ORACLE

Fully updated for Java SE 8 (JDK 8)

Java
A Beginner's Guide
Sixth Edition

Create, Compile, and Run Java Programs Today

Herbert Schildt

Oracle Press™

Java™

A Beginner's Guide

Sixth Edition

About the Author

Best-selling author **Herbert Schildt** has written extensively about programming for nearly three decades and is a leading authority on the Java language. His books have sold millions of copies worldwide and have been translated into all major foreign languages. He is the author of numerous books on Java, including *Java: The Complete Reference, Herb Schildt's Java Programming Cookbook,* and *Swing: A Beginner's Guide*. He has also written extensively about C, C++, and C#. Although interested in all facets of computing, his primary focus is computer languages, including compilers, interpreters, and robotic control languages. He also has an active interest in the standardization of languages. Schildt holds both graduate and undergraduate degrees from the University of Illinois. He can be reached at his consulting office at (217) 586-4683. His website is **www.HerbSchildt.com**.

About the Technical Reviewer

Dr. Danny Coward has worked on all editions of the Java platform. He led the definition of Java Servlets into the first version of the Java EE platform and beyond, web services into the Java ME platform, and the strategy and planning for Java SE 7. He founded JavaFX technology and, most recently, designed the largest addition to the Java EE 7 standard, the Java WebSocket API. From coding in Java, to designing APIs with industry experts, to serving for several years as an executive to the Java Community Process, he has a uniquely broad perspective into multiple aspects of Java technology. Additionally, he is the author of *JavaWebSocket Programming* and an upcoming book on Java EE. Dr. Coward holds a bachelor's, master's, and doctorate in mathematics from the University of Oxford.

Java™
A Beginner's Guide

Sixth Edition

Herbert Schildt

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

ISBN: 978-0-07-180926-9

MHID: 0-07-180926-0

e-book conversion by Cenveo® Publisher Services

Version 1.0

The material in this e-book also appears in the print version of this title: ISBN: 978-0-07-180925-2, MHID: 0-07-180925-2

Contents at a Glance

Contents

Introduction

The purpose of this book is to teach you the fundamentals of Java programming. It uses a step-by-step approach complete with numerous examples, self tests, and projects. It assumes no previous programming experience. The book starts with the basics, such as how to compile and run a Java program. It then discusses the keywords, features, and constructs that form the core of the Java language. You'll also find coverage of some of Java's most advanced features, including multithreaded programming and generics. An introduction to the fundamentals of Swing and JavaFX concludes the book. By the time you finish, you will have a firm grasp of the essentials of Java programming.

It is important to state at the outset that this book is just a starting point. Java is more than just the elements that define the language. Java also includes extensive libraries and tools that aid in the development of programs. To be a top-notch Java programmer implies mastery of these areas, too. After completing this book, you will have the knowledge to pursue any and all other aspects of Java.

The Evolution of Java

Only a few languages have fundamentally reshaped the very essence of programming. In this elite group, one stands out because its impact was both rapid and widespread. This language is, of course, Java. It is not an overstatement to say that the original release of Java 1.0 in 1995 by Sun Microsystems, Inc., caused a revolution in programming. This revolution radically transformed the Web into a highly interactive environment. In the process, Java set a new standard in computer language design.

Over the years, Java has continued to grow, evolve, and otherwise redefine itself. Unlike many other languages, which are slow to incorporate new features, Java has often been at the forefront of computer language development. One reason for this is the culture of innovation and change that came to surround Java. As a result, Java has gone through several upgrades—some relatively small, others more significant.

The first major update to Java was version 1.1. The features added by Java 1.1 were more substantial than the increase in the minor revision number would have you think. For example, Java 1.1 added many new library elements, redefined the way events are handled, and reconfigured many features of the original 1.0 library.

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, itself. 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.

The next upgrade of Java was J2SE 1.3. This version of Java 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. The release of J2SE 1.4 further enhanced Java. This release contained several important new features, including chained exceptions, channel-based I/O, and the **assert** keyword.

The release of J2SE 5 created nothing short of a second Java revolution. Unlike most of the previous Java upgrades, which offered important but incremental improvements, J2SE 5 fundamentally expanded the scope, power, and range of the language. To give you an idea of the magnitude of the changes caused by J2SE 5, here is a list of its major new features:

- Generics
- Autoboxing/unboxing
- Enumerations
- The enhanced "for-each" style **for** loop
- Variable-length arguments (varargs)
- Static import
- Annotations

This is not a list of minor tweaks or incremental upgrades. Each item in the list represents a significant addition to the Java language. Some, such as generics, the enhanced **for** loop, 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.

