PRENTICE
HALL

CORE JAVA®

Volume I—Fundamentals

TENTH EDITION

CAY S. HORSTMANN

Core Java®

Volume I—Fundamentals

Tenth Edition

This page intentionally left blank

Core Java®

Volume I—Fundamentals

Tenth Edition

Cay S. Horstmann

PRENTICE
HALL

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

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the United States, please contact international@pearsoned.com.

Visit us on the Web: informit.com/ph

Library of Congress Cataloging-in-Publication Data
Names: Horstmann, Cay S., 1959- author.
Title: Core Java / Cay S. Horstmann.
Description: Tenth edition. | New York : Prentice Hall, [2016] | Includes index.
Identifiers: LCCN 2015038763| ISBN 9780134177304 (volume 1 : pbk. : alk. paper) | ISBN 0134177304 (volume 1 : pbk. : alk. paper)
Subjects: LCSH: Java (Computer program language)
Classification: LCC QA76.73.J38 H6753 2016 | DDC 005.13/3—dc23
LC record available at http://lccn.loc.gov/2015038763

500 Oracle Parkway, Redwood Shores, CA 94065

ISBN-13: 978-0-13-417730-4
ISBN-10: 0-13-417730-4

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, December 2015

Contents

Chapter 3: Fundamental Programming Structures in Java 41

Preface

To the Reader

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

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

The book you have in your hands is the first volume of the tenth edition of *Core Java®*. Each edition closely followed a release of the Java Development Kit, and each time, we rewrote the book to take advantage of the newest Java features. This edition has been updated to reflect the features of Java Standard Edition (SE) 8.

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

In this book you will find lots of sample code demonstrating almost every language and library feature that we discuss. We keep the sample programs purposefully simple to focus on the major points, but, for the most part, they aren't fake and they don't cut corners. They should make good starting points for your own code.

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

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

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

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

When writing a book, errors and inaccuracies are inevitable. We'd very much like to know about them. But, of course, we'd prefer to learn about each of them only once. We have put up a list of frequently asked questions, bug fixes, and workarounds on a web page at `http://horstmann.com/corejava`. Strategically placed at the end of the errata page (to encourage you to read through it first) is a form you can use to report bugs and suggest improvements. Please don't be disappointed if we don't answer every query or don't get back to you immediately. We do read all e-mail and appreciate your input to make future editions of this book clearer and more informative.

A Tour of This Book

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

In **Chapter 2**, we tell you how to download and install the JDK and the program examples for this book. Then we guide you through compiling and running three typical Java programs—a console application, a graphical application, and an applet—using the plain JDK, a Java-enabled text editor, and a Java IDE.

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

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

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

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

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

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

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

Chapter 10 starts the coverage of GUI programming. We show how you can make windows, how to paint on them, how to draw with geometric shapes, how to format text in multiple fonts, and how to display images.

Chapter 11 is a detailed discussion of the event model of the AWT, the *abstract window toolkit*. You'll see how to write code that responds to events, such as mouse clicks or key presses. Along the way you'll see how to handle basic GUI elements such as buttons and panels.

Chapter 12 discusses the Swing GUI toolkit in great detail. The Swing toolkit allows you to build cross-platform graphical user interfaces. You'll learn all about the various kinds of buttons, text components, borders, sliders, list boxes, menus, and dialog boxes. However, some of the more advanced components are discussed in Volume II.

Chapter 13 shows you how to deploy your programs, either as applications or applets. We describe how to package programs in JAR files, and how to deliver applications over the Internet with the Java Web Start and applet mechanisms. We also explain how Java programs can store and retrieve configuration information once they have been deployed.

Chapter 14 finishes the book with a discussion of concurrency, which enables you to program tasks to be done in parallel. This is an important and exciting application of Java technology in an era where most processors have multiple cores that you want to keep busy.

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

Conventions

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

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

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

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

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

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

`Application Programming Interface` 1.2

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

Listing 1.1 `InputTest/InputTest.java`

Sample Code

The web site for this book at `http://horstmann.com/corejava` contains all sample code from the book, in compressed form. You can expand the file either with one of the familiar unzipping programs or simply with the jar utility that is part of the Java Development Kit. See Chapter 2 for more information on installing the Java Development Kit and the sample code.

