Thinking in Java

Fourth Edition

Bruce Eckel
President, MindView, Inc.

Sample Chapters

Comments from readers:

Thinking In Java should be read cover to cover by every Java programmer, then kept close at hand for frequent reference. The exercises are challenging, and the chapter on Collections is superb! Not only did this book help me to pass the Sun Certified Java Programmer exam; it's also the first book I turn to whenever I have a Java question. **Jim Pleger, Loudoun County (Virginia) Government**

Much better than any other Java book I've seen. Make that "by an order of magnitude"... very complete, with excellent right-to-the-point examples and intelligent, not dumbed-down, explanations ... In contrast to many other Java books I found it to be unusually mature, consistent, intellectually honest, well-written and precise. IMHO, an ideal book for studying Java. **Anatoly Vorobey, Technion University, Haifa, Israel**

One of the absolutely best programming tutorials I've seen for any language. **Joakim Ziegler, FIX sysop**

Thank you for your wonderful, wonderful book on Java. **Dr. Gavin Pillay, Registrar, King Edward VIII Hospital, South Africa**

Thank you again for your awesome book. I was really floundering (being a non-C programmer), but your book has brought me up to speed as fast as I could read it. It's really cool to be able to understand the underlying principles and concepts from the start, rather than having to try to build that conceptual model through trial and error. Hopefully I will be able to attend your seminar in the not-too-distant future. **Randall R. Hawley, Automation Technician, Eli Lilly & Co.**

The best computer book writing I have seen. **Tom Holland**

This is one of the best books I've read about a programming language... The best book ever written on Java. **Ravindra Pai, Oracle Corporation, SUNOS product line**

This is the best book on Java that I have ever found! You have done a great job. Your depth is amazing. I will be purchasing the book when it is published. I have been learning Java since October 96. I have read a few books, and consider yours a "MUST READ." These past few months we have been focused on a product written entirely in Java. Your book has helped solidify topics I was shaky on and has expanded my knowledge base. I have even used some of your explanations as information in interviewing contractors to help our team. I have found how much Java knowledge they have by asking them about things I have learned from reading your book (e.g., the difference between arrays and Vectors). Your book is great! **Steve Wilkinson, Senior Staff Specialist, MCI Telecommunications**

Great book. Best book on Java I have seen so far. **Jeff Sinclair, Software Engineer, Kestral Computing**

Thank you for *Thinking in Java*. It's time someone went beyond mere language description to a thoughtful, penetrating analytic tutorial that doesn't kowtow to The Manufacturers. I've read almost all the others—only yours and Patrick Winston's have found a place in my heart. I'm already recommending it to customers. Thanks again. **Richard Brooks, Java Consultant, Sun Professional Services, Dallas**

Bruce, your book is wonderful! Your explanations are clear and direct. Through your fantastic book I have gained a tremendous amount of Java knowledge. The exercises are also FANTASTIC and do an excellent job reinforcing the ideas explained throughout the chapters. I look forward to reading more books written by you. Thank you for the tremendous service that you are providing by writing such great books. My code will be much better after reading *Thinking in Java*. I thank you and I'm sure any programmers who will have to maintain my code are also grateful to you. **Yvonne Watkins, Java Artisan, Discover Technologies, Inc.**

Other books cover the WHAT of Java (describing the syntax and the libraries) or the HOW of Java (practical programming examples). *Thinking in Java* is the only book I know that explains the WHY of Java; why it was designed the way it was, why it works the way it does, why it sometimes doesn't work, why it's better than C++, why it's not. Although it also does a good job of teaching the what and how of the language, *Thinking in Java* is definitely the thinking person's choice in a Java book. **Robert S. Stephenson**

Thanks for writing a great book. The more I read it the better I like it. My students like it, too. **Chuck Iverson**

I just want to commend you for your work on *Thinking in Java*. It is people like you that dignify the future of the Internet and I just want to thank you for your effort. It is very much appreciated. **Patrick Barrell, Network Officer Mamco, QAF Mfg. Inc.**

