, and c: " + ob.a + " " +
                       ob.b + " " + ob.getc());
  }
}
```

As you can see, inside the **Test** class, **a** uses default access, which for this example is the same as specifying **public**. **b** is explicitly specified as **public**. Member **c** is given private access. This means that it cannot be accessed by code outside of its class. So, inside the **AccessTest** class, **c** cannot be used directly. It must be accessed through its public methods: **setc( )** and **getc( )**. If you were to remove the comment symbol from the beginning of the following line,

```
// ob.c = 100; // Error!
```

then you would not be able to compile this program because of the access violation.

To see how access control can be applied to a more practical example, consider the following improved version of the **Stack** class shown at the end of Chapter 6.

```
// This class defines an integer stack that can hold 10 values.
class Stack {
  /* Now, both stck and tos are private. This means
     that they cannot be accidentally or maliciously
     altered in a way that would be harmful to the stack.
  */
  private int stck[] = new int[10];
  private int tos;

  // Initialize top-of-stack
  Stack() {
    tos = -1;
  }

  // Push an item onto the stack
  void push(int item) {
    if(tos==9)
      System.out.println("Stack is full.");
    else
      stck[++tos] = item;
  }
```

```
  // Pop an item from the stack
  int pop() {
    if(tos < 0) {
      System.out.println("Stack underflow.");
      return 0;
    }
    else
      return stck[tos--];
  }
}
```

As you can see, now both **stck**, which holds the stack, and **tos**, which is the index of the top of the stack, are specified as **private**. This means that they cannot be accessed or altered except through **push( )** and **pop( )**. Making **tos** private, for example, prevents other parts of your program from inadvertently setting it to a value that is beyond the end of the **stck** array.

The following program demonstrates the improved **Stack** class. Try removing the commented-out lines to prove to yourself that the **stck** and **tos** members are, indeed, inaccessible.

```
class TestStack {
  public static void main(String args[]) {
    Stack mystack1 = new Stack();
    Stack mystack2 = new Stack();

    // push some numbers onto the stack
    for(int i=0; i<10; i++) mystack1.push(i);
    for(int i=10; i<20; i++) mystack2.push(i);

    // pop those numbers off the stack
    System.out.println("Stack in mystack1:");
    for(int i=0; i<10; i++)
       System.out.println(mystack1.pop());

    System.out.println("Stack in mystack2:");

    for(int i=0; i<10; i++)
       System.out.println(mystack2.pop());

    // these statements are not legal
    // mystack1.tos = -2;
    // mystack2.stck[3] = 100;
  }
}
```

Although methods will usually provide access to the data defined by a class, this does not always have to be the case. It is perfectly proper to allow an instance variable to be public when there is good reason to do so. For example, most of the simple classes in this book were created with little concern about controlling access to instance variables for the sake of simplicity. However, in most real-world classes, you will need to allow operations on data only through methods. The next chapter will return to the topic of access control. As you will see, it is particularly important when inheritance is involved.

Understanding static

There will be times when you will want to define a class member that will be used independently of any object of that class. Normally, a class member must be accessed only in conjunction with an object of its class. However, it is possible to create a member that can be used by itself, without reference to a specific instance. To create such a member, precede its declaration with the keyword **static**. When a member is declared **static**, it can be accessed before any objects of its class are created, and without reference to any object. You can declare both methods and variables to be **static**. The most common example of a **static** member is **main( )**. **main( )** is declared as **static** because it must be called before any objects exist.

Instance variables declared as **static** are, essentially, global variables. When objects of its class are declared, no copy of a **static** variable is made. Instead, all instances of the class share the same **static** variable.

Methods declared as **static** have several restrictions:

- They can only directly call other **static** methods.
- They can only directly access **static** data.
- They cannot refer to **this** or **super** in any way. (The keyword **super** relates to inheritance and is described in the next chapter.)

If you need to do computation in order to initialize your **static** variables, you can declare a **static** block that gets executed exactly once, when the class is first loaded. The following example shows a class that has a **static** method, some **static** variables, and a **static** initialization block:

```
// Demonstrate static variables, methods, and blocks.
class UseStatic {
  static int a = 3;
  static int b;

  static void meth(int x) {
    System.out.println("x = " + x);
    System.out.println("a = " + a);
    System.out.println("b = " + b);
  }

  static {
    System.out.println("Static block initialized.");
    b = a * 4;
  }

  public static void main(String args[]) {
    meth(42);
  }
}
```

As soon as the **UseStatic** class is loaded, all of the **static** statements are run. First, **a** is set to **3**, then the **static** block executes, which prints a message and then initializes **b** to **a*4** or **12**. Then **main( )** is called, which calls **meth( )**, passing **42** to **x**. The three **println( )** statements refer to the two **static** variables **a** and **b**, as well as to the local variable **x**.

Here is the output of the program:

```
Static block initialized.
x = 42
a = 3
b = 12
```

Outside of the class in which they are defined, **static** methods and variables can be used independently of any object. To do so, you need only specify the name of their class followed by the dot operator. For example, if you wish to call a **static** method from outside its class, you can do so using the following general form:

classname.method( )

Here, *classname* is the name of the class in which the **static** method is declared. As you can see, this format is similar to that used to call non-**static** methods through object-reference variables. A **static** variable can be accessed in the same way—by use of the dot operator on the name of the class. This is how Java implements a controlled version of global methods and global variables.

Here is an example. Inside **main( )**, the **static** method **callme( )** and the **static** variable **b** are accessed through their class name **StaticDemo**.

```
class StaticDemo {
  static int a = 42;
  static int b = 99;

  static void callme() {
    System.out.println("a = " + a);
  }
}

class StaticByName {
  public static void main(String args[]) {
    StaticDemo.callme();
    System.out.println("b = " + StaticDemo.b);
  }
}
```

Here is the output of this program:

```
a = 42
b = 99
```

Introducing final

A field can be declared as **final**. Doing so prevents its contents from being modified, making it, essentially, a constant. This means that you must initialize a **final** field when it is declared. You can do this in one of two ways: First, you can give it a value when it is declared. Second, you can assign it a value within a constructor. The first approach is the most common. Here is an example:

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
final int FILE_SAVE = 3;
final int FILE_SAVEAS = 4;
final int FILE_QUIT = 5;
```

Subsequent parts of your program can now use **FILE_OPEN**, etc., as if they were constants, without fear that a value has been changed. It is a common coding convention to choose all uppercase identifiers for **final** fields, as this example shows.

In addition to fields, both method parameters and local variables can be declared **final**. Declaring a parameter **final** prevents it from being changed within the method. Declaring a local variable **final** prevents it from being assigned a value more than once.

The keyword **final** can also be applied to methods, but its meaning is substantially different than when it is applied to variables. This additional usage of **final** is described in the next chapter, when inheritance is described.

Arrays Revisited

Arrays were introduced earlier in this book, before classes had been discussed. Now that you know about classes, an important point can be made about arrays: they are implemented as objects. Because of this, there is a special array attribute that you will want to take advantage of. Specifically, the size of an array—that is, the number of elements that an array can hold—is found in its **length** instance variable. All arrays have this variable, and it will always hold the size of the array. Here is a program that demonstrates this property:

```
// This program demonstrates the length array member.
class Length {
  public static void main(String args[]) {
    int a1[] = new int[10];
    int a2[] = {3, 5, 7, 1, 8, 99, 44, -10};
    int a3[] = {4, 3, 2, 1};

    System.out.println("length of a1 is " + a1.length);
    System.out.println("length of a2 is " + a2.length);
    System.out.println("length of a3 is " + a3.length);
  }
}
```

This program displays the following output:

```
length of a1 is 10
length of a2 is 8
length of a3 is 4
```

As you can see, the size of each array is displayed. Keep in mind that the value of **length** has nothing to do with the number of elements that are actually in use. It only reflects the number of elements that the array is designed to hold.

You can put the **length** member to good use in many situations. For example, here is an improved version of the **Stack** class. As you might recall, the earlier versions of this class always created a ten-element stack. The following version lets you create stacks of any size. The value of **stck.length** is used to prevent the stack from overflowing.

```
// Improved Stack class that uses the length array member.
class Stack {
  private int stck[];
  private int tos;

  // allocate and initialize stack
  Stack(int size) {
    stck = new int[size];
    tos = -1;
  }

  // Push an item onto the stack
  void push(int item) {
    if(tos==stck.length-1) // use length member
      System.out.println("Stack is full.");
    else
      stck[++tos] = item;
  }

  // Pop an item from the stack
  int pop() {
    if(tos < 0) {
      System.out.println("Stack underflow.");
      return 0;
    }
    else
      return stck[tos--];
  }
}

class TestStack2 {
  public static void main(String args[]) {
    Stack mystack1 = new Stack(5);
    Stack mystack2 = new Stack(8);

    // push some numbers onto the stack
    for(int i=0; i<5; i++) mystack1.push(i);
    for(int i=0; i<8; i++) mystack2.push(i);

    // pop those numbers off the stack
    System.out.println("Stack in mystack1:");
    for(int i=0; i<5; i++)
      System.out.println(mystack1.pop());

    System.out.println("Stack in mystack2:");
    for(int i=0; i<8; i++)
      System.out.println(mystack2.pop());
  }
}
```

Notice that the program creates two stacks: one five elements deep and the other eight elements deep. As you can see, the fact that arrays maintain their own length information makes it easy to create stacks of any size.

Introducing Nested and Inner Classes

It is possible to define a class within another class; such classes are known as *nested classes*. The scope of a nested class is bounded by the scope of its enclosing class. Thus, if class B is defined within class A, then B does not exist independently of A. A nested class has access to the members, including private members, of the class in which it is nested. However, the enclosing class does not have access to the members of the nested class. A nested class that is declared directly within its enclosing class scope is a member of its enclosing class. It is also possible to declare a nested class that is local to a block.

There are two types of nested classes: *static* and *non-static*. A static nested class is one that has the **static** modifier applied. Because it is static, it must access the non-static members of its enclosing class through an object. That is, it cannot refer to non-static members of its enclosing class directly. Because of this restriction, static nested classes are seldom used.

The most important type of nested class is the *inner* class. An inner class is a non-static nested class. It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do.

The following program illustrates how to define and use an inner class. The class named **Outer** has one instance variable named **outer_x**, one instance method named **test( )**, and defines one inner class called **Inner**.

```
// Demonstrate an inner class.
class Outer {
  int outer_x = 100;

  void test() {
    Inner inner = new Inner();
    inner.display();
  }

  // this is an inner class
  class Inner {
    void display() {
      System.out.println("display: outer_x = " + outer_x);
    }
  }
}

class InnerClassDemo {
  public static void main(String args[]) {
    Outer outer = new Outer();
    outer.test();
  }
}
```

Output from this application is shown here:

```
display: outer_x = 100
```

In the program, an inner class named **Inner** is defined within the scope of class **Outer**. Therefore, any code in class **Inner** can directly access the variable **outer_x**. An instance method named **display( )** is defined inside **Inner**. This method displays **outer_x** on the standard output stream. The **main( )** method of **InnerClassDemo** creates an instance of class **Outer** and invokes its **test( )** method. That method creates an instance of class **Inner** and the **display( )** method is called.

It is important to realize that an instance of **Inner** can be created only in the context of class **Outer**. The Java compiler generates an error message otherwise. In general, an inner class instance is often created by code within its enclosing scope, as the example does.

As explained, an inner class has access to all of the members of its enclosing class, but the reverse is not true. Members of the inner class are known only within the scope of the inner class and may not be used by the outer class. For example,

```
// This program will not compile.
class Outer {
  int outer_x = 100;

  void test() {
    Inner inner = new Inner();
    inner.display();
  }

  // this is an inner class
  class Inner {
    int y = 10; // y is local to Inner

    void display() {
      System.out.println("display: outer_x = " + outer_x);
    }
  }

  void showy() {
    System.out.println(y); // error, y not known here!
  }
}

class InnerClassDemo {
  public static void main(String args[]) {
    Outer outer = new Outer();
    outer.test();
  }
}
```

Here, **y** is declared as an instance variable of **Inner**. Thus, it is not known outside of that class and it cannot be used by **showy( )**.

Although we have been focusing on inner classes declared as members within an outer class scope, it is possible to define inner classes within any block scope. For example, you can define a nested class within the block defined by a method or even within the body of a **for** loop, as this next program shows:

```
// Define an inner class within a for loop.
class Outer {
  int outer_x = 100;

  void test() {
    for(int i=0; i<10; i++) {
      class Inner {
        void display() {
          System.out.println("display: outer_x = " + outer_x);
        }
      }
      Inner inner = new Inner();
      inner.display();
    }
  }
}

class InnerClassDemo {
  public static void main(String args[]) {
    Outer outer = new Outer();
    outer.test();
  }
}
```

The output from this version of the program is shown here:

```
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
```

While nested classes are not applicable to all situations, they are particularly helpful when handling events. We will return to the topic of nested classes in Chapter 24. There you will see how inner classes can be used to simplify the code needed to handle certain types of events. You will also learn about *anonymous inner classes*, which are inner classes that don't have a name.