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 Java Development Kit (JDK) was called JDK 5. In order to maintain consistency, however, 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, and Sun once again decided to change the name of the Java platform. First, notice that the "2" has been dropped. Thus, the platform now had the name *Java SE,* and the official product name was *Java Platform, Standard Edition 6,* with the development kit being called JDK 6. As with J2SE 5, the 6 in Java SE 6 is the product version number. The internal, developer version number is 1.6.

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

The next release of Java was called Java SE 7, with the development kit being called JDK 7. It has an internal version number of 1.7. Java SE 7 was the first major release of Java after Sun Microsystems was acquired by Oracle. Java SE 7 added several new features, including significant additions to the language and the API libraries. Some of the most important features added by Java SE 7 were those 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, including

- A **String** can control a **switch** statement.
- Binary integer literals.
- Underscores in numeric literals.
- An expanded **try** statement, called **try**-*with-resources,* that supports automatic resource management.
- 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** (multicatch) and better type checking for exceptions that are rethrown.

As you can see, even though the Project Coin features were considered to be small changes to the language, their benefits were much larger than the qualifier "small" would suggest. In particular, the **try**-with-resources statement profoundly affects the way that a substantial amount of code is written.

Java SE 8

The newest release of Java is Java SE 8, with the development 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. In the process, lambda expressions can simplify and reduce the amount of source code needed to create certain constructs. 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.

In addition to lambda expressions, JDK 8 adds many other important new features. For example, beginning with JDK 8, it is now possible to define a default implementation for a method specified by an interface. JDK 8 also bundles support for JavaFX, Java's new GUI framework. JavaFX is expected to soon play an important part in nearly all Java applications, ultimately replacing Swing for most GUI-based projects. 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. The material in this book has been updated to reflect Java SE 8, with many new features, updates, and additions indicated throughout.

How This Book Is Organized

This book presents an evenly paced tutorial in which each section builds upon the previous one. It contains 17 chapters, each discussing an aspect of Java. This book is unique because it includes several special elements that reinforce what you are learning.

Key Skills & Concepts

Each chapter begins with a set of critical skills that you will be learning.

Self Test

Each chapter concludes with a Self Test that lets you test your knowledge. The answers are in Appendix A.

Ask the Expert

Sprinkled throughout the book are special "Ask the Expert" boxes. These contain additional information or interesting commentary about a topic. They use a Question/Answer format.

Try This Elements

Each chapter contains one or more Try This elements, which are projects that show you how to apply what you are learning. In many cases, these are real-world examples that you can use as starting points for your own programs.

No Previous Programming Experience Required

This book assumes no previous programming experience. Thus, if you have never programmed before, you can use this book. If you do have some previous programming experience, you will be able to advance a bit more quickly. Keep in mind, however, that Java differs in several key ways from other popular computer languages. It is important not to jump to conclusions. Thus, even for the experienced programmer, a careful reading is advised.

Required Software

To compile and run all of the programs in this book, you will need the latest Java Development Kit (JDK) from Oracle, which, at the time of this writing, is JDK 8. This is the JDK for Java SE 8. Instructions for obtaining the Java JDK are given in Chapter 1.

If you are using an earlier version of Java, you will still be able to use this book, but you won't be able to compile and run the programs that use Java's newer features.

Don't Forget: Code on the Web

Remember, the source code for all of the examples and projects in this book is available free of charge on the Web at **www.oraclepressbooks.com**.

Special Thanks

Special thanks to Danny Coward, the technical editor for this edition of the book. Danny has worked on several of my books and his advice, insights, and suggestions have always been of great value and much appreciated.