I really, really appreciate your enthusiasm and your work. I download every revision of your online books and am looking into languages and exploring what I would never have dared (C#, C++, Python, and Ruby, as a side effect). I have at least 15 other Java books (I needed 3 to make both JavaScript and PHP viable!) and subscriptions to Dr. Dobbs, JavaPro, JDJ, JavaWorld, etc., as a result of my pursuit of Java (and Enterprise Java) and certification but I still keep your book in higher esteem. It truly is a thinking man's book. I subscribe to your newsletter and hope to one day sit down and solve some of the problems you extend for the solutions guides for you (I'll buy the guides!) in appreciation. But in the meantime, thanks a lot. **Joshua Long, www.starbuxman.com**

Most of the Java books out there are fine for a start, and most just have beginning stuff and a lot of the same examples. Yours is by far the best advanced thinking book I've seen. Please publish it soon! ... I also bought *Thinking in C++* just because I was so impressed with *Thinking in Java*. **George Laframboise, LightWorx Technology Consulting, Inc.**

I wrote to you earlier about my favorable impressions regarding your *Thinking in C++* (a book that stands prominently on my shelf here at work). And today I've been able to delve into Java with your e-book in my virtual hand, and I must say (in my best Chevy Chase from *Modern Problems*), "I like it!" Very informative and explanatory, without reading like a dry textbook. You cover the most important yet the least covered concepts of Java development: the whys. **Sean Brady**

I develop in both Java and C++, and both of your books have been lifesavers for me. If I am stumped about a particular concept, I know that I can count on your books to a) explain the thought to me clearly and b) have solid examples that pertain to what I am trying to accomplish. I have yet to find another author that I continually whole-heartedly recommend to anyone who is willing to listen. **Josh Asbury, A^3 Software Consulting, Cincinnati, Ohio**

Your examples are clear and easy to understand. You took care of many important details of Java that can't be found easily in the weak Java documentation. And you don't waste the reader's time with the basic facts a programmer already knows. **Kai Engert, Innovative Software, Germany**

I'm a great fan of your *Thinking in C++* and have recommended it to associates. As I go through the electronic version of your Java book, I'm finding that you've retained the same high level of writing. Thank you! **Peter R. Neuwald**

VERY well-written Java book...I think you've done a GREAT job on it. As the leader of a Chicago-area Java special interest group, I've favorably mentioned your book and Web site several times at our recent meetings. I would like to use *Thinking in Java* as the basis for a part of each monthly SIG meeting, in which we review and discuss each chapter in succession. **Mark Ertes**

By the way, printed TIJ2 in Russian is still selling great, and remains bestseller. Learning Java became synonym of reading TIJ2, isn't that nice? **Ivan Porty, translator and publisher of *Thinking in Java 2nd Edition* in Russian**

I really appreciate your work and your book is good. I recommend it here to our users and Ph.D. students. **Hugues Leroy // Irisa-Inria Rennes France, Head of Scientific Computing and Industrial Tranfert**

OK, I've only read about 40 pages of *Thinking in Java*, but I've already found it to be the most clearly written and presented programming book I've come across...and I'm a writer, myself, so I am probably a little critical. I have *Thinking in C++* on order and can't wait to crack it—I'm fairly new to programming and am hitting learning curves head-on everywhere. So this is just a quick note to say thanks for your excellent work. I had begun to burn a little low on enthusiasm from slogging through the mucky, murky prose of most computer books—even ones that came with glowing recommendations. I feel a whole lot better now. **Glenn Becker, Educational Theatre Association**

Thank you for making your wonderful book available. I have found it immensely useful in finally understanding what I experienced as confusing in Java and C++. Reading your book has been very satisfying. **Felix Bizaoui, Twin Oaks Industries, Louisa, Va.**

I must congratulate you on an excellent book. I decided to have a look at *Thinking in Java* based on my experience with *Thinking in C++*, and I was not disappointed. **Jaco van der Merwe, Software Specialist, DataFusion Systems Ltd, Stellenbosch, South Africa**

This has to be one of the best Java books I've seen. **E.F. Pritchard, Senior Software Engineer, Cambridge Animation Systems Ltd., United Kingdom**

Your book makes all the other Java books I've read or flipped through seem doubly useless and insulting. **Brett Porter, Senior Programmer, Art & Logic**