Acknowledgments

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

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

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

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

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

Cay Horstmann
Biel/Bienne, Switzerland
November 2015

CHAPTER 1

An Introduction to Java

In this chapter

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

1.1 Java as a Programming Platform

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

“As a computer language, Java’s hype is overdone: Java is certainly a *good* programming language. There is no doubt that it is one of the better languages available to serious programmers. We think it could *potentially* have been a great programming language, but it is probably too late for that. Once a language is out in the field, the ugly reality of compatibility with existing code sets in."

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

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

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

1.2 The Java "White Paper" Buzzwords

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

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

10. Multithreaded
11. Dynamic

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

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

1.2.1 Simple

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

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

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

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

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

1.2.2 Object-Oriented

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

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

1.2.3 Distributed

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

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

1.2.4 Robust

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

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

1.2.5 Secure

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

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

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

Originally, the Java attitude towards downloaded code was "Bring it on!" Untrusted code was executed in a sandbox environment where it could not impact the host system. Users were assured that nothing bad could happen because Java code, no matter where it came from, was incapable of escaping from the sandbox.

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

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

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

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

1.2.6 Architecture-Neutral

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

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

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

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

1.2.7 Portable

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

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

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

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

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

1.2.8 Interpreted

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

This seems a real stretch. Anyone who has used Lisp, Smalltalk, Visual Basic, Python, R, or Scala knows what a “rapid and exploratory” development process is. You try out something, and you instantly see the result. Java development environments are not focused on that experience.

1.2.9 High-Performance

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

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

1.2.10 Multithreaded

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

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

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

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

1.2.11 Dynamic

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

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

NOTE: Shortly after the initial success of Java, Microsoft released a product called J++ with a programming language and virtual machine that were almost identical to Java. At this point, Microsoft is no longer supporting J++ and has instead introduced another language called C# that also has many similarities with Java but runs on a different virtual machine. This book does not cover J++ or C#.

1.3 Java Applets and the Internet

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

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

Figure 1.1 shows a good example of a dynamic web page that carries out sophisticated calculations. The Jmol applet displays molecular structures. By using the mouse, you can rotate and zoom each molecule to better understand its structure. This kind of direct manipulation is not achievable with static web pages, but applets make it possible. (You can find this applet at `http://jmol.sourceforge.net`.)

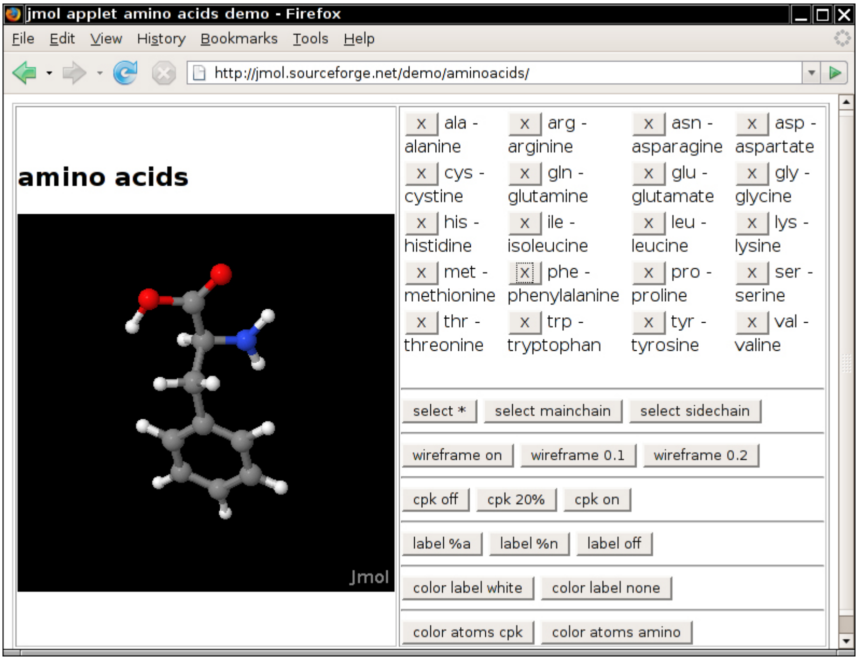

Figure 1.1 The Jmol applet

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