For Further Study

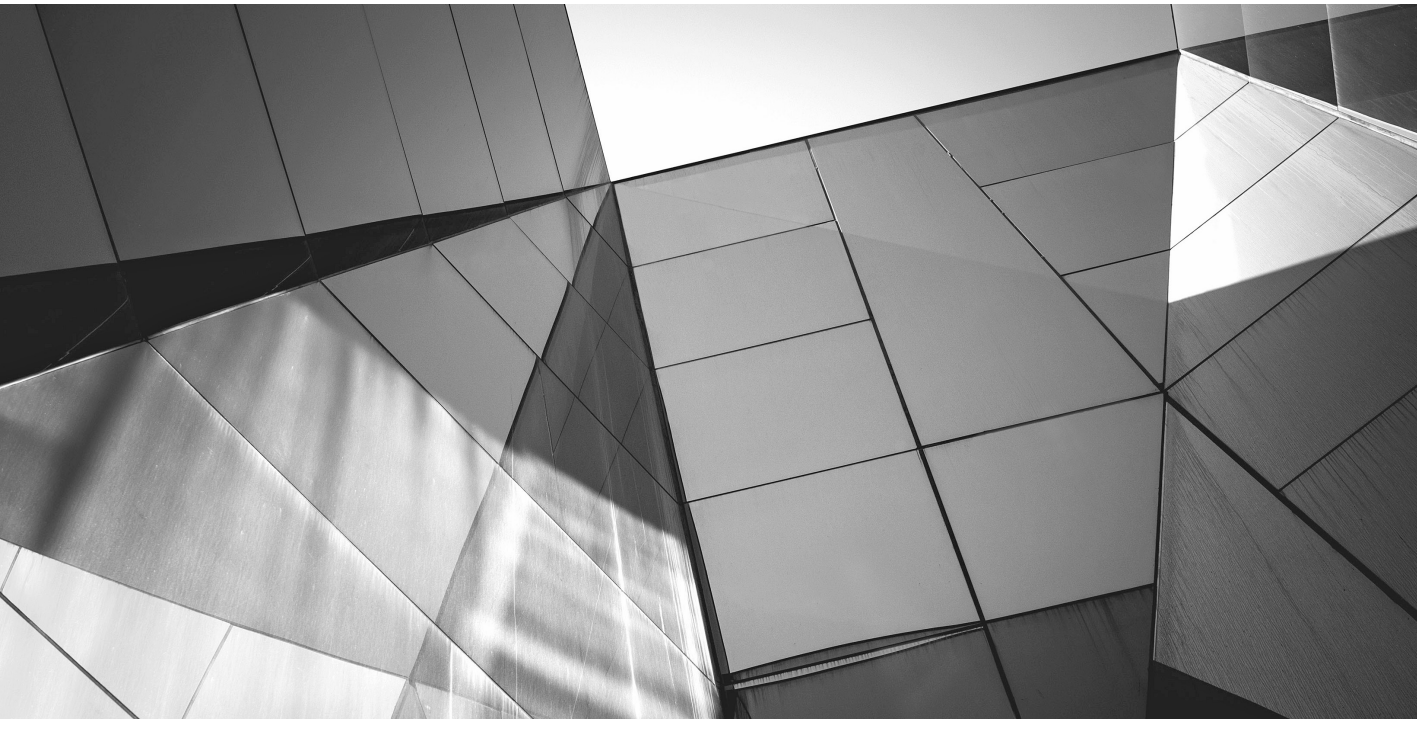
Java: A Beginner's Guide is your gateway to the Herb Schildt series of Java programming books. Here are some others that you will find of interest:

Java: The Complete Reference

Herb Schildt's Java Programming Cookbook

The Art of Java

Swing: A Beginner's Guide



Chapter 1

Java Fundamentals

Key Skills & Concepts

- Know the history and philosophy of Java
- Understand Java's contribution to the Internet
- Understand the importance of bytecode
- Know the Java buzzwords
- Understand the foundational principles of object-oriented programming
- Create, compile, and run a simple Java program
- Use variables
- Use the **if** and **for** control statements
- Create blocks of code
- Understand how statements are positioned, indented, and terminated
- Know the Java keywords
- Understand the rules for Java identifiers

The rise of the Internet and the World Wide Web fundamentally reshaped computing. Prior to the Web, the cyber landscape was dominated by stand-alone PCs. Today, nearly all computers are connected to the Internet. The Internet, itself, was transformed—originally offering a convenient way to share files and information. Today it is a vast, distributed computing universe. With these changes came a new way to program: Java.

Java is the preeminent language of the Internet, but it is more than that. Java revolutionized programming, changing the way that we think about both the form and the function of a program. To be a professional programmer today implies the ability to program in Java—it is that important. In the course of this book, you will learn the skills needed to master it.

The purpose of this chapter is to introduce you to Java, including its history, its design philosophy, and several of its most important features. By far, the hardest thing about learning a programming language is the fact that no element exists in isolation. Instead, the components of the language work in conjunction with each other. This interrelatedness is especially pronounced in Java. In fact, it is difficult to discuss one aspect of Java without involving others. To help overcome this problem, this chapter provides a brief overview of several Java features, including the general form of a Java program, some basic control structures, and operators. It does not go into too many details but, rather, concentrates on the general concepts common to any Java program.