One final point: Nested classes were not allowed by the original 1.0 specification for Java. They were added by Java 1.1.

Exploring the String Class

Although the **String** class will be examined in depth in Part II of this book, a short exploration of it is warranted now, because we will be using strings in some of the example programs shown toward the end of Part I. **String** is probably the most commonly used class in Java's class library. The obvious reason for this is that strings are a very important part of programming.

The first thing to understand about strings is that every string you create is actually an object of type **String**. Even string constants are actually **String** objects. For example, in the statement

```
System.out.println("This is a String, too");
```

the string "This is a String, too" is a **String** object.

The second thing to understand about strings is that objects of type **String** are immutable; once a **String** object is created, its contents cannot be altered. While this may seem like a serious restriction, it is not, for two reasons:

- If you need to change a string, you can always create a new one that contains the modifications.
- Java defines peer classes of **String**, called **StringBuffer** and **StringBuilder**, which allow strings to be altered, so all of the normal string manipulations are still available in Java. (**StringBuffer** and **StringBuilder** are described in Part II of this book.)

Strings can be constructed in a variety of ways. The easiest is to use a statement like this:

```
String myString = "this is a test";
```

Once you have created a **String** object, you can use it anywhere that a string is allowed. For example, this statement displays **myString**:

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

Java defines one operator for **String** objects: **+**. It is used to concatenate two strings. For example, this statement

```
String myString = "I" + " like " + "Java.";
```

results in **myString** containing "I like Java."

The following program demonstrates the preceding concepts:

```
// Demonstrating Strings.
class StringDemo {
  public static void main(String args[]) {
    String strOb1 = "First String";
    String strOb2 = "Second String";
    String strOb3 = strOb1 + " and " + strOb2;

    System.out.println(strOb1);
```

```
    System.out.println(strOb2);
    System.out.println(strOb3);
  }
}
```

The output produced by this program is shown here:

```
First String
Second String
First String and Second String
```

The **String** class contains several methods that you can use. Here are a few. You can test two strings for equality by using **equals( )**. You can obtain the length of a string by calling the **length( )** method. You can obtain the character at a specified index within a string by calling **charAt( )**. The general forms of these three methods are shown here:

boolean equals(*secondStr*)
int length( )
char charAt(*index*)

Here is a program that demonstrates these methods:

```
// Demonstrating some String methods.
class StringDemo2 {
  public static void main(String args[]) {
    String strOb1 = "First String";
    String strOb2 = "Second String";
    String strOb3 = strOb1;

    System.out.println("Length of strOb1: " +
                       strOb1.length());

    System.out.println("Char at index 3 in strOb1: " +
                       strOb1.charAt(3));

    if(strOb1.equals(strOb2))
      System.out.println("strOb1 == strOb2");
    else
      System.out.println("strOb1 != strOb2");

    if(strOb1.equals(strOb3))
      System.out.println("strOb1 == strOb3");
    else
      System.out.println("strOb1 != strOb3");
  }
}
```

This program generates the following output:

```
Length of strOb1: 12
Char at index 3 in strOb1: s
strOb1 != strOb2
strOb1 == strOb3
```

Of course, you can have arrays of strings, just like you can have arrays of any other type of object. For example:

```
// Demonstrate String arrays.
class StringDemo3 {
  public static void main(String args[]) {
    String str[] = { "one", "two", "three" };

    for(int i=0; i<str.length; i++)
      System.out.println("str[" + i + "]: " +
                          str[i]);
  }
}
```

Here is the output from this program:

```
str[0]: one
str[1]: two
str[2]: three
```

As you will see in the following section, string arrays play an important part in many Java programs.

Using Command-Line Arguments

Sometimes you will want to pass information into a program when you run it. This is accomplished by passing *command-line arguments* to **main( )**. A command-line argument is the information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy—they are stored as strings in a **String** array passed to the **args** parameter of **main( )**. The first command-line argument is stored at **args[0]**, the second at **args[1]**, and so on. For example, the following program displays all of the command-line arguments that it is called with:

```
// Display all command-line arguments.
class CommandLine {
  public static void main(String args[]) {
    for(int i=0; i<args.length; i++)
      System.out.println("args[" + i + "]: " +
                          args[i]);
  }
}
```

Try executing this program, as shown here:

```
java CommandLine this is a test 100 -1
```

When you do, you will see the following output:

```
args[0]: this
args[1]: is
args[2]: a
args[3]: test
args[4]: 100
args[5]: -1
```

REMEMBER All command-line arguments are passed as strings. You must convert numeric values to their internal forms manually, as explained in Chapter 17.

Varargs: Variable-Length Arguments

Beginning with JDK 5, Java has included a feature that simplifies the creation of methods that need to take a variable number of arguments. This feature is called *varargs* and it is short for *variable-length arguments*. A method that takes a variable number of arguments is called a *variable-arity method*, or simply a *varargs method*.

Situations that require that a variable number of arguments be passed to a method are not unusual. For example, a method that opens an Internet connection might take a user name, password, filename, protocol, and so on, but supply defaults if some of this information is not provided. In this situation, it would be convenient to pass only the arguments to which the defaults did not apply. Another example is the **printf( )** method that is part of Java's I/O library. As you will see in Chapter 20, it takes a variable number of arguments, which it formats and then outputs.

Prior to JDK 5, variable-length arguments could be handled two ways, neither of which was particularly pleasing. First, if the maximum number of arguments was small and known, then you could create overloaded versions of the method, one for each way the method could be called. Although this works and is suitable for some cases, it applies to only a narrow class of situations.

In cases where the maximum number of potential arguments was larger, or unknowable, a second approach was used in which the arguments were put into an array, and then the array was passed to the method. This approach is illustrated by the following program:

```
// Use an array to pass a variable number of
// arguments to a method. This is the old-style
// approach to variable-length arguments.
class PassArray {
  static void vaTest(int v[]) {
    System.out.print("Number of args: " + v.length +
                     " Contents: ");

    for(int x : v)
      System.out.print(x + " ");
    System.out.println();
  }

  public static void main(String args[])
  {
    // Notice how an array must be created to
    // hold the arguments.
    int n1[] = { 10 };
    int n2[] = { 1, 2, 3 };
    int n3[] = { };

    vaTest(n1); // 1 arg
    vaTest(n2); // 3 args
    vaTest(n3); // no args
  }
}
```

The output from the program is shown here:

```
Number of args: 1 Contents: 10
Number of args: 3 Contents: 1 2 3
Number of args: 0 Contents:
```

In the program, the method **vaTest( )** is passed its arguments through the array **v**. This old-style approach to variable-length arguments does enable **vaTest( )** to take an arbitrary number of arguments. However, it requires that these arguments be manually packaged into an array prior to calling **vaTest( )**. Not only is it tedious to construct an array each time **vaTest( )** is called, it is potentially error-prone. The varargs feature offers a simpler, better option.

A variable-length argument is specified by three periods (**...**). For example, here is how **vaTest( )** is written using a vararg:

```
static void vaTest(int ... v) {
```

This syntax tells the compiler that **vaTest( )** can be called with zero or more arguments. As a result, **v** is implicitly declared as an array of type **int[ ]**. Thus, inside **vaTest( )**, **v** is accessed using the normal array syntax. Here is the preceding program rewritten using a vararg:

```
// Demonstrate variable-length arguments.
class VarArgs {

  // vaTest() now uses a vararg.
  static void vaTest(int ... v) {
    System.out.print("Number of args: " + v.length +
                     " Contents: ");

    for(int x : v)
      System.out.print(x + " ");

    System.out.println();
  }

  public static void main(String args[])
  {
    // Notice how vaTest() can be called with a
    // variable number of arguments.
    vaTest(10);      // 1 arg
    vaTest(1, 2, 3); // 3 args
    vaTest();        // no args
  }
}
```

The output from the program is the same as the original version.

There are two important things to notice about this program. First, as explained, inside **vaTest( )**, **v** is operated on as an array. This is because **v** *is* an array. The ... syntax simply tells the compiler that a variable number of arguments will be used, and that these arguments will be stored in the array referred to by **v**. Second, in **main( )**, **vaTest( )** is called with different numbers of arguments, including no arguments at all. The arguments are automatically put in an array and passed to **v**. In the case of no arguments, the length of the array is zero.

A method can have "normal" parameters along with a variable-length parameter. However, the variable-length parameter must be the last parameter declared by the method. For example, this method declaration is perfectly acceptable:

```
int doIt(int a, int b, double c, int ... vals) {
```

In this case, the first three arguments used in a call to **doIt( )** are matched to the first three parameters. Then, any remaining arguments are assumed to belong to **vals**.

Remember, the varargs parameter must be last. For example, the following declaration is incorrect:

```
int doIt(int a, int b, double c, int ... vals, boolean stopFlag) { // Error!
```

Here, there is an attempt to declare a regular parameter after the varargs parameter, which is illegal.

There is one more restriction to be aware of: there must be only one varargs parameter. For example, this declaration is also invalid:

```
int doIt(int a, int b, double c, int ... vals, double ... morevals) { // Error!
```

The attempt to declare the second varargs parameter is illegal.

Here is a reworked version of the **vaTest( )** method that takes a regular argument and a variable-length argument:

```
// Use varargs with standard arguments.
class VarArgs2 {

  // Here, msg is a normal parameter and v is a
  // varargs parameter.
  static void vaTest(String msg, int ... v) {
    System.out.print(msg + v.length +
                     " Contents: ");

    for(int x : v)
      System.out.print(x + " ");

    System.out.println();
  }

  public static void main(String args[])
  {
    vaTest("One vararg: ", 10);
    vaTest("Three varargs: ", 1, 2, 3);
    vaTest("No varargs: ");
  }
}
```

The output from this program is shown here:

```
One vararg: 1 Contents: 10
Three varargs: 3 Contents: 1 2 3
No varargs: 0 Contents:
```

Overloading Vararg Methods

You can overload a method that takes a variable-length argument. For example, the following program overloads **vaTest( )** three times:

```
// Varargs and overloading.
class VarArgs3 {

  static void vaTest(int ... v) {
    System.out.print("vaTest(int ...): " +
                     "Number of args: " + v.length +
                     " Contents: ");

    for(int x : v)
      System.out.print(x + " ");

    System.out.println();
  }

  static void vaTest(boolean ... v) {
    System.out.print("vaTest(boolean ...) " +
                     "Number of args: " + v.length +
                     " Contents: ");

    for(boolean x : v)
      System.out.print(x + " ");

    System.out.println();
  }

  static void vaTest(String msg, int ... v) {
    System.out.print("vaTest(String, int ...): " +
                     msg + v.length +
                     " Contents: ");

    for(int x : v)
      System.out.print(x + " ");

    System.out.println();
  }

  public static void main(String args[])
  {
    vaTest(1, 2, 3);
    vaTest("Testing: ", 10, 20);
    vaTest(true, false, false);
  }
}
```

The output produced by this program is shown here:

```
vaTest(int ...): Number of args: 3 Contents: 1 2 3
vaTest(String, int ...): Testing: 2 Contents: 10 20
vaTest(boolean ...) Number of args: 3 Contents: true false false
```

This program illustrates both ways that a varargs method can be overloaded. First, the types of its vararg parameter can differ. This is the case for **vaTest(int ...)** and **vaTest(boolean ...)**. Remember, the **...** causes the parameter to be treated as an array of the specified type. Therefore, just as you can overload methods by using different types of array parameters, you can overload vararg methods by using different types of varargs. In this case, Java uses the type difference to determine which overloaded method to call.

The second way to overload a varargs method is to add one or more normal parameters. This is what was done with **vaTest(String, int ...)**. In this case, Java uses both the number of arguments and the type of the arguments to determine which method to call.

NOTE A varargs method can also be overloaded by a non-varargs method. For example, **vaTest(int x)** is a valid overload of **vaTest( )** in the foregoing program. This version is invoked only when one **int** argument is present. When two or more **int** arguments are passed, the varargs version **vaTest (int...v)** is used.