I have been reading your book for a week or two and compared to the books I have read earlier on Java, your book seems to have given me a great start. I have recommended this book to a lot of my friends and they have rated it excellent. Please accept my congratulations for coming out with an excellent book. **Rama Krishna Bhupathi, Software Engineer, TCSI Corporation, San Jose**

Just wanted to say what a "brilliant" piece of work your book is. I've been using it as a major reference for in-house Java work. I find that the table of contents is just right for quickly locating the section that is required. It's also nice to see a book that is not just a rehash of the API nor treats the programmer like a dummy. **Grant Sayer, Java Components Group Leader, Ceedata Systems Pty Ltd, Australia**

Wow! A readable, in-depth Java book. There are a lot of poor (and admittedly a couple of good) Java books out there, but from what I've seen yours is definitely one of the best. **John Root, Web Developer, Department of Social Security, London**

I've *just* started *Thinking in Java*. I expect it to be very good because I really liked *Thinking in C++* (which I read as an experienced C++ programmer, trying to stay ahead of the curve) ... You are a wonderful author. **Kevin K. Lewis, Technologist, ObjectSpace, Inc.**

I think it's a great book. I learned all I know about Java from this book. Thank you for making it available for free over the Internet. If you wouldn't have I'd know nothing about Java at all. But the best thing is that your book isn't a commercial brochure for Java. It also shows the bad sides of Java. YOU have done a great job here. **Frederik Fix, Belgium**

I have been hooked to your books all the time. A couple of years ago, when I wanted to start with C++, it was *C++ Inside & Out* which took me around the fascinating world of C++. It helped me in getting better opportunities in life. Now, in pursuit of more knowledge and when I wanted to learn Java, I bumped into *Thinking in Java*—no doubts in my mind as to whether I need some other book. Just fantastic. It is more like rediscovering myself as I get along with the book. It is just a month since I started with Java, and heartfelt thanks to you, I am understanding it better now. **Anand Kumar S., Software Engineer, Computervision, India**

Your book stands out as an excellent general introduction. **Peter Robinson, University of Cambridge Computer Laboratory**

It's by far the best material I have come across to help me learn Java and I just want you to know how lucky I feel to have found it. THANKS! **Chuck Peterson, Product Leader, Internet Product Line, IVIS International**

The book is great. It's the third book on Java I've started and I'm about two-thirds of the way through it now. I plan to finish this one. I found out about it because it is used in some internal classes at Lucent Technologies and a friend told me the book was on the Net. Good work. **Jerry Nowlin, MTS, Lucent Technologies**

Of the six or so Java books I've accumulated to date, your *Thinking in Java* is by far the best and clearest. **Michael Van Waas, Ph.D., President, TMR Associates**

I just want to say thanks for *Thinking in Java*. What a wonderful book you've made here! Not to mention downloadable for free! As a student I find your books invaluable (I have a copy of *C++ Inside Out*, another great book about C++), because they not only teach me the how-to, but also the whys, which are of course very important in building a strong foundation in languages such as C++ or Java. I have quite a lot of friends here who love programming just as I do, and I've told them about your books. They think it's great! Thanks again! By the way, I'm Indonesian and I live in Java. **Ray Frederick Djajadinata, Student at Trisakti University, Jakarta**

The mere fact that you have made this work free over the Net puts me into shock. I thought I'd let you know how much I appreciate and respect what you're doing. **Shane LeBouthillier, Computer Engineering student, University of Alberta, Canada**

I have to tell you how much I look forward to reading your monthly column. As a newbie to the world of object oriented programming, I appreciate the time and thoughtfulness that you give to even the most elementary topic. I have downloaded your book, but you can bet that I will purchase the hard copy when it is published. Thanks for all of your help. **Dan Cashmer, B. C. Ziegler & Co.**

Just want to congratulate you on a job well done. First I stumbled upon the PDF version of *Thinking in Java*. Even before I finished reading it, I ran to the store and found *Thinking in C++*. Now, I have been in the computer business for over eight years, as a consultant, software engineer, teacher/trainer, and recently as self-employed, so I'd like to think that I have seen enough (not "have seen it all," mind you, but enough). However, these books cause my girlfriend

to call me a ”geek.” Not that I have anything against the concept—it is just that I thought this phase was well beyond me. But I find myself truly enjoying both books, like no other computer book I have touched or bought so far. Excellent writing style, very nice introduction of every new topic, and lots of wisdom in the books. Well done. **Simon Goland, simonsez@smartt.com, Simon Says Consulting, Inc.**