The Origins of Java

Computer language innovation is driven forward by two factors: improvements in the art of programming and changes in the computing environment. Java is no exception. Building upon the rich legacy inherited from C and C++, Java adds refinements and features that reflect the current state of the art in programming. Responding to the rise of the online environment, Java offers features that streamline programming for a highly distributed architecture.

Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems in 1991. This language was initially called "Oak" but was renamed "Java" in 1995. Somewhat surprisingly, the original impetus for Java was not the Internet! Instead, the primary motivation was the need for a platform-independent language that could be used to create software to be embedded in various consumer electronic devices, such as toasters, microwave ovens, and remote controls. As you can probably guess, many different types of CPUs are used as controllers. The trouble was that (at that time) most computer languages were designed to be compiled for a specific target. For example, consider C++.

Although it was possible to compile a C++ program for just about any type of CPU, to do so required a full C++ compiler targeted for that CPU. The problem, however, is that compilers are expensive and time-consuming to create. In an attempt to find a better solution, Gosling and others worked on a portable, cross-platform language that could 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 emerged 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 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. 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 many types of computers, operating systems, and CPUs.

What was once an irritating but a 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. This realization caused the focus of Java to switch from consumer electronics to Internet programming. So, while it was the desire for an architecture-neutral programming language that provided the initial spark, it was the Internet that ultimately led to Java's large-scale success.

How Java Relates to C and C++

Java is directly related to both C and C++. Java inherits its syntax from C. Its object model is adapted from C++. Java's relationship with C and C++ is important for several reasons. First, many programmers are familiar with the C/C++ syntax. This makes it easy for a C/C++ programmer to learn Java and, conversely, for a Java programmer to learn C/C++.

Second, Java's designers did not "reinvent the wheel." Instead, they further refined an already highly successful programming paradigm. The modern age of programming began with C. It moved to C++, and now to Java. By inheriting and building upon that rich heritage, Java provides a powerful, logically consistent programming environment that takes the best of the past and adds new features required by the online environment. Perhaps most important, because of their similarities, C, C++, and Java define a common, conceptual framework for the professional programmer. Programmers do not face major rifts when switching from one language to another.

One of the central design philosophies of both C and C++ is that the programmer is in charge! Java also inherits this philosophy. 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.

Java has one other attribute in common with C and C++: it 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. There is no better way to produce a top-flight professional programming language.

Because of the similarities between Java and C++, especially their support for object-oriented programming, it is tempting to think of Java as simply the "Internet version of C++." However, to do so would be a mistake. Java has significant practical and philosophical differences. Although Java was influenced by C++, it is not an enhanced version of C++. For example, it is neither upwardly nor downwardly compatible with C++. Of course, the similarities with C++ are significant, and if you are a C++ programmer, you will feel right at home with Java. Another 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. They will coexist for many years to come.

How Java Relates to C#

A few years after the creation of Java, Microsoft developed the C# language. This is important because C# is closely related to Java. In fact, many of C#'s features directly parallel Java. Both Java and C# share the same general C++-style 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 means that if you already know C#, then learning Java will be especially easy. Conversely, if C# is in your future, then your knowledge of Java will come in handy.

Given the similarity between Java and C#, one might naturally ask, "Will C# replace Java?" The answer is No. Java and C# are optimized for two different types of computing environments. Just as C++ and Java will coexist for a long time to come, so will C# and Java.

Java's Contribution to 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 that 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 safely downloaded and executed on the client computer, 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 is considered by many to be 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 different 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 in *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 because many modern languages are designed to be compiled into executable code due to 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. Safety is also enhanced by certain restrictions that exist in the Java language.

When a program is interpreted, it generally runs slower than the same program 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

Ask the Expert

Q: I have heard about a special type of Java program called a servlet. What is it?

A: A *servlet* is a small program that executes on a server. Just as applets dynamically extend the functionality of a web browser, servlets dynamically extend the functionality of a web server. It is helpful to understand that 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. Thus, with the advent of the servlet, Java spanned both sides of the client/server connection. Although the creation of servlets is beyond the scope of this beginner's guide, they are something that you will want to study further as you advance in Java programming. (Coverage of servlets can be found in my book *Java: The Complete Reference,* published by Oracle Press/McGraw-Hill Education.)

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.

The Java Buzzwords

No overview of Java 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 played an important role in molding the final form of the language. The key considerations were summed up by the Java design team in the following list of buzzwords.