Varargs and Ambiguity

Somewhat unexpected errors can result when overloading a method that takes a variable-length argument. These errors involve ambiguity because it is possible to create an ambiguous call to an overloaded varargs method. For example, consider the following program:

```
// Varargs, overloading, and ambiguity.
//
// This program contains an error and will
// not compile!
class VarArgs4 {

  static void vaTest(int ... v) {
    System.out.print("vaTest(int ...): " +
                     "Number of args: " + v.length +
                     " Contents: ");

    for(int x : v)
      System.out.print(x + " ");

    System.out.println();
  }

  static void vaTest(boolean ... v) {
    System.out.print("vaTest(boolean ...) " +
                     "Number of args: " + v.length +
                     " Contents: ");

    for(boolean x : v)
      System.out.print(x + " ");

    System.out.println();
  }

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

```
    vaTest(1, 2, 3); // OK
    vaTest(true, false, false); // OK

    vaTest(); // Error: Ambiguous!
  }
}
```

In this program, the overloading of **vaTest( )** is perfectly correct. However, this program will not compile because of the following call:

```
vaTest(); // Error: Ambiguous!
```

Because the vararg parameter can be empty, this call could be translated into a call to **vaTest(int ...)** or **vaTest(boolean ...)**. Both are equally valid. Thus, the call is inherently ambiguous.

Here is another example of ambiguity. The following overloaded versions of **vaTest( )** are inherently ambiguous even though one takes a normal parameter:

```
static void vaTest(int ... v) { // ...

static void vaTest(int n, int ... v) { // ...
```

Although the parameter lists of **vaTest( )** differ, there is no way for the compiler to resolve the following call:

```
vaTest(1)
```

Does this translate into a call to **vaTest(int ...)**, with one varargs argument, or into a call to **vaTest(int, int ...)** with no varargs arguments? There is no way for the compiler to answer this question. Thus, the situation is ambiguous.

Because of ambiguity errors like those just shown, sometimes you will need to forego overloading and simply use two different method names. Also, in some cases, ambiguity errors expose a conceptual flaw in your code, which you can remedy by more carefully crafting a solution.

CHAPTER

8 Inheritance

Inheritance is one of the cornerstones of object-oriented programming because it allows the creation of hierarchical classifications. Using inheritance, you can create a general class that defines traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to it. In the terminology of Java, a class that is inherited is called a *superclass*. The class that does the inheriting is called a *subclass*. Therefore, a subclass is a specialized version of a superclass. It inherits all of the members defined by the superclass and adds its own, unique elements.

Inheritance Basics

To inherit a class, you simply incorporate the definition of one class into another by using the **extends** keyword. To see how, let's begin with a short example. The following program creates a superclass called **A** and a subclass called **B**. Notice how the keyword **extends** is used to create a subclass of **A**.

```
// A simple example of inheritance.

// Create a superclass.
class A {
  int i, j;

  void showij() {
    System.out.println("i and j: " + i + " " + j);
  }
}

// Create a subclass by extending class A.
class B extends A {
  int k;

  void showk() {
    System.out.println("k: " + k);
  }
```

```
  void sum() {
    System.out.println("i+j+k: " + (i+j+k));
  }
}

class SimpleInheritance {
  public static void main(String args []) {
    A superOb = new A();
    B subOb = new B();

    // The superclass may be used by itself.
    superOb.i = 10;
    superOb.j = 20;
    System.out.println("Contents of superOb: ");
    superOb.showij();
    System.out.println();

    /* The subclass has access to all public members of
       its superclass. */
    subOb.i = 7;
    subOb.j = 8;
    subOb.k = 9;
    System.out.println("Contents of subOb: ");
    subOb.showij();
    subOb.showk();
    System.out.println();

    System.out.println("Sum of i, j and k in subOb:");
    subOb.sum();
  }
}
```

The output from this program is shown here:

```
Contents of superOb:
i and j: 10 20

Contents of subOb:
i and j: 7 8
k: 9

Sum of i, j and k in subOb:
i+j+k: 24
```

As you can see, the subclass **B** includes all of the members of its superclass, **A**. This is why **subOb** can access **i** and **j** and call **showij( )**. Also, inside **sum( )**, **i** and **j** can be referred to directly, as if they were part of **B**.

Even though **A** is a superclass for **B**, it is also a completely independent, stand-alone class. Being a superclass for a subclass does not mean that the superclass cannot be used by itself. Further, a subclass can be a superclass for another subclass.

The general form of a **class** declaration that inherits a superclass is shown here:

class *subclass-name* extends *superclass-name* {
  // body of class
}

You can only specify one superclass for any subclass that you create. Java does not support the inheritance of multiple superclasses into a single subclass. You can, as stated, create a hierarchy of inheritance in which a subclass becomes a superclass of another subclass. However, no class can be a superclass of itself.

Member Access and Inheritance

Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as **private**. For example, consider the following simple class hierarchy:

```
/* In a class hierarchy, private members remain
   private to their class.

   This program contains an error and will not
   compile.
*/

// Create a superclass.
class A {
  int i; // public by default
  private int j; // private to A

  void setij(int x, int y) {
    i = x;
    j = y;
  }
}

// A's j is not accessible here.
class B extends A {
  int total;

  void sum() {
    total = i + j; // ERROR, j is not accessible here
  }
}

class Access {
  public static void main(String args[]) {
    B subOb = new B();

    subOb.setij(10, 12);

    subOb.sum();
    System.out.println("Total is " + subOb.total);
  }
}
```

This program will not compile because the use of **j** inside the **sum( )** method of **B** causes an access violation. Since **j** is declared as **private**, it is only accessible by other members of its own class. Subclasses have no access to it.

REMEMBER A class member that has been declared as private will remain private to its class. It is not accessible by any code outside its class, including subclasses.

A More Practical Example

Let's look at a more practical example that will help illustrate the power of inheritance. Here, the final version of the **Box** class developed in the preceding chapter will be extended to include a fourth component called **weight**. Thus, the new class will contain a box's width, height, depth, and weight.

```
// This program uses inheritance to extend Box.
class Box {
  double width;
  double height;
  double depth;

  // construct clone of an object
  Box(Box ob) { // pass object to constructor
    width = ob.width;
    height = ob.height;
    depth = ob.depth;
  }

  // constructor used when all dimensions specified
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  // constructor used when no dimensions specified
  Box() {
    width = -1;  // use  -1 to indicate
    height = -1; // an uninitialized
    depth = -1;  // box
  }

  // constructor used when cube is created
  Box(double len) {
    width = height = depth = len;
  }

  // compute and return volume
  double volume() {
    return width * height * depth;
  }
}

// Here, Box is extended to include weight.
class BoxWeight extends Box {
```

```
  double weight; // weight of box

  // constructor for BoxWeight
  BoxWeight(double w, double h, double d, double m) {
    width = w;
    height = h;
    depth = d;
    weight = m;
  }
}

class DemoBoxWeight {
  public static void main(String args[]) {
    BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
    BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
    double vol;

    vol = mybox1.volume();
    System.out.println("Volume of mybox1 is " + vol);
    System.out.println("Weight of mybox1 is " + mybox1.weight);
    System.out.println();

    vol = mybox2.volume();
    System.out.println("Volume of mybox2 is " + vol);
    System.out.println("Weight of mybox2 is " + mybox2.weight);
  }
}
```

The output from this program is shown here:

```
Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3

Volume of mybox2 is 24.0
Weight of mybox2 is 0.076
```

BoxWeight inherits all of the characteristics of **Box** and adds to them the **weight** component. It is not necessary for **BoxWeight** to re-create all of the features found in **Box**. It can simply extend **Box** to meet its own purposes.

A major advantage of inheritance is that once you have created a superclass that defines the attributes common to a set of objects, it can be used to create any number of more specific subclasses. Each subclass can precisely tailor its own classification. For example, the following class inherits **Box** and adds a color attribute:

```
// Here, Box is extended to include color.
class ColorBox extends Box {
  int color; // color of box

  ColorBox(double w, double h, double d, int c) {
    width = w;
    height = h;
    depth = d;
    color = c;
  }
}
```

Remember, once you have created a superclass that defines the general aspects of an object, that superclass can be inherited to form specialized classes. Each subclass simply adds its own unique attributes. This is the essence of inheritance.

A Superclass Variable Can Reference a Subclass Object

A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass. You will find this aspect of inheritance quite useful in a variety of situations. For example, consider the following:

```
class RefDemo {
  public static void main(String args[]) {
    BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);
    Box plainbox = new Box();
    double vol;

    vol = weightbox.volume();
    System.out.println("Volume of weightbox is " + vol);
    System.out.println("Weight of weightbox is " +
                       weightbox.weight);
    System.out.println();

    // assign BoxWeight reference to Box reference
    plainbox = weightbox;

    vol = plainbox.volume(); // OK, volume() defined in Box
    System.out.println("Volume of plainbox is " + vol);

    /* The following statement is invalid because plainbox
       does not define a weight member. */
//  System.out.println("Weight of plainbox is " + plainbox.weight);
  }
}
```

Here, **weightbox** is a reference to **BoxWeight** objects, and **plainbox** is a reference to **Box** objects. Since **BoxWeight** is a subclass of **Box**, it is permissible to assign **plainbox** a reference to the **weightbox** object.

It is important to understand that it is the type of the reference variable—not the type of the object that it refers to—that determines what members can be accessed. That is, when a reference to a subclass object is assigned to a superclass reference variable, you will have access only to those parts of the object defined by the superclass. This is why **plainbox** can't access **weight** even when it refers to a **BoxWeight** object. If you think about it, this makes sense, because the superclass has no knowledge of what a subclass adds to it. This is why the last line of code in the preceding fragment is commented out. It is not possible for a **Box** reference to access the **weight** field, because **Box** does not define one.

Although the preceding may seem a bit esoteric, it has some important practical applications—two of which are discussed later in this chapter.

Using super

In the preceding examples, classes derived from **Box** were not implemented as efficiently or as robustly as they could have been. For example, the constructor for **BoxWeight** explicitly initializes the **width**, **height**, and **depth** fields of **Box**. Not only does this duplicate code found in its superclass, which is inefficient, but it implies that a subclass must be granted access to these members. However, there will be times when you will want to create a superclass that keeps the details of its implementation to itself (that is, that keeps its data members private). In this case, there would be no way for a subclass to directly access or initialize these variables on its own. Since encapsulation is a primary attribute of OOP, it is not surprising that Java provides a solution to this problem. Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword **super**.

super has two general forms. The first calls the superclass' constructor. The second is used to access a member of the superclass that has been hidden by a member of a subclass. Each use is examined here.

Using super to Call Superclass Constructors

A subclass can call a constructor defined by its superclass by use of the following form of **super**:

super(*arg-list*);

Here, *arg-list* specifies any arguments needed by the constructor in the superclass. **super( )** must always be the first statement executed inside a subclass' constructor.

To see how **super( )** is used, consider this improved version of the **BoxWeight** class:

```
// BoxWeight now uses super to initialize its Box attributes.
class BoxWeight extends Box {
  double weight; // weight of box

  // initialize width, height, and depth using super()
  BoxWeight(double w, double h, double d, double m) {
    super(w, h, d); // call superclass constructor
    weight = m;
  }
}
```

Here, **BoxWeight( )** calls **super( )** with the arguments **w**, **h**, and **d**. This causes the **Box** constructor to be called, which initializes **width**, **height**, and **depth** using these values. **BoxWeight** no longer initializes these values itself. It only needs to initialize the value unique to it: **weight**. This leaves **Box** free to make these values **private** if desired.

In the preceding example, **super( )** was called with three arguments. Since constructors can be overloaded, **super( )** can be called using any form defined by the superclass. The constructor executed will be the one that matches the arguments. For example, here is a complete implementation of **BoxWeight** that provides constructors for the various ways that a box can be constructed. In each case, **super( )** is called using the appropriate arguments. Notice that **width**, **height**, and **depth** have been made private within **Box**.