I must say that your *Thinking in Java* is great! That is exactly the kind of documentation I was looking for. Especially the sections about good and poor software design using Java. **Dirk Duehr, Lexikon Verlag, Bertelsmann AG, Germany**

Thank you for writing two great books (*Thinking in C++*, *Thinking in Java*). You have helped me immensely in my progression to object oriented programming. **Donald Lawson, DCL Enterprises**

Thank you for taking the time to write a really helpful book on Java. If teaching makes you understand something, by now you must be pretty pleased with yourself. **Dominic Turner, GEAC Support**

It’s the best Java book I have ever read—and I read some. **Jean-Yves MENGANT, Chief Software Architect NAT-SYSTEM, Paris, France**

Thinking in Java gives the best coverage and explanation. Very easy to read, and I mean the code fragments as well. **Ron Chan, Ph.D., Expert Choice, Inc., Pittsburgh, Pa.**

Your book is great. I have read lots of programming books and your book still adds insights to programming in my mind. **Ningjian Wang, Information System Engineer, The Vanguard Group**

Thinking in Java is an excellent and readable book. I recommend it to all my students. **Dr. Paul Gorman, Department of Computer Science, University of Otago, Dunedin, New Zealand**

With your book, I have now understood what object oriented programming means. ... I believe that Java is much more straightforward and often even easier than Perl. **Torsten Römer, Orange Denmark**

You make it possible for the proverbial free lunch to exist, not just a soup kitchen type of lunch but a gourmet delight for those who appreciate good software and books about it. **Jose Suriol, Scylax Corporation**

Thanks for the opportunity of watching this book grow into a masterpiece! IT IS THE BEST book on the subject that I’ve read or browsed. **Jeff Lapchinsky, Programmer, Net Results Technologies**

Your book is concise, accessible and a joy to read. **Keith Ritchie, Java Research & Development Team, KL Group Inc.**

It truly is the best book I’ve read on Java! **Daniel Eng**

The best book I have seen on Java! **Rich Hoffarth, Senior Architect, West Group**

Thank you for a wonderful book. I’m having a lot of fun going through the chapters. **Fred Trimble, Actium Corporation**

You have mastered the art of slowly and successfully making us grasp the details. You make learning VERY easy and satisfying. Thank you for a truly wonderful tutorial. **Rajesh Rau, Software Consultant**

Thinking in Java rocks the free world! **Miko O'Sullivan, President, Idocs Inc.**

About *Thinking in C++*:

Winner of the 1995 Software Development Magazine Jolt Award for Best Book of the Year

"This book is a tremendous achievement. You owe it to yourself to have a copy on your shelf. The chapter on iostreams is the most comprehensive and understandable treatment of that subject I've seen to date."

Al Stevens
Contributing Editor, *Doctor Dobbs Journal*

"Eckel's book is the only one to so clearly explain how to rethink program construction for object orientation. That the book is also an excellent tutorial on the ins and outs of C++ is an added bonus."

Andrew Binstock
Editor, *Unix Review*

"Bruce continues to amaze me with his insight into C++, and *Thinking in C++* is his best collection of ideas yet. If you want clear answers to difficult questions about C++, buy this outstanding book."

Gary Entsminger
Author, *The Tao of Objects*

"*Thinking in C++* patiently and methodically explores the issues of when and how to use inlines, references, operator overloading, inheritance, and dynamic objects, as well as advanced topics such as the proper use of templates, exceptions and multiple inheritance. The entire effort is woven in a fabric that includes Eckel's own philosophy of object and program design. A must for every C++ developer's bookshelf, *Thinking in C++* is the one C++ book you must have if you're doing serious development with C++."

Richard Hale Shaw
Contributing Editor, *PC Magazine*

Thinking in Java

Fourth Edition

Bruce Eckel
President, MindView, Inc.