Simple	Java has a concise, cohesive set of features that makes it easy to learn and use.
Secure	Java provides a secure means of creating Internet applications.
Portable	Java programs can execute in any environment for which there is a Java run-time system.
Object-oriented	Java embodies the modern, object-oriented programming philosophy.
Robust	Java encourages error-free programming by being strictly typed and performing run-time checks.
Multithreaded	Java provides integrated support for multithreaded programming.
Architecture-neutral	Java is not tied to a specific machine or operating system architecture.
Interpreted	Java supports cross-platform code through the use of Java bytecode.
High performance	The Java bytecode is highly optimized for speed of execution.
Distributed	Java was designed with the distributed environment of the Internet in mind.
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.

Ask the Expert

Q: **To address the issues of portability and security, why was it necessary to create a new computer language such as Java; couldn't a language like C++ be adapted? In other words, couldn't a C++ compiler that outputs bytecode be created?**

A: While it would be possible for a C++ compiler to generate something similar to bytecode rather than executable code, C++ has features that discourage its use for the creation of Internet programs—the most important feature being C++'s support for pointers. A *pointer* is the address of some object stored in memory. Using a pointer, it would be possible to access resources outside the program itself, resulting in a security breach. Java does not support pointers, thus eliminating this problem.

Object-Oriented Programming

At the center of Java is object-oriented programming (OOP). The object-oriented methodology is inseparable from Java, and all Java programs are, to at least some extent, object-oriented. Because of OOP's importance to Java, it is useful to understand in a general way OOP's basic principles before you write even a simple Java program. Later in this book, you will see how to put these concepts into practice.

OOP is a powerful way to approach the job of programming. Programming methodologies have changed dramatically since the invention of the computer, primarily to accommodate the increasing complexity of programs. For example, when computers were first invented, programming was done by toggling in the binary machine instructions using the computer's 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, 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. Although FORTRAN was a very impressive first step, it is hardly a language that encourages clear, easy-to-understand programs.

The 1960s gave birth to structured programming. This is the method encouraged by languages such as C and Pascal. The use of structured languages made it possible to write moderately complex programs fairly easily. Structured languages are characterized by their support for stand-alone subroutines, local variables, rich control constructs, and their lack of reliance upon the GOTO. Although structured languages are a powerful tool, even they reach their limit when a project becomes too large.

Consider this: At each milestone in the development of programming, techniques and tools were created to allow the programmer to deal with increasingly greater complexity. Each step of the way, the new approach took the best elements of the previous methods and moved forward. Prior to the invention of OOP, many projects were nearing (or exceeding) the point

where the structured approach no longer works. Object-oriented methods were created to help programmers break through these barriers.

Object-oriented programming took the best ideas of structured programming and combined them with several new concepts. The result was a different way of organizing a program. In the most general sense, a program can be organized in one of two ways: around its code (what is happening) or around its data (what is being affected). Using only structured programming techniques, programs are typically organized around code. This approach can be thought of as "code acting on data."

Object-oriented programs work the other way around. They are organized around data, with the key principle being "data controlling access to code." In an object-oriented language, you define the data and the routines that are permitted to act on that data. Thus, a data type defines precisely what sort of operations can be applied to that data.

To support the principles of object-oriented programming, all OOP languages, including Java, have three traits in common: encapsulation, polymorphism, and inheritance. Let's examine each.

Encapsulation

Encapsulation is a programming mechanism that binds together code and the data it manipulates, and that keeps both safe from outside interference and misuse. In an object-oriented language, code and data can be bound together in such a way that a self-contained *black box* is created. Within the box are all necessary data and code. When code and data are linked together in this fashion, an object is created. In other words, an object is the device that supports encapsulation.

Within an object, code, data, or both may be *private* to that object or *public.* Private code or data is known to and accessible by only another part of the object. That is, private code or data cannot be accessed by a piece of the program that exists outside the object. When code or data is public, other parts of your program can access it even though it is defined within an object. Typically, the public parts of an object are used to provide a controlled interface to the private elements of the object.

Java's basic unit 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 form of an object. It specifies both the data and the code that will operate on that data. Java uses a class specification to construct *objects.* Objects are instances of a class. Thus, a class is essentially a set of plans that specify how to build an object.

The code and data that constitute a class 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. Method* is Java's term for a subroutine. 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.*

Polymorphism

Polymorphism (from Greek, meaning "many forms") is the quality that allows one interface to access a general class of actions. The specific action is determined by the exact nature of the situation. A simple example of polymorphism is found in the steering wheel of an automobile.