```
// A complete implementation of BoxWeight.
class Box {
  private double width;
  private double height;
  private double depth;

  // construct clone of an object
  Box(Box ob) { // pass object to constructor
    width = ob.width;
    height = ob.height;
    depth = ob.depth;
  }

  // constructor used when all dimensions specified
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  // constructor used when no dimensions specified
  Box() {
    width = -1;  // use -1 to indicate
    height = -1; // an uninitialized
    depth = -1;  // box
  }

  // constructor used when cube is created
  Box(double len) {
    width = height = depth = len;
  }

  // compute and return volume
  double volume() {
    return width * height * depth;
  }
 }

// BoxWeight now fully implements all constructors.
class BoxWeight extends Box {
  double weight; // weight of box

  // construct clone of an object
  BoxWeight(BoxWeight ob) { // pass object to constructor
    super(ob);
    weight = ob.weight;
  }

  // constructor when all parameters are specified
  BoxWeight(double w, double h, double d, double m) {
```

```
    super(w, h, d); // call superclass constructor
    weight = m;
  }

  // default constructor
  BoxWeight() {
    super();
    weight = -1;
  }

  // constructor used when cube is created
  BoxWeight(double len, double m) {
    super(len);
    weight = m;
  }
}

class DemoSuper {
  public static void main(String args[]) {
    BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
    BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
    BoxWeight mybox3 = new BoxWeight(); // default
    BoxWeight mycube = new BoxWeight(3, 2);
    BoxWeight myclone = new BoxWeight(mybox1);
    double vol;

    vol = mybox1.volume();
    System.out.println("Volume of mybox1 is " + vol);
    System.out.println("Weight of mybox1 is " + mybox1.weight);
    System.out.println();

    vol = mybox2.volume();
    System.out.println("Volume of mybox2 is " + vol);
    System.out.println("Weight of mybox2 is " + mybox2.weight);
    System.out.println();

    vol = mybox3.volume();
    System.out.println("Volume of mybox3 is " + vol);
    System.out.println("Weight of mybox3 is " + mybox3.weight);
    System.out.println();

    vol = myclone.volume();
    System.out.println("Volume of myclone is " + vol);
    System.out.println("Weight of myclone is " + myclone.weight);
    System.out.println();

    vol = mycube.volume();
    System.out.println("Volume of mycube is " + vol);
    System.out.println("Weight of mycube is " + mycube.weight);
    System.out.println();
  }
}
```

This program generates the following output:

```
Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3

Volume of mybox2 is 24.0
Weight of mybox2 is 0.076

Volume of mybox3 is -1.0
Weight of mybox3 is -1.0

Volume of myclone is 3000.0
Weight of myclone is 34.3

Volume of mycube is 27.0
Weight of mycube is 2.0
```

Pay special attention to this constructor in **BoxWeight**:

```
// construct clone of an object
BoxWeight(BoxWeight ob) { // pass object to constructor
  super(ob);
  weight = ob.weight;
}
```

Notice that **super( )** is passed an object of type **BoxWeight—**not of type **Box**. This still invokes the constructor **Box(Box ob)**. As mentioned earlier, a superclass variable can be used to reference any object derived from that class. Thus, we are able to pass a **BoxWeight** object to the **Box** constructor. Of course, **Box** only has knowledge of its own members.

Let's review the key concepts behind **super( )**. When a subclass calls **super( )**, it is calling the constructor of its immediate superclass. Thus, **super( )** always refers to the superclass immediately above the calling class. This is true even in a multileveled hierarchy. Also, **super( )** must always be the first statement executed inside a subclass constructor.

A Second Use for super

The second form of **super** acts somewhat like **this**, except that it always refers to the superclass of the subclass in which it is used. This usage has the following general form:

super.*member*

Here, *member* can be either a method or an instance variable.

This second form of **super** is most applicable to situations in which member names of a subclass hide members by the same name in the superclass. Consider this simple class hierarchy:

```
// Using super to overcome name hiding.
class A {
  int i;
}

// Create a subclass by extending class A.
```

```
class B extends A {
  int i; // this i hides the i in A

  B(int a, int b) {
    super.i = a; // i in A
    i = b; // i in B
  }

  void show() {
    System.out.println("i in superclass: " + super.i);
    System.out.println("i in subclass: " + i);
  }
}

class UseSuper {
  public static void main(String args[]) {
    B subOb = new B(1, 2);

    subOb.show();
  }
}
```

This program displays the following:

```
i in superclass: 1
i in subclass: 2
```

Although the instance variable **i** in **B** hides the **i** in **A**, **super** allows access to the **i** defined in the superclass. As you will see, **super** can also be used to call methods that are hidden by a subclass.

Creating a Multilevel Hierarchy

Up to this point, we have been using simple class hierarchies that consist of only a superclass and a subclass. However, you can build hierarchies that contain as many layers of inheritance as you like. As mentioned, it is perfectly acceptable to use a subclass as a superclass of another. For example, given three classes called **A**, **B**, and **C**, **C** can be a subclass of **B**, which is a subclass of **A**. When this type of situation occurs, each subclass inherits all of the traits found in all of its superclasses. In this case, **C** inherits all aspects of **B** and **A**. To see how a multilevel hierarchy can be useful, consider the following program. In it, the subclass **BoxWeight** is used as a superclass to create the subclass called **Shipment**. **Shipment** inherits all of the traits of **BoxWeight** and **Box**, and adds a field called **cost**, which holds the cost of shipping such a parcel.

```
// Extend BoxWeight to include shipping costs.

// Start with Box.
class Box {
  private double width;
  private double height;
  private double depth;
```

```
  // construct clone of an object
  Box(Box ob) { // pass object to constructor
    width = ob.width;
    height = ob.height;
    depth = ob.depth;
  }

  // constructor used when all dimensions specified
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  // constructor used when no dimensions specified
  Box() {
    width = -1;  // use -1 to indicate
    height = -1; // an uninitialized
    depth = -1;  // box
  }

  // constructor used when cube is created
  Box(double len) {
    width = height = depth = len;
  }

  // compute and return volume
  double volume() {
    return width * height * depth;
  }
}

// Add weight.
class BoxWeight extends Box {
  double weight; // weight of box

  // construct clone of an object
  BoxWeight(BoxWeight ob) { // pass object to constructor
    super(ob);
    weight = ob.weight;
  }

  // constructor when all parameters are specified
  BoxWeight(double w, double h, double d, double m) {
    super(w, h, d); // call superclass constructor
    weight = m;
  }

  // default constructor
  BoxWeight() {
    super();
    weight = -1;
  }
```

```
  // constructor used when cube is created
  BoxWeight(double len, double m) {
    super(len);
    weight = m;
  }
}

// Add shipping costs.
class Shipment extends BoxWeight {
  double cost;

  // construct clone of an object
  Shipment(Shipment ob) { // pass object to constructor
    super(ob);
    cost = ob.cost;
  }

  // constructor when all parameters are specified
  Shipment(double w, double h, double d,
            double m, double c) {
    super(w, h, d, m); // call superclass constructor
    cost = c;
  }

  // default constructor
  Shipment() {
    super();
    cost = -1;
  }

  // constructor used when cube is created
  Shipment(double len, double m, double c) {
    super(len, m);
    cost = c;
  }
}

class DemoShipment {
  public static void main(String args[]) {
    Shipment shipment1 =
               new Shipment(10, 20, 15, 10, 3.41);
    Shipment shipment2 =
               new Shipment(2, 3, 4, 0.76, 1.28);

    double vol;

    vol = shipment1.volume();
    System.out.println("Volume of shipment1 is " + vol);
    System.out.println("Weight of shipment1 is "
                        + shipment1.weight);
    System.out.println("Shipping cost: $" + shipment1.cost);
    System.out.println();
```

```
    vol = shipment2.volume();
    System.out.println("Volume of shipment2 is " + vol);
    System.out.println("Weight of shipment2 is "
                        + shipment2.weight);
    System.out.println("Shipping cost: $" + shipment2.cost);
  }
}
```

The output of this program is shown here:

```
Volume of shipment1 is 3000.0
Weight of shipment1 is 10.0
Shipping cost: $3.41

Volume of shipment2 is 24.0
Weight of shipment2 is 0.76
Shipping cost: $1.28
```

Because of inheritance, **Shipment** can make use of the previously defined classes of **Box** and **BoxWeight**, adding only the extra information it needs for its own, specific application. This is part of the value of inheritance; it allows the reuse of code.

This example illustrates one other important point: **super( )** always refers to the constructor in the closest superclass. The **super( )** in **Shipment** calls the constructor in **BoxWeight**. The **super( )** in **BoxWeight** calls the constructor in **Box**. In a class hierarchy, if a superclass constructor requires parameters, then all subclasses must pass those parameters "up the line." This is true whether or not a subclass needs parameters of its own.

NOTE In the preceding program, the entire class hierarchy, including **Box**, **BoxWeight**, and **Shipment**, is shown all in one file. This is for your convenience only. In Java, all three classes could have been placed into their own files and compiled separately. In fact, using separate files is the norm, not the exception, in creating class hierarchies.

When Constructors Are Executed

When a class hierarchy is created, in what order are the constructors for the classes that make up the hierarchy executed? For example, given a subclass called **B** and a superclass called **A**, is **A**'s constructor executed before **B**'s, or vice versa? The answer is that in a class hierarchy, constructors complete their execution in order of derivation, from superclass to subclass. Further, since **super( )** must be the first statement executed in a subclass' constructor, this order is the same whether or not **super( )** is used. If **super( )** is not used, then the default or parameterless constructor of each superclass will be executed. The following program illustrates when constructors are executed:

```
// Demonstrate when constructors are executed.

// Create a super class.
class A {
  A() {
    System.out.println("Inside A's constructor.");
  }
}
```

```
// Create a subclass by extending class A.
class B extends A {
  B() {
    System.out.println("Inside B's constructor.");
  }
}

// Create another subclass by extending B.
class C extends B {
  C() {
    System.out.println("Inside C's constructor.");
  }
}

class CallingCons {
  public static void main(String args[]) {
    C c = new C();
  }
}
```

The output from this program is shown here:

```
Inside A's constructor
Inside B's constructor
Inside C's constructor
```

As you can see, the constructors are executed in order of derivation.

If you think about it, it makes sense that constructors complete their execution in order of derivation. Because a superclass has no knowledge of any subclass, any initialization it needs to perform is separate from and possibly prerequisite to any initialization performed by the subclass. Therefore, it must complete its execution first.

Method Overriding

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass. When an overridden method is called from within its subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden. Consider the following:

```
// Method overriding.
class A {
  int i, j;
  A(int a, int b) {
    i = a;
    j = b;
  }

  // display i and j
  void show() {
    System.out.println("i and j: " + i + " " + j);
  }
}
```

```
class B extends A {
  int k;

  B(int a, int b, int c) {
    super(a, b);
    k = c;
  }

  // display k - this overrides show() in A
  void show() {
    System.out.println("k: " + k);
  }
}

class Override {
  public static void main(String args[]) {
    B subOb = new B(1, 2, 3);

   subOb.show(); // this calls show() in B
  }
}
```

The output produced by this program is shown here:

```
k: 3
```

When **show( )** is invoked on an object of type **B**, the version of **show( )** defined within **B** is used. That is, the version of **show( )** inside **B** overrides the version declared in **A**.

If you wish to access the superclass version of an overridden method, you can do so by using **super**. For example, in this version of **B**, the superclass version of **show( )** is invoked within the subclass' version. This allows all instance variables to be displayed.

```
class B extends A {
  int k;

  B(int a, int b, int c) {
    super(a, b);
    k = c;
  }

  void show() {
    super.show(); // this calls A's show()
    System.out.println("k: " + k);
  }
}
```

If you substitute this version of **A** into the previous program, you will see the following output:

```
i and j: 1 2
k: 3
```

Here, **super.show( )** calls the superclass version of **show( )**.

Method overriding occurs *only* when the names and the type signatures of the two methods are identical. If they are not, then the two methods are simply overloaded. For example, consider this modified version of the preceding example:

```
// Methods with differing type signatures are overloaded – not
// overridden.
class A {
  int i, j;

  A(int a, int b) {
    i = a;
    j = b;
  }

  // display i and j
  void show() {
    System.out.println("i and j: " + i + " " + j);
  }
}

// Create a subclass by extending class A.
class B extends A {
  int k;

  B(int a, int b, int c) {
    super(a, b);
    k = c;
  }

  // overload show()
  void show(String msg) {
    System.out.println(msg + k);
  }
}

class Override {
  public static void main(String args[]) {
    B subOb = new B(1, 2, 3);

    subOb.show("This is k: "); // this calls show() in B
    subOb.show(); // this calls show() in A
  }
}
```

The output produced by this program is shown here:

```
This is k: 3
i and j: 1 2
```

The version of **show( )** in **B** takes a string parameter. This makes its type signature different from the one in **A**, which takes no parameters. Therefore, no overriding (or name hiding) takes place. Instead, the version of **show( )** in **B** simply overloads the version of **show( )** in **A**.

Dynamic Method Dispatch

While the examples in the preceding section demonstrate the mechanics of method overriding, they do not show its power. Indeed, if there were nothing more to method overriding than a name space convention, then it would be, at best, an interesting curiosity, but of little real value. However, this is not the case. Method overriding forms the basis for one of Java's most powerful concepts: *dynamic method dispatch*. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

Let's begin by restating an important principle: a superclass reference variable can refer to a subclass object. Java uses this fact to resolve calls to overridden methods at run time. Here is how. When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time. When different types of objects are referred to, different versions of an overridden method will be called. In other words, *it is the type of the object being referred to* (not the type of the reference variable) that determines which version of an overridden method will be executed. Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

Here is an example that illustrates dynamic method dispatch:

```
// Dynamic Method Dispatch
class A {
  void callme() {
    System.out.println("Inside A's callme method");
  }
}

class B extends A {
  // override callme()
  void callme() {
    System.out.println("Inside B's callme method");
  }
}

class C extends A {
  // override callme()
  void callme() {
    System.out.println("Inside C's callme method");
  }
}

class Dispatch {
  public static void main(String args[]) {
    A a = new A(); // object of type A
    B b = new B(); // object of type B
    C c = new C(); // object of type C
```

```
    A r; // obtain a reference of type A

    r = a; // r refers to an A object
    r.callme(); // calls A's version of callme

    r = b; // r refers to a B object
    r.callme(); // calls B's version of callme

    r = c; // r refers to a C object
    r.callme(); // calls C's version of callme
  }
}
```

The output from the program is shown here:

```
Inside A's callme method
Inside B's callme method
Inside C's callme method
```

This program creates one superclass called **A** and two subclasses of it, called **B** and **C**. Subclasses **B** and **C** override **callme( )** declared in **A**. Inside the **main( )** method, objects of type **A**, **B**, and **C** are declared. Also, a reference of type **A**, called **r**, is declared. The program then in turn assigns a reference to each type of object to **r** and uses that reference to invoke **callme( )**. As the output shows, the version of **callme( )** executed is determined by the type of object being referred to at the time of the call. Had it been determined by the type of the reference variable, **r**, you would see three calls to **A**'s **callme( )** method.

NOTE Readers familiar with C++ or C# will recognize that overridden methods in Java are similar to virtual functions in those languages.

Why Overridden Methods?

As stated earlier, overridden methods allow Java to support run-time polymorphism. Polymorphism is essential to object-oriented programming for one reason: it allows a general class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods. Overridden methods are another way that Java implements the "one interface, multiple methods" aspect of polymorphism.

Part of the key to successfully applying polymorphism is understanding that the superclasses and subclasses form a hierarchy which moves from lesser to greater specialization. Used correctly, the superclass provides all elements that a subclass can use directly. It also defines those methods that the derived class must implement on its own. This allows the subclass the flexibility to define its own methods, yet still enforces a consistent interface. Thus, by combining inheritance with overridden methods, a superclass can define the general form of the methods that will be used by all of its subclasses.

Dynamic, run-time polymorphism is one of the most powerful mechanisms that object-oriented design brings to bear on code reuse and robustness. The ability of existing code libraries to call methods on instances of new classes without recompiling while maintaining a clean abstract interface is a profoundly powerful tool.

Applying Method Overriding

Let's look at a more practical example that uses method overriding. The following program creates a superclass called **Figure** that stores the dimensions of a two-dimensional object. It also defines a method called **area( )** that computes the area of an object. The program derives two subclasses from **Figure**. The first is **Rectangle** and the second is **Triangle**. Each of these subclasses overrides **area( )** so that it returns the area of a rectangle and a triangle, respectively.