Upper Saddle River, NJ ● Boston ● Indianapolis ● San Francisco
New York ● Toronto ● Montreal ● London ● Munich ● Paris
Madrid ● Capetown ● Sydney ● Tokyo ● Singapore ● Mexico City

Visit us on the Web: www.prenhallprofessional.com

Cover design and interior design by Daniel Will-Harris, www.Will-Harris.com

Library of Congress Cataloging-in-Publication Data:

Eckel, Bruce.
Thinking in Java / Bruce Eckel.—4th ed.
p. cm.
Includes bibliographical references and index.
ISBN 0-13-187248-6 (pbk. : alk. paper)
1. Java (Computer program language) I. Title.
QA76.73.J38E25 2006
005.13'3—dc22

2005036339

ISBN 0-13-187248-6

Text printed in the United States on recycled paper at Courier in Stoughton, Massachusetts.

First printing, January 2006

Seminars and Consulting

Bruce Eckel
and his associates
are available for training in:

- Object-oriented design
- Java
- Design patterns

Consulting:

- Starting your OO design process
- Design reviews
- Code reviews
- Problem analysis

Public seminars are periodically held on various topics for individuals and small-staff training; check the calendar and seminar section at **www.MindView.net** for more information.

Dedication

To Dawn

Overview

What's Inside

Preface

I originally approached Java as "just another programming language," which in many senses it is.

But as time passed and I studied it more deeply, I began to see that the fundamental intent of this language was different from other languages I had seen up to that point.

Programming is about managing complexity: the complexity of the problem you want to solve, laid upon the complexity of the machine in which it is solved. Because of this complexity, most of our programming projects fail. And yet, of all the programming languages of which I am aware, almost none have gone all out and decided that their *main* design goal would be to conquer the complexity of developing and maintaining programs.[1] Of course, many language design decisions were made with complexity in mind, but at some point there were always other issues that were considered essential to be added into the mix. Inevitably, those other issues are what cause programmers to eventually "hit the wall" with that language. For example, C++ had to be backwards-compatible with C (to allow easy migration for C programmers), as well as efficient. Those are both very useful goals and account for much of the success of C++, but they also expose extra complexity that prevents some projects from being finished (certainly, you can blame programmers and management, but if a language can help by catching your mistakes, why shouldn't it?). As another example, Visual BASIC (VB) was tied to BASIC, which wasn't really designed to be an extensible language, so all the extensions piled upon VB have produced some truly unmaintainable syntax. Perl is backwards-compatible with awk, sed, grep, and other Unix tools it was meant to replace, and as a result it is often accused of producing "write-only code" (that is, after a while you can't read it). On the other hand, C++, VB, Perl, and other languages like Smalltalk had *some* of their design efforts focused on the issue of complexity and as a result are remarkably successful in solving certain types of problems.

What has impressed me most as I have come to understand Java is that somewhere in the mix of Sun's design objectives, it seems that there was a goal of reducing complexity *for the programmer*. As if to say, "We care about reducing the time and difficulty of producing robust code." In the early days, this goal resulted in code that didn't run very fast (although this has improved over time), but it has indeed produced amazing reductions in development time—half or less of the time that it takes to create an equivalent C++ program. This result alone can save incredible amounts of time and money, but Java doesn't stop there. It goes on to wrap many of the complex tasks that have become important, such as multithreading and network programming, in language features or libraries that can at times make those tasks easy. And finally, it tackles some really big complexity problems: cross-platform programs, dynamic code changes, and even security, each of which can fit on your complexity spectrum anywhere from "impediment" to "show-stopper." So despite the performance problems that we've seen, the promise of Java is tremendous: It can make us significantly more productive programmers.

In all ways—creating the programs, working in teams, building user interfaces to communicate with the user, running the programs on different types of machines, and easily writing programs that communicate across the Internet—Java increases the communication bandwidth *between people*.

---

[1] However, I believe that the Python language comes closest to doing exactly that. See *www.Python.org*.

I think that the results of the communication revolution may not be seen from the effects of moving large quantities of bits around. We shall see the true revolution because we will all communicate with each other more easily: one-on-one, but also in groups and as a planet. I've heard it suggested that the next revolution is the formation of a kind of global mind that results from enough people and enough interconnectedness. Java may or may not be the tool that foments that revolution, but at least the possibility has made me feel like I'm doing something meaningful by attempting to teach the language.