```
// Using run-time polymorphism.
class Figure {
  double dim1;
  double dim2;

  Figure(double a, double b) {
    dim1 = a;
    dim2 = b;
  }

  double area() {
    System.out.println("Area for Figure is undefined.");
    return 0;
  }
}

class Rectangle extends Figure {
  Rectangle(double a, double b) {
    super(a, b);
  }

  // override area for rectangle
  double area() {
    System.out.println("Inside Area for Rectangle.");
    return dim1 * dim2;
  }
}

class Triangle extends Figure {
  Triangle(double a, double b) {
    super(a, b);
  }

  // override area for right triangle
  double area() {
    System.out.println("Inside Area for Triangle.");
    return dim1 * dim2 / 2;
  }
}

class FindAreas {
  public static void main(String args[]) {
    Figure f = new Figure(10, 10);
    Rectangle r = new Rectangle(9, 5);
```

```
    Triangle t = new Triangle(10, 8);
    Figure figref;

    figref = r;
    System.out.println("Area is " + figref.area());

    figref = t;
    System.out.println("Area is " + figref.area());

    figref = f;
    System.out.println("Area is " + figref.area());
  }
}
```

The output from the program is shown here:

```
Inside Area for Rectangle.
Area is 45
Inside Area for Triangle.
Area is 40
Area for Figure is undefined.
Area is 0
```

Through the dual mechanisms of inheritance and run-time polymorphism, it is possible to define one consistent interface that is used by several different, yet related, types of objects. In this case, if an object is derived from **Figure**, then its area can be obtained by calling **area( )**. The interface to this operation is the same no matter what type of figure is being used.

Using Abstract Classes

There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. That is, sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement. One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method. This is the case with the class **Figure** used in the preceding example. The definition of **area( )** is simply a placeholder. It will not compute and display the area of any type of object.

As you will see as you create your own class libraries, it is not uncommon for a method to have no meaningful definition in the context of its superclass. You can handle this situation two ways. One way, as shown in the previous example, is to simply have it report a warning message. While this approach can be useful in certain situations—such as debugging—it is not usually appropriate. You may have methods that must be overridden by the subclass in order for the subclass to have any meaning. Consider the class **Triangle**. It has no meaning if **area( )** is not defined. In this case, you want some way to ensure that a subclass does, indeed, override all necessary methods. Java's solution to this problem is the *abstract method*.

You can require that certain methods be overridden by subclasses by specifying the **abstract** type modifier. These methods are sometimes referred to as *subclasser responsibility* because they have no implementation specified in the superclass. Thus, a subclass must override them—it cannot simply use the version defined in the superclass. To declare an abstract method, use this general form:

abstract *type name*(*parameter-list*);

As you can see, no method body is present.

Any class that contains one or more abstract methods must also be declared abstract. To declare a class abstract, you simply use the **abstract** keyword in front of the **class** keyword at the beginning of the class declaration. There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the **new** operator. Such objects would be useless, because an abstract class is not fully defined. Also, you cannot declare abstract constructors, or abstract static methods. Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be declared **abstract** itself.

Here is a simple example of a class with an abstract method, followed by a class which implements that method:

```
// A Simple demonstration of abstract.
abstract class A {
  abstract void callme();

  // concrete methods are still allowed in abstract classes
  void callmetoo() {
    System.out.println("This is a concrete method.");
  }
}

class B extends A {
  void callme() {
    System.out.println("B's implementation of callme.");
  }
}

class AbstractDemo {
  public static void main(String args[]) {
    B b = new B();

    b.callme();
    b.callmetoo();
  }
}
```

Notice that no objects of class **A** are declared in the program. As mentioned, it is not possible to instantiate an abstract class. One other point: class **A** implements a concrete method called **callmetoo( )**. This is perfectly acceptable. Abstract classes can include as much implementation as they see fit.

Although abstract classes cannot be used to instantiate objects, they can be used to create object references, because Java's approach to run-time polymorphism is implemented through the use of superclass references. Thus, it must be possible to create a reference to an abstract class so that it can be used to point to a subclass object. You will see this feature put to use in the next example.

Using an abstract class, you can improve the **Figure** class shown earlier. Since there is no meaningful concept of area for an undefined two-dimensional figure, the following version of the program declares **area( )** as abstract inside **Figure**. This, of course, means that all classes derived from **Figure** must override **area( )**.

```
// Using abstract methods and classes.
abstract class Figure {
  double dim1;
  double dim2;

  Figure(double a, double b) {
    dim1 = a;
    dim2 = b;
  }

  // area is now an abstract method
  abstract double area();
}

class Rectangle extends Figure {
  Rectangle(double a, double b) {
    super(a, b);
  }

  // override area for rectangle
  double area() {
    System.out.println("Inside Area for Rectangle.");
    return dim1 * dim2;
  }
}

class Triangle extends Figure {
  Triangle(double a, double b) {
    super(a, b);
  }

  // override area for right triangle
  double area() {
    System.out.println("Inside Area for Triangle.");
    return dim1 * dim2 / 2;
  }
}

class AbstractAreas {
  public static void main(String args[]) {
  // Figure f = new Figure(10, 10); // illegal now
    Rectangle r = new Rectangle(9, 5);
    Triangle t = new Triangle(10, 8);
    Figure figref; // this is OK, no object is created

    figref = r;
    System.out.println("Area is " + figref.area());

    figref = t;
```

```
    System.out.println("Area is " + figref.area());
  }
}
```

As the comment inside **main( )** indicates, it is no longer possible to declare objects of type **Figure**, since it is now abstract. And, all subclasses of **Figure** must override **area( )**. To prove this to yourself, try creating a subclass that does not override **area( )**. You will receive a compile-time error.

Although it is not possible to create an object of type **Figure**, you can create a reference variable of type **Figure**. The variable **figref** is declared as a reference to **Figure**, which means that it can be used to refer to an object of any class derived from **Figure**. As explained, it is through superclass reference variables that overridden methods are resolved at run time.

Using final with Inheritance

The keyword **final** has three uses. First, it can be used to create the equivalent of a named constant. This use was described in the preceding chapter. The other two uses of **final** apply to inheritance. Both are examined here.

Using final to Prevent Overriding

While method overriding is one of Java's most powerful features, there will be times when you will want to prevent it from occurring. To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration. Methods declared as **final** cannot be overridden. The following fragment illustrates **final**:

```
class A {
  final void meth() {
    System.out.println("This is a final method.");
  }
}

class B extends A {
  void meth() { // ERROR! Can't override.
    System.out.println("Illegal!");
  }
}
```

Because **meth( )** is declared as **final**, it cannot be overridden in **B**. If you attempt to do so, a compile-time error will result.

Methods declared as **final** can sometimes provide a performance enhancement: The compiler is free to *inline* calls to them because it "knows" they will not be overridden by a subclass. When a small **final** method is called, often the Java compiler can copy the bytecode for the subroutine directly inline with the compiled code of the calling method, thus eliminating the costly overhead associated with a method call. Inlining is an option only with **final** methods. Normally, Java resolves calls to methods dynamically, at run time. This is called *late binding*. However, since **final** methods cannot be overridden, a call to one can be resolved at compile time. This is called early binding.

Using final to Prevent Inheritance

Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with **final**. Declaring a class as **final** implicitly declares all of its methods as **final**, too. As you might expect, it is illegal to declare a class as both **abstract** and **final** since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.

Here is an example of a **final** class:

```
final class A {
  //...
}

// The following class is illegal.
class B extends A { // ERROR! Can't subclass A
  //...
}
```

As the comments imply, it is illegal for **B** to inherit **A** since **A** is declared as **final**.

The Object Class

There is one special class, **Object**, defined by Java. All other classes are subclasses of **Object**. That is, **Object** is a superclass of all other classes. This means that a reference variable of type **Object** can refer to an object of any other class. Also, since arrays are implemented as classes, a variable of type **Object** can also refer to any array.

Object defines the following methods, which means that they are available in every object.

Method	Purpose
Object clone( )	Creates a new object that is the same as the object being cloned.
boolean equals(Object *object*)	Determines whether one object is equal to another.
void finalize( )	Called before an unused object is recycled.
Class<?> getClass( )	Obtains the class of an object at run time.
int hashCode( )	Returns the hash code associated with the invoking object.
void notify( )	Resumes execution of a thread waiting on the invoking object.
void notifyAll( )	Resumes execution of all threads waiting on the invoking object.
String toString( )	Returns a string that describes the object.
void wait( ) void wait(long *milliseconds*) void wait(long *milliseconds*, int *nanoseconds*)	Waits on another thread of execution.

The methods **getClass( )**, **notify( )**, **notifyAll( )**, and **wait( )** are declared as **final**. You may override the others. These methods are described elsewhere in this book. However, notice two methods now: **equals( )** and **toString( )**. The **equals( )** method compares two objects. It returns **true** if the objects are equal, and **false** otherwise. The precise definition of equality can vary, depending on the type of objects being compared. The **toString( )** method returns a string that contains a description of the object on which it is called. Also, this method is automatically called when an object is output using **println( )**. Many classes override this method. Doing so allows them to tailor a description specifically for the types of objects that they create.

One last point: Notice the unusual syntax in the return type for **getClass( )**. This relates to Java's *generics* feature, which is described in Chapter 14.

CHAPTER

9 Packages and Interfaces

This chapter examines two of Java's most innovative features: packages and interfaces. *Packages* are containers for classes. They are used to keep the class name space compartmentalized. For example, a package allows you to create a class named **List**, which you can store in your own package without concern that it will collide with some other class named **List** stored elsewhere. Packages are stored in a hierarchical manner and are explicitly imported into new class definitions.

In previous chapters, you have seen how methods define the interface to the data in a class. Through the use of the **interface** keyword, Java allows you to fully abstract an interface from its implementation. Using **interface**, you can specify a set of methods that can be implemented by one or more classes. In its traditional form, the **interface**, itself, does not actually define any implementation. Although they are similar to abstract classes, **interface**s have an additional capability: A class can implement more than one interface. By contrast, a class can only inherit a single superclass (abstract or otherwise).

Packages

In the preceding chapters, the name of each example class was taken from the same name space. This means that a unique name had to be used for each class to avoid name collisions. After a while, without some way to manage the name space, you could run out of convenient, descriptive names for individual classes. You also need some way to be assured that the name you choose for a class will be reasonably unique and not collide with class names chosen by other programmers. (Imagine a small group of programmers fighting over who gets to use the name "Foobar" as a class name. Or, imagine the entire Internet community arguing over who first named a class "Espresso.") Thankfully, Java provides a mechanism for partitioning the class name space into more manageable chunks. This mechanism is the package. The package is both a naming and a visibility control mechanism. You can define classes inside a package that are not accessible by code outside that package. You can also define class members that are exposed only to other members of the same package. This allows your classes to have intimate knowledge of each other, but not expose that knowledge to the rest of the world.

Defining a Package

To create a package is quite easy: simply include a **package** command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package. The **package** statement defines a name space in which classes are stored. If you omit the **package** statement, the class names are put into the default package, which has no name. (This is why you haven't had to worry about packages before now.) While the default package is fine for short, sample programs, it is inadequate for real applications. Most of the time, you will define a package for your code.

This is the general form of the **package** statement:

package *pkg*;

Here, *pkg* is the name of the package. For example, the following statement creates a package called **MyPackage**:

```
package MyPackage;
```

Java uses file system directories to store packages. For example, the **.class** files for any classes you declare to be part of **MyPackage** must be stored in a directory called **MyPackage**. Remember that case is significant, and the directory name must match the package name exactly.

More than one file can include the same **package** statement. The **package** statement simply specifies to which package the classes defined in a file belong. It does not exclude other classes in other files from being part of that same package. Most real-world packages are spread across many files.

You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period. The general form of a multileveled package statement is shown here:

package *pkg1*[.*pkg2*[.*pkg3*]];

A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as

```
package java.awt.image;
```

needs to be stored in **java\awt\image** in a Windows environment. Be sure to choose your package names carefully. You cannot rename a package without renaming the directory in which the classes are stored.

Finding Packages and CLASSPATH

As just explained, packages are mirrored by directories. This raises an important question: How does the Java run-time system know where to look for packages that you create? The answer has three parts. First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in a subdirectory of the current directory, it will be found. Second, you can specify a directory path or paths by setting the **CLASSPATH** environmental variable. Third, you can use the **-classpath** option with **java** and **javac** to specify the path to your classes.

For example, consider the following package specification:

```
package MyPack
```

In order for a program to find **MyPack**, one of three things must be true. Either the program can be executed from a directory immediately above **MyPack**, or the **CLASSPATH** must be set to include the path to **MyPack**, or the **-classpath** option must specify the path to **MyPack** when the program is run via **java**.

When the second two options are used, the class path *must not* include **MyPack**, itself. It must simply specify the *path to* **MyPack**. For example, in a Windows environment, if the path to **MyPack** is

```
C:\MyPrograms\Java\MyPack
```

then the class path to **MyPack** is

```
C:\MyPrograms\Java
```

The easiest way to try the examples shown in this book is to simply create the package directories below your current development directory, put the **.class** files into the appropriate directories, and then execute the programs from the development directory. This is the approach used in the following example.

A Short Package Example

Keeping the preceding discussion in mind, you can try this simple package:

```
// A simple package
package MyPack;

class Balance {
  String name;
  double bal;

  Balance(String n, double b) {
    name = n;
    bal = b;
  }

  void show() {
    if(bal<0)
      System.out.print("--> ");
    System.out.println(name + ": $" + bal);
  }
}

class AccountBalance {
  public static void main(String args[]) {
    Balance current[] = new Balance[3];

    current[0] = new Balance("K. J. Fielding", 123.23);
    current[1] = new Balance("Will Tell", 157.02);
    current[2] = new Balance("Tom Jackson", -12.33);
```

```
    for(int i=0; i<3; i++) current[i].show();
  }
}
```

Call this file **AccountBalance.java** and put it in a directory called **MyPack**.

Next, compile the file. Make sure that the resulting **.class** file is also in the **MyPack** directory. Then, try executing the **AccountBalance** class, using the following command line:

```
java MyPack.AccountBalance
```

Remember, you will need to be in the directory above **MyPack** when you execute this command. (Alternatively, you can use one of the other two options described in the preceding section to specify the path **MyPack**.)

As explained, **AccountBalance** is now part of the package **MyPack**. This means that it cannot be executed by itself. That is, you cannot use this command line:

```
java AccountBalance
```

AccountBalance must be qualified with its package name.

Access Protection

In the preceding chapters, you learned about various aspects of Java's access control mechanism and its access modifiers. For example, you already know that access to a **private** member of a class is granted only to other members of that class. Packages add another dimension to access control. As you will see, Java provides many levels of protection to allow fine-grained control over the visibility of variables and methods within classes, subclasses, and packages.

Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods. Packages act as containers for classes and other subordinate packages. Classes act as containers for data and code. The class is Java's smallest unit of abstraction. Because of the interplay between classes and packages, Java addresses four categories of visibility for class members:

- Subclasses in the same package
- Non-subclasses in the same package
- Subclasses in different packages
- Classes that are neither in the same package nor subclasses

The three access modifiers, **private**, **public**, and **protected**, provide a variety of ways to produce the many levels of access required by these categories. Table 9-1 sums up the interactions.

While Java's access control mechanism may seem complicated, we can simplify it as follows. Anything declared **public** can be accessed from anywhere. Anything declared **private** cannot be seen outside of its class. When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. This is the default access. If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element **protected**.

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Table 9-1 Class Member Access

Table 9-1 applies only to members of classes. A non-nested class has only two possible access levels: default and public. When a class is declared as **public**, it is accessible by any other code. If a class has default access, then it can only be accessed by other code within its same package. When a class is public, it must be the only public class declared in the file, and the file must have the same name as the class.

An Access Example

The following example shows all combinations of the access control modifiers. This example has two packages and five classes. Remember that the classes for the two different packages need to be stored in directories named after their respective packages—in this case, **p1** and **p2**.

The source for the first package defines three classes: **Protection**, **Derived**, and **SamePackage**. The first class defines four **int** variables in each of the legal protection modes. The variable **n** is declared with the default protection, **n_pri** is **private**, **n_pro** is **protected**, and **n_pub** is **public**.

Each subsequent class in this example will try to access the variables in an instance of this class. The lines that will not compile due to access restrictions are commented out. Before each of these lines is a comment listing the places from which this level of protection would allow access.

The second class, **Derived**, is a subclass of **Protection** in the same package, **p1**. This grants **Derived** access to every variable in **Protection** except for **n_pri**, the **private** one. The third class, **SamePackage**, is not a subclass of **Protection**, but is in the same package and also has access to all but **n_pri**.

This is file **Protection.java**:

```
package p1;

public class Protection {
  int n = 1;
  private int n_pri = 2;
  protected int n_pro = 3;
  public int n_pub = 4;

  public Protection() {
    System.out.println("base constructor");
    System.out.println("n = " + n);
    System.out.println("n_pri = " + n_pri);
    System.out.println("n_pro = " + n_pro);
```

```
    System.out.println("n_pub = " + n_pub);
  }
}
```

This is file **Derived.java**:

```
package p1;

class Derived extends Protection {
  Derived() {
    System.out.println("derived constructor");
    System.out.println("n = " + n);

// class only
// System.out.println("n_pri = "4 + n_pri);

   System.out.println("n_pro = " + n_pro);
   System.out.println("n_pub = " + n_pub);
  }
}
```

This is file **SamePackage.java**:

```
package p1;

class SamePackage {
  SamePackage() {

    Protection p = new Protection();
    System.out.println("same package constructor");
    System.out.println("n = " + p.n);

// class only
// System.out.println("n_pri = " + p.n_pri);

   System.out.println("n_pro = " + p.n_pro);
   System.out.println("n_pub = " + p.n_pub);
  }
}
```

Following is the source code for the other package, **p2**. The two classes defined in **p2** cover the other two conditions that are affected by access control. The first class, **Protection2**, is a subclass of **p1.Protection**. This grants access to all of **p1.Protection**'s variables except for **n_pri** (because it is **private**) and **n**, the variable declared with the default protection. Remember, the default only allows access from within the class or the package, not extra-package subclasses. Finally, the class **OtherPackage** has access to only one variable, **n_pub**, which was declared **public**.

This is file **Protection2.java**:

```
package p2;

class Protection2 extends p1.Protection {
  Protection2() {
```

```
    System.out.println("derived other package constructor");

//  class or package only
//  System.out.println("n = " + n);

//  class only
//  System.out.println("n_pri = " + n_pri);

    System.out.println("n_pro = " + n_pro);
    System.out.println("n_pub = " + n_pub);
  }
}
```

This is file **OtherPackage.java**:

```
package p2;

class OtherPackage {
  OtherPackage() {
    p1.Protection p = new p1.Protection();
    System.out.println("other package constructor");

//  class or package only
//  System.out.println("n = " + p.n);

//  class only
//  System.out.println("n_pri = " + p.n_pri);

//  class, subclass or package only
//  System.out.println("n_pro = " + p.n_pro);

    System.out.println("n_pub = " + p.n_pub);
  }
}
```

If you want to try these two packages, here are two test files you can use. The one for package **p1** is shown here:

```
// Demo package p1.
package p1;

// Instantiate the various classes in p1.
public class Demo {
  public static void main(String args[]) {
    Protection ob1 = new Protection();
    Derived ob2 = new Derived();
    SamePackage ob3 = new SamePackage();
  }
}
```

The test file for **p2** is shown next:

```
// Demo package p2.
package p2;
```

```
// Instantiate the various classes in p2.
public class Demo {
  public static void main(String args[]) {
    Protection2 ob1 = new Protection2();
    OtherPackage ob2 = new OtherPackage();
  }
}
```

Importing Packages

Given that packages exist and are a good mechanism for compartmentalizing diverse classes from each other, it is easy to see why all of the built-in Java classes are stored in packages. There are no core Java classes in the unnamed default package; all of the standard classes are stored in some named package. Since classes within packages must be fully qualified with their package name or names, it could become tedious to type in the long dot-separated package path name for every class you want to use. For this reason, Java includes the **import** statement to bring certain classes, or entire packages, into visibility. Once imported, a class can be referred to directly, using only its name. The **import** statement is a convenience to the programmer and is not technically needed to write a complete Java program. If you are going to refer to a few dozen classes in your application, however, the **import** statement will save a lot of typing.

In a Java source file, **import** statements occur immediately following the **package** statement (if it exists) and before any class definitions. This is the general form of the **import** statement:

import *pkg1* [.*pkg2*].(*classname* | *);

Here, *pkg1* is the name of a top-level package, and *pkg2* is the name of a subordinate package inside the outer package separated by a dot (.). There is no practical limit on the depth of a package hierarchy, except that imposed by the file system. Finally, you specify either an explicit *classname* or a star (*), which indicates that the Java compiler should import the entire package. This code fragment shows both forms in use:

```
import java.util.Date;
import java.io.*;
```

All of the standard Java classes included with Java are stored in a package called **java**. The basic language functions are stored in a package inside of the **java** package called **java.lang**. Normally, you have to import every package or class that you want to use, but since Java is useless without much of the functionality in **java.lang**, it is implicitly imported by the compiler for all programs. This is equivalent to the following line being at the top of all of your programs:

```
import java.lang.*;
```

If a class with the same name exists in two different packages that you import using the star form, the compiler will remain silent, unless you try to use one of the classes. In that case, you will get a compile-time error and have to explicitly name the class specifying its package.

It must be emphasized that the **import** statement is optional. Any place you use a class name, you can use its *fully qualified name*, which includes its full package hierarchy. For example, this fragment uses an import statement:

```
import java.util.*;
class MyDate extends Date {
}
```

The same example without the **import** statement looks like this:

```
class MyDate extends java.util.Date {
}
```

In this version, **Date** is fully-qualified.

As shown in Table 9-1, when a package is imported, only those items within the package declared as **public** will be available to non-subclasses in the importing code. For example, if you want the **Balance** class of the package **MyPack** shown earlier to be available as a stand-alone class for general use outside of **MyPack**, then you will need to declare it as **public** and put it into its own file, as shown here:

```
package MyPack;

/* Now, the Balance class, its constructor, and its
   show() method are public. This means that they can
   be used by non-subclass code outside their package.
*/
public class Balance {
  String name;
  double bal;

  public Balance(String n, double b) {
    name = n;
    bal = b;
  }

  public void show() {
    if(bal<0)
      System.out.print("--> ");
    System.out.println(name + ": $" + bal);
  }
}
```

As you can see, the **Balance** class is now **public**. Also, its constructor and its **show( )** method are **public**, too. This means that they can be accessed by any type of code outside the **MyPack** package. For example, here **TestBalance** imports **MyPack** and is then able to make use of the **Balance** class:

```
import MyPack.*;

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

```
    /* Because Balance is public, you may use Balance
       class and call its constructor. */
    Balance test = new Balance("J. J. Jaspers", 99.88);

    test.show(); // you may also call show()
  }
}
```

As an experiment, remove the **public** specifier from the **Balance** class and then try compiling **TestBalance**. As explained, errors will result.

Interfaces

Using the keyword **interface**, you can fully abstract a class' interface from its implementation. That is, using **interface**, you can specify what a class must do, but not how it does it. Interfaces are syntactically similar to classes, but they lack instance variables, and, as a general rule, their methods are declared without any body. In practice, this means that you can define interfaces that don't make assumptions about how they are implemented. Once it is defined, any number of classes can implement an **interface**. Also, one class can implement any number of interfaces.

To implement an interface, a class must provide the complete set of methods required by the interface. However, each class is free to determine the details of its own implementation. By providing the **interface** keyword, Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism.