Java SE5 and SE6

This edition of the book benefits greatly from the improvements made to the Java language in what Sun originally called JDK 1.5, and then later changed to JDK5 or J2SE5, then finally they dropped the outdated "2" and changed it to Java SE5. Many of the Java SE5 language changes were designed to improve the experience of the programmer. As you shall see, the Java language designers did not completely succeed at this task, but in general they made large steps in the right direction.

One of the important goals of this edition is to completely absorb the improvements of Java SE5/6, and to introduce and use them throughout this book. This means that this edition takes the somewhat bold step of being "Java SE5/6-only," and much of the code in the book will not compile with earlier versions of Java; the build system will complain and stop if you try. However, I think the benefits are worth the risk.

If you are somehow fettered to earlier versions of Java, I have covered the bases by providing free downloads of previous editions of this book via *www.MindView.net*. For various reasons, I have decided not to provide the current edition of the book in free electronic form, but only the prior editions.

Java SE6

This book was a monumental, time-consuming project, and before it was published, Java SE6 (code-named *mustang*) appeared in beta form. Although there were a few minor changes in Java SE6 that improved some of the examples in the book, for the most part the focus of Java SE6 did not affect the content of this book; the features were primarily speed improvements and library features that were outside the purview of this text.

The code in this book was successfully tested with a release candidate of Java SE6, so I do not expect any changes that will affect the content of this book. If there are any important changes by the time Java SE6 is officially released, these will be reflected in the book's source code, which is downloadable from *www.MindView.net*.

The cover indicates that this book is for "Java SE5/6," which means "written for Java SE5 and the very significant changes that version introduced into the language, but is equally applicable to Java SE6."

The 4th edition

The satisfaction of doing a new edition of a book is in getting things "right," according to what I have learned since the last edition came out. Often these insights are in the nature of the saying "A learning experience is what you get when you don't get what you want," and my opportunity is to fix something embarrassing or simply tedious. Just as often, creating the next edition produces fascinating new ideas, and the embarrassment is far outweighed by the delight of discovery and the ability to express ideas in a better form than what I have previously achieved.

There is also the challenge that whispers in the back of my brain, that of making the book something that owners of previous editions will want to buy. This presses me to improve, rewrite and reorganize everything that I can, to make the book a new and valuable experience for dedicated readers.

Changes

The CD-ROM that has traditionally been packaged as part of this book is not part of this edition. The essential part of that CD, the *Thinking in C* multimedia seminar (created for MindView by Chuck Allison), is now available as a downloadable Flash presentation. The goal of that seminar is to prepare those who are not familiar enough with C syntax to understand the material presented in this book. Although two of the chapters in this book give decent introductory syntax coverage, they may not be enough for people without an adequate background, and *Thinking in C* is intended to help those people get to the necessary level.

The *Concurrency* chapter (formerly called "Multithreading") has been completely rewritten to match the major changes in the Java SE5 concurrency libraries, but it still gives you a basic foundation in the core ideas of concurrency. Without that core, it's hard to understand more complex issues of threading. I spent many months working on this, immersed in that netherworld called "concurrency," and in the end the chapter is something that not only provides a basic foundation but also ventures into more advanced territory.

There is a new chapter on every significant new Java SE5 language feature, and the other new features have been woven into modifications made to the existing material. Because of my continuing study of design patterns, more patterns have been introduced throughout the book as well.

The book has undergone significant reorganization. Much of this has come from the teaching process together with a realization that, perhaps, my perception of what a "chapter" was could stand some rethought. I have tended towards an unconsidered belief that a topic had to be "big enough" to justify being a chapter. But especially while teaching design patterns, I find that seminar attendees do best if I introduce a single pattern and then we immediately do an exercise, even if it means I only speak for a brief time (I discovered that this pace was also more enjoyable for me as a teacher). So in this version of the book I've tried to break chapters up by topic, and not worry about the resulting length of the chapters. I think it has been an improvement.

I have also come to realize the importance of code testing. Without a built-in test framework with tests that are run every time you do a build of your system, you have no way of knowing if your code is reliable or not. To accomplish this in the book, I created a test framework to display and validate the output of each program. (The framework was written in Python; you can find it in the downloadable code for this book at *www.MindView.net*.) Testing in general is covered in the supplement you will find at *http://MindView.net/Books/BetterJava*, which introduces what I now believe are fundamental skills that all programmers should have in their basic toolkit.

In addition, I've gone over every single example in the book and asked myself, "Why did I do it this way?" In most cases I have done some modification and improvement, both to make the examples more consistent within themselves and also to demonstrate what I consider to be best practices in Java coding (at least, within the limitations of an introductory text). Many of the existing examples have had very significant redesign and reimplementation. Examples that no longer made sense to me were removed, and new examples have been added.

Readers have made many, many wonderful comments about the first three editions of this book, which has naturally been very pleasant for me. However, every now and then, someone will have

complaints, and for some reason one complaint that comes up periodically is "The book is too big." In my mind it is faint damnation indeed if "too many pages" is your only gripe. (One is reminded of the Emperor of Austria's complaint about Mozart's work: "Too many notes!" Not that I am in any way trying to compare myself to Mozart.) In addition, I can only assume that such a complaint comes from someone who is yet to be acquainted with the vastness of the Java language itself and has not seen the rest of the books on the subject. Despite this, one of the things I have attempted to do in this edition is trim out the portions that have become obsolete, or at least nonessential. In general, I've tried to go over everything, remove what is no longer necessary, include changes, and improve everything I could. I feel comfortable removing portions because the original material remains on the Web site (*www.MindView.net*), in the form of the freely downloadable 1st through 3rd editions of the book, and in the downloadable supplements for this book.

For those of you who still can't stand the size of the book, I do apologize. Believe it or not, I have worked hard to keep the size down.

Note on the cover design

The cover of *Thinking in Java* is inspired by the American Arts & Crafts Movement that began near the turn of the century and reached its zenith between 1900 and 1920. It began in England as a reaction to both the machine production of the Industrial Revolution and the highly ornamental style of the Victorian era. Arts & Crafts emphasized spare design, the forms of nature as seen in the art nouveau movement, hand-crafting, and the importance of the individual craftsperson, and yet it did not eschew the use of modern tools. There are many echoes with the situation we have today: the turn of the century, the evolution from the raw beginnings of the computer revolution to something more refined and meaningful, and the emphasis on software craftsmanship rather than just manufacturing code.

I see Java in this same way: as an attempt to elevate the programmer away from an operating system mechanic and toward being a "software craftsman."

Both the author and the book/cover designer (who have been friends since childhood) find inspiration in this movement, and both own furniture, lamps, and other pieces that are either original or inspired by this period.

The other theme in this cover suggests a collection box that a naturalist might use to display the insect specimens that he or she has preserved. These insects are objects that are placed within the box objects. The box objects are themselves placed within the "cover object," which illustrates the fundamental concept of aggregation in object-oriented programming. Of course, a programmer cannot help but make the association with "bugs," and here the bugs have been captured and presumably killed in a specimen jar, and finally confined within a small display box, as if to imply Java's ability to find, display, and subdue bugs (which is truly one of its most powerful attributes).

In this edition, I created the watercolor painting that you see as the cover background.

Acknowledgements

First, thanks to associates who have worked with me to give seminars, provide consulting, and develop teaching projects: Dave Bartlett, Bill Venners, Chuck Allison, Jeremy Meyer, and Jamie King. I appreciate your patience as I continue to try to develop the best model for independent folks like us to work together.

Recently, no doubt because of the Internet, I have become associated with a surprisingly large number of people who assist me in my endeavors, usually working from their own home offices. In the past, I would have had to pay for a pretty big office space to accommodate all these folks, but because of the Net, FedEx, and the telephone, I'm able to benefit from their help without the extra costs. In my attempts to learn to "play well with others," you have all been very helpful, and I hope to continue learning how to make my own work better through the efforts of others. Paula Steuer has been invaluable in taking over my haphazard business practices and making them sane (thanks for prodding me when I don't want to do something, Paula). Jonathan Wilcox, Esq., has sifted through my corporate structure and turned over every possible rock that might hide scorpions, and frog-marched us through the process of putting everything straight, legally. Thanks for your care and persistence. Sharlynn Cobaugh