Interfaces are designed to support dynamic method resolution at run time. Normally, in order for a method to be called from one class to another, both classes need to be present at compile time so the Java compiler can check to ensure that the method signatures are compatible. This requirement by itself makes for a static and nonextensible classing environment. Inevitably in a system like this, functionality gets pushed up higher and higher in the class hierarchy so that the mechanisms will be available to more and more subclasses. Interfaces are designed to avoid this problem. They disconnect the definition of a method or set of methods from the inheritance hierarchy. Since interfaces are in a different hierarchy from classes, it is possible for classes that are unrelated in terms of the class hierarchy to implement the same interface. This is where the real power of interfaces is realized.

Defining an Interface

An interface is defined much like a class. This is a simplified general form of an interface:

access interface *name* {
  return-type method-name1(*parameter-list*);
  return-type method-name2(*parameter-list*);

  type final-varname1 = *value*;
  type final-varname2 = *value*;
  //...
  return-type method-nameN(*parameter-list*);
  type final-varnameN = *value*;
}

When no access modifier is included, then default access results, and the interface is only available to other members of the package in which it is declared. When it is declared as **public**, the interface can be used by any other code. In this case, the interface must be the only public interface declared in the file, and the file must have the same name as the interface. *name* is the name of the interface, and can be any valid identifier. Notice that the methods that are declared have no bodies. They end with a semicolon after the parameter list. They are, essentially, abstract methods. Each class that includes such an interface must implement all of the methods.

Before continuing an important point needs to be made. JDK 8 added a feature to **interface** that makes a significant change to its capabilities. Prior to JDK 8, an interface could not define any implementation whatsoever. This is the type of interface that the preceding simplified form shows, in which no method declaration supplies a body. Thus, prior to JDK 8, an interface could define only "what," but not "how." JDK 8 changes this. Beginning with JDK 8, it is possible to add a *default implementation* to an interface method. Thus, it is now possible for **interface** to specify some behavior. However, default methods constitute what is, in essence, a special-use feature, and the original intent behind **interface** still remains. Therefore, as a general rule, you will still often create and use interfaces in which no default methods exist. For this reason, we will begin by discussing the interface in its traditional form. The default method is described at the end of this chapter.

As the general form shows, variables can be declared inside of interface declarations. They are implicitly **final** and **static**, meaning they cannot be changed by the implementing class. They must also be initialized. All methods and variables are implicitly **public**.

Here is an example of an interface definition. It declares a simple interface that contains one method called **callback( )** that takes a single integer parameter.

```
interface Callback {
  void callback(int param);
}
```

Implementing Interfaces

Once an **interface** has been defined, one or more classes can implement that interface. To implement an interface, include the **implements** clause in a class definition, and then create the methods required by the interface. The general form of a class that includes the **implements** clause looks like this:

class *classname* [extends *superclass*] [implements *interface* [,*interface...*]] {
  // class-body
}

If a class implements more than one interface, the interfaces are separated with a comma. If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface. The methods that implement an interface must be declared **public**. Also, the type signature of the implementing method must match exactly the type signature specified in the **interface** definition.

Here is a small example class that implements the **Callback** interface shown earlier:

```
class Client implements Callback {
  // Implement Callback's interface
```

```
  public void callback(int p) {

    System.out.println("callback called with " + p);
  }
}
```

Notice that **callback( )** is declared using the **public** access modifier.

REMEMBER When you implement an interface method, it must be declared as **public**.

It is both permissible and common for classes that implement interfaces to define additional members of their own. For example, the following version of **Client** implements **callback( )** and adds the method **nonIfaceMeth( )**:

```
class Client implements Callback {
  // Implement Callback's interface
  public void callback(int p) {
    System.out.println("callback called with " + p);
  }

  void nonIfaceMeth() {
    System.out.println("Classes that implement interfaces " +
                       "may also define other members, too.");
  }
}
```

Accessing Implementations Through Interface References

You can declare variables as object references that use an interface rather than a class type. Any instance of any class that implements the declared interface can be referred to by such a variable. When you call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to. This is one of the key features of interfaces. The method to be executed is looked up dynamically at run time, allowing classes to be created later than the code which calls methods on them. The calling code can dispatch through an interface without having to know anything about the "callee." This process is similar to using a superclass reference to access a subclass object, as described in Chapter 8.

CAUTION Because dynamic lookup of a method at run time incurs a significant overhead when compared with the normal method invocation in Java, you should be careful not to use interfaces casually in performance-critical code.

The following example calls the **callback( )** method via an interface reference variable:

```
class TestIface {
  public static void main(String args[]) {
    Callback c = new Client();
    c.callback(42);
  }
}
```

The output of this program is shown here:

```
callback called with 42
```

Notice that variable **c** is declared to be of the interface type **Callback**, yet it was assigned an instance of **Client**. Although **c** can be used to access the **callback( )** method, it cannot access any other members of the **Client** class. An interface reference variable has knowledge only of the methods declared by its **interface** declaration. Thus, **c** could not be used to access **nonIfaceMeth( )** since it is defined by **Client** but not **Callback**.

While the preceding example shows, mechanically, how an interface reference variable can access an implementation object, it does not demonstrate the polymorphic power of such a reference. To sample this usage, first create the second implementation of **Callback**, shown here:

```
// Another implementation of Callback.
class AnotherClient implements Callback {
  // Implement Callback's interface
  public void callback(int p) {
    System.out.println("Another version of callback");
    System.out.println("p squared is " + (p*p));
  }
}
```

Now, try the following class:

```
class TestIface2 {
  public static void main(String args[]) {
    Callback c = new Client();
    AnotherClient ob = new AnotherClient();

    c.callback(42);

    c = ob; // c now refers to AnotherClient object
    c.callback(42);
  }
}
```

The output from this program is shown here:

```
callback called with 42
Another version of callback
p squared is 1764
```

As you can see, the version of **callback( )** that is called is determined by the type of object that **c** refers to at run time. While this is a very simple example, you will see another, more practical one shortly.

Partial Implementations

If a class includes an interface but does not fully implement the methods required by that interface, then that class must be declared as **abstract**. For example:

```
abstract class Incomplete implements Callback {
  int a, b;
```

```
  void show() {
    System.out.println(a + " " + b);
  }
  //...
}
```

Here, the class **Incomplete** does not implement **callback( )** and must be declared as **abstract**. Any class that inherits **Incomplete** must implement **callback( )** or be declared **abstract** itself.

Nested Interfaces

An interface can be declared a member of a class or another interface. Such an interface is called a *member interface* or a *nested interface*. A nested interface can be declared as **public**, **private**, or **protected**. This differs from a top-level interface, which must either be declared as **public** or use the default access level, as previously described. When a nested interface is used outside of its enclosing scope, it must be qualified by the name of the class or interface of which it is a member. Thus, outside of the class or interface in which a nested interface is declared, its name must be fully qualified.

Here is an example that demonstrates a nested interface:

```
// A nested interface example.

// This class contains a member interface.
class A {
  // this is a nested interface
  public interface NestedIF {
    boolean isNotNegative(int x);
  }
}

// B implements the nested interface.
class B implements A.NestedIF {
  public boolean isNotNegative(int x) {
    return x < 0 ? false: true;
  }
}

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

    // use a nested interface reference
    A.NestedIF nif = new B();

    if(nif.isNotNegative(10))
      System.out.println("10 is not negative");
    if(nif.isNotNegative(-12))
      System.out.println("this won't be displayed");
  }
}
```

Notice that **A** defines a member interface called **NestedIF** and that it is declared **public**. Next, **B** implements the nested interface by specifying

```
implements A.NestedIF
```

Notice that the name is fully qualified by the enclosing class' name. Inside the **main( )** method, an **A.NestedIF** reference called **nif** is created, and it is assigned a reference to a **B** object. Because **B** implements **A.NestedIF**, this is legal.

Applying Interfaces

To understand the power of interfaces, let's look at a more practical example. In earlier chapters, you developed a class called **Stack** that implemented a simple fixed-size stack. However, there are many ways to implement a stack. For example, the stack can be of a fixed size or it can be "growable." The stack can also be held in an array, a linked list, a binary tree, and so on. No matter how the stack is implemented, the interface to the stack remains the same. That is, the methods **push( )** and **pop( )** define the interface to the stack independently of the details of the implementation. Because the interface to a stack is separate from its implementation, it is easy to define a stack interface, leaving it to each implementation to define the specifics. Let's look at two examples.

First, here is the interface that defines an integer stack. Put this in a file called **IntStack.java**. This interface will be used by both stack implementations.

```
// Define an integer stack interface.
interface IntStack {
  void push(int item); // store an item
  int pop(); // retrieve an item
}
```

The following program creates a class called **FixedStack** that implements a fixed-length version of an integer stack:

```
// An implementation of IntStack that uses fixed storage.
class FixedStack implements IntStack {
  private int stck[];
  private int tos;

  // allocate and initialize stack
  FixedStack(int size) {
    stck = new int[size];
    tos = -1;
  }

  // Push an item onto the stack
  public void push(int item) {
    if(tos==stck.length-1) // use length member
      System.out.println("Stack is full.");
    else
      stck[++tos] = item;
  }
```

```
  // Pop an item from the stack
  public int pop() {
    if(tos < 0) {
      System.out.println("Stack underflow.");
      return 0;
    }
    else
      return stck[tos--];
  }
}

class IFTest {
  public static void main(String args[]) {
    FixedStack mystack1 = new FixedStack(5);
    FixedStack mystack2 = new FixedStack(8);

    // push some numbers onto the stack
    for(int i=0; i<5; i++) mystack1.push(i);
    for(int i=0; i<8; i++) mystack2.push(i);

    // pop those numbers off the stack
    System.out.println("Stack in mystack1:");
    for(int i=0; i<5; i++)
       System.out.println(mystack1.pop());

    System.out.println("Stack in mystack2:");
    for(int i=0; i<8; i++)
       System.out.println(mystack2.pop());
  }
}
```

Following is another implementation of **IntStack** that creates a dynamic stack by use of the same **interface** definition. In this implementation, each stack is constructed with an initial length. If this initial length is exceeded, then the stack is increased in size. Each time more room is needed, the size of the stack is doubled.

```
// Implement a "growable" stack.
class DynStack implements IntStack {
  private int stck[];
  private int tos;

  // allocate and initialize stack
  DynStack(int size) {
    stck = new int[size];
    tos = -1;
  }

  // Push an item onto the stack
  public void push(int item) {
    // if stack is full, allocate a larger stack
    if(tos==stck.length-1) {
      int temp[] = new int[stck.length * 2]; // double size
      for(int i=0; i<stck.length; i++) temp[i] = stck[i];
```

```
      stck = temp;
      stck[++tos] = item;
    }
    else
      stck[++tos] = item;
  }

  // Pop an item from the stack
  public int pop() {
    if(tos < 0) {
      System.out.println("Stack underflow.");
      return 0;
    }
    else
      return stck[tos--];
  }
}

class IFTest2 {
  public static void main(String args[]) {
    DynStack mystack1 = new DynStack(5);
    DynStack mystack2 = new DynStack(8);

    // these loops cause each stack to grow
    for(int i=0; i<12; i++) mystack1.push(i);
    for(int i=0; i<20; i++) mystack2.push(i);

    System.out.println("Stack in mystack1:");
    for(int i=0; i<12; i++)
       System.out.println(mystack1.pop());

    System.out.println("Stack in mystack2:");
    for(int i=0; i<20; i++)
       System.out.println(mystack2.pop());
  }
}
```

The following class uses both the **FixedStack** and **DynStack** implementations. It does so through an interface reference. This means that calls to **push( )** and **pop( )** are resolved at run time rather than at compile time.

```
/* Create an interface variable and
   access stacks through it.
*/
class IFTest3 {
  public static void main(String args[]) {
    IntStack mystack; // create an interface reference variable
    DynStack ds = new DynStack(5);
    FixedStack fs = new FixedStack(8);

    mystack = ds; // load dynamic stack
    // push some numbers onto the stack
    for(int i=0; i<12; i++) mystack.push(i);
```

```
    mystack = fs; // load fixed stack
    for(int i=0; i<8; i++) mystack.push(i);

    mystack = ds;
    System.out.println("Values in dynamic stack:");
    for(int i=0; i<12; i++)
       System.out.println(mystack.pop());

    mystack = fs;
    System.out.println("Values in fixed stack:");
    for(int i=0; i<8; i++)
       System.out.println(mystack.pop());
  }
}
```

In this program, **mystack** is a reference to the **IntStack** interface. Thus, when it refers to **ds**, it uses the versions of **push( )** and **pop( )** defined by the **DynStack** implementation. When it refers to **fs**, it uses the versions of **push( )** and **pop( )** defined by **FixedStack**. As explained, these determinations are made at run time. Accessing multiple implementations of an interface through an interface reference variable is the most powerful way that Java achieves run-time polymorphism.

Variables in Interfaces

You can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables that are initialized to the desired values. When you include that interface in a class (that is, when you "implement" the interface), all of those variable names will be in scope as constants. (This is similar to using a header file in C/C++ to create a large number of **#defined** constants or **const** declarations.) If an interface contains no methods, then any class that includes such an interface doesn't actually implement anything. It is as if that class were importing the constant fields into the class name space as **final** variables. The next example uses this technique to implement an automated "decision maker":

```
import java.util.Random;

interface SharedConstants {
  int NO = 0;
  int YES = 1;
  int MAYBE = 2;
  int LATER = 3;
  int SOON = 4;
  int NEVER = 5;
}

class Question implements SharedConstants {
  Random rand = new Random();
  int ask() {
    int prob = (int) (100 * rand.nextDouble());
    if (prob < 30)
```

```
      return NO;           // 30%
    else if (prob < 60)
      return YES;          // 30%
    else if (prob < 75)
      return LATER;        // 15%
    else if (prob < 98)
      return SOON;         // 13%

    else
      return NEVER;        // 2%
  }
}

class AskMe implements SharedConstants {
  static void answer(int result) {
    switch(result) {
      case NO:
        System.out.println("No");
        break;
      case YES:
        System.out.println("Yes");
        break;
      case MAYBE:
        System.out.println("Maybe");
        break;
      case LATER:
        System.out.println("Later");
        break;
      case SOON:
        System.out.println("Soon");
        break;
      case NEVER:
        System.out.println("Never");
        break;
    }
  }

  public static void main(String args[]) {
    Question q = new Question();

    answer(q.ask());
    answer(q.ask());
    answer(q.ask());
    answer(q.ask());
  }
}
```

Notice that this program makes use of one of Java's standard classes: **Random**. This class provides pseudorandom numbers. It contains several methods that allow you to obtain random numbers in the form required by your program. In this example, the method **nextDouble( )** is used. It returns random numbers in the range 0.0 to 1.0.

In this sample program, the two classes, **Question** and **AskMe**, both implement the **SharedConstants** interface where **NO**, **YES**, **MAYBE**, **SOON**, **LATER**, and **NEVER** are defined. Inside each class, the code refers to these constants as if each class had defined or inherited them directly. Here is the output of a sample run of this program. Note that the results are different each time it is run.

```
Later
Soon
No
Yes
```

NOTE The technique of using an interface to define shared constants, as just described, is controversial. It is described here for completeness.

Interfaces Can Be Extended

One interface can inherit another by use of the keyword **extends**. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all methods required by the interface inheritance chain. Following is an example:

```
// One interface can extend another.
interface A {
  void meth1();
  void meth2();
}

// B now includes meth1() and meth2() -- it adds meth3().
interface B extends A {
  void meth3();
}

// This class must implement all of A and B
class MyClass implements B {
  public void meth1() {
    System.out.println("Implement meth1().");
  }

  public void meth2() {
    System.out.println("Implement meth2().");
  }

  public void meth3() {
    System.out.println("Implement meth3().");
  }
}

class IFExtend {
  public static void main(String arg[]) {
    MyClass ob = new MyClass();
```

```
    ob.meth1();
    ob.meth2();
    ob.meth3();
  }
}
```

As an experiment, you might want to try removing the implementation for **meth1( )** in **MyClass**. This will cause a compile-time error. As stated earlier, any class that implements an interface must implement all methods required by that interface, including any that are inherited from other interfaces.

Default Interface Methods

As explained earlier, prior to JDK 8, an interface could not define any implementation whatsoever. This meant that for all previous versions of Java, the methods specified by an interface were abstract, containing no body. This is the traditional form of an interface and is the type of interface that the preceding discussions have used. The release of JDK 8 has changed this by adding a new capability to **interface** called the *default method*. A default method lets you define a default implementation for an interface method. In other words, by use of a default method, it is now possible for an interface method to provide a body, rather than being abstract. During its development, the default method was also referred to as an *extension method,* and you will likely see both terms used.

A primary motivation for the default method was to provide a means by which interfaces could be expanded without breaking existing code. Recall that there must be implementations for all methods defined by an interface. In the past, if a new method were added to a popular, widely used interface, then the addition of that method would break existing code because no implementation would be found for that new method. The default method solves this problem by supplying an implementation that will be used if no other implementation is explicitly provided. Thus, the addition of a default method will not cause preexisting code to break.

Another motivation for the default method was the desire to specify methods in an interface that are, essentially, optional, depending on how the interface is used. For example, an interface might define a group of methods that act on a sequence of elements. One of these methods might be called **remove( )**, and its purpose is to remove an element from the sequence. However, if the interface is intended to support both modifiable and nonmodifiable sequences, then **remove( )** is essentially optional because it won't be used by nonmodifiable sequences. In the past, a class that implemented a nonmodifiable sequence would have had to define an empty implementation of **remove( )**, even though it was not needed. Today, a default implementation for **remove( )** can be specified in the interface that does nothing (or throws an exception). Providing this default prevents a class used for nonmodifiable sequences from having to define its own, placeholder version of **remove( )**. Thus, by providing a default, the interface makes the implementation of **remove( )** by a class optional.

It is important to point out that the addition of default methods does not change a key aspect of **interface**: its inability to maintain state information. An interface still cannot have instance variables, for example. Thus, the defining difference between an interface and a class is that a class can maintain state information, but an interface cannot. Furthermore, it is still not possible to create an instance of an interface by itself. It must be implemented by a class. Therefore, even though, beginning with JDK 8, an interface can define default methods, the interface must still be implemented by a class if an instance is to be created.

One last point: As a general rule, default methods constitute a special-purpose feature. Interfaces that you create will still be used primarily to specify *what* and not *how*. However, the inclusion of the default method gives you added flexibility.

Default Method Fundamentals

An interface default method is defined similar to the way a method is defined by a **class**. The primary difference is that the declaration is preceded by the keyword **default**. For example, consider this simple interface:

```
public interface MyIF {
  // This is a "normal" interface method declaration.
  // It does NOT define a default implementation.
  int getNumber();

  // This is a default method. Notice that it provides
  // a default implementation.
  default String getString() {
    return "Default String";
  }
}
```

MyIF declares two methods. The first, **getNumber( )**, is a standard interface method declaration. It defines no implementation whatsoever. The second method is **getString( )**, and it does include a default implementation. In this case, it simply returns the string "Default String". Pay special attention to the way **getString( )** is declared. Its declaration is preceded by the **default** modifier. This syntax can be generalized. To define a default method, precede its declaration with **default**.

Because **getString( )** includes a default implementation, it is not necessary for an implementing class to override it. In other words, if an implementing class does not provide its own implementation, the default is used. For example, the **MyIFImp** class shown next is perfectly valid:

```
// Implement MyIF.
class MyIFImp implements MyIF {
  // Only getNumber() defined by MyIF needs to be implemented.
  // getString() can be allowed to default.
  public int getNumber() {
    return 100;
  }
}
```

The following code creates an instance of **MyIFImp** and uses it to call both **getNumber( )** and **getString( )**.

```
// Use the default method.
class DefaultMethodDemo {
```

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

    MyIFImp obj = new MyIFImp();

    // Can call getNumber(), because it is explicitly
    // implemented by MyIFImp:
    System.out.println(obj.getNumber());

    // Can also call getString(), because of default
    // implementation:
    System.out.println(obj.getString());
  }
}
```

The output is shown here:

```
100
Default String
```

As you can see, the default implementation of **getString( )** was automatically used. It was not necessary for **MyIFImp** to define it. Thus, for **getString( )**, implementation by a class is optional. (Of course, its implementation by a class will be *required* if the class uses **getString( )** for some purpose beyond that supported by its default.)

It is both possible and common for an implementing class to define its own implementation of a default method. For example, **MyIFImp2** overrides **getString( )**:

```
class MyIFImp2 implements MyIF {
  // Here, implementations for both getNumber( ) and getString( ) are provided.
  public int getNumber() {
    return 100;
  }

  public String getString() {
    return "This is a different string.";
  }
}
```

Now, when **getString( )** is called, a different string is returned.

A More Practical Example

Although the preceding shows the mechanics of using default methods, it doesn't illustrate their usefulness in a more practical setting. To do this, let's once again return to the **IntStack** interface shown earlier in this chapter. For the sake of discussion, assume that **IntStack** is widely used and many programs rely on it. Further assume that we now want to add a method to **IntStack** that clears the stack, enabling the stack to be re-used. Thus, we want to evolve the **IntStack** interface so that it defines new functionality, but we don't want to break any preexisting code. In the past, this would be impossible, but with the inclusion of default methods, it is now easy to do. For example, the **IntStack** interface can be enhanced like this:

```
interface IntStack {
  void push(int item); // store an item
  int pop(); // retrieve an item

  // Because clear( ) has a default, it need not be
  // implemented by a preexisting class that uses IntStack.
  default void clear() {
    System.out.println("clear() not implemented.");
  }
}
```

Here, the default behavior of **clear( )** simply displays a message indicating that it is not implemented. This is acceptable because no preexisting class that implements **IntStack** would ever call **clear( )** because it was not defined by the earlier version of **IntStack**. However, **clear( )** can be implemented by a new class that implements **IntStack**. Furthermore, **clear( )** needs to be defined by a new implementation only if it is used. Thus, the default method gives you

- a way to gracefully evolve interfaces over time, and
- a way to provide optional functionality without requiring that a class provide a placeholder implementation when that functionality is not needed.

One other point: In real-world code, **clear( )** would have thrown an exception, rather than displaying an error message. Exceptions are described in the next chapter. After working through that material, you might want to try modifying **clear( )** so that its default implementation throws an **UnsupportedOperationException**.

Multiple Inheritance Issues

As explained earlier in this book, Java does not support the multiple inheritance of classes. Now that an interface can include default methods, you might be wondering if an interface can provide a way around this restriction. The answer is, essentially, no. Recall that there is still a key difference between a class and an interface: a class can maintain state information (especially through the use of instance variables), but an interface cannot.

The preceding notwithstanding, default methods do offer a bit of what one would normally associate with the concept of multiple inheritance. For example, you might have a class that implements two interfaces. If each of these interfaces provides default methods, then some behavior is inherited from both. Thus, to a limited extent, default methods do support multiple inheritance of behavior. As you might guess, in such a situation, it is possible that a name conflict will occur.

For example, assume that two interfaces called **Alpha** and **Beta** are implemented by a class called **MyClass**. What happens if both **Alpha** and **Beta** provide a method called **reset( )** for which both declare a default implementation? Is the version by **Alpha** or the version by **Beta** used by **MyClass**? Or, consider a situation in which **Beta** extends **Alpha**. Which version of the default method is used? Or, what if **MyClass** provides its own implementation of the method? To handle these and other similar types of situations, Java defines a set of rules that resolves such conflicts.

First, in all cases, a class implementation takes priority over an interface default implementation. Thus, if **MyClass** provides an override of the **reset( )** default method, **MyClass**' version is used. This is the case even if **MyClass** implements both **Alpha** and **Beta**. In this case, both defaults are overridden by **MyClass**' implementation.

Second, in cases in which a class implements two interfaces that both have the same default method, but the class does not override that method, then an error will result. Continuing with the example, if **MyClass** implements both **Alpha** and **Beta**, but does not override **reset( )**, then an error will occur.

In cases in which one interface inherits another, with both defining a common default method, the inheriting interface's version of the method takes precedence. Therefore, continuing the example, if **Beta** extends **Alpha**, then **Beta**'s version of **reset( )** will be used.

It is possible to explicitly refer to a default implementation in an inherited interface by using a new form of **super**. Its general form is shown here:

InterfaceName.super.*methodName( )*

For example, if **Beta** wants to refer to **Alpha**'s default for **reset( )**, it can use this statement:

```
Alpha.super.reset();
```

Use static Methods in an Interface

JDK 8 added another new capability to **interface**: the ability to define one or more **static** methods. Like **static** methods in a class, a **static** method defined by an interface can be called independently of any object. Thus, no implementation of the interface is necessary, and no instance of the interface is required, in order to call a **static** method. Instead, a **static** method is called by specifying the interface name, followed by a period, followed by the method name. Here is the general form:

InterfaceName.staticMethodName

Notice that this is similar to the way that a **static** method in a class is called.

The following shows an example of a **static** method in an interface by adding one to **MyIF**, shown in the previous section. The **static** method is **getDefaultNumber( )**. It returns zero.

```
public interface MyIF {
  // This is a "normal" interface method declaration.
  // It does NOT define a default implementation.
  int getNumber();

  // This is a default method. Notice that it provides
  // a default implementation.
  default String getString() {
    return "Default String";
  }
```

```
  // This is a static interface method.
  static int getDefaultNumber() {
    return 0;
  }
}
```

The **getDefaultNumber( )** method can be called, as shown here:

```
int defNum = MyIF.getDefaultNumber();
```

As mentioned, no implementation or instance of **MyIF** is required to call **getDefaultNumber( )** because it is **static**.

One last point: **static** interface methods are not inherited by either an implementing class or a subinterface.

Final Thoughts on Packages and Interfaces

Although the examples we've included in this book do not make frequent use of packages or interfaces, both of these tools are an important part of the Java programming environment. Virtually all real programs that you write in Java will be contained within packages. A number will probably implement interfaces as well. It is important, therefore, that you be comfortable with their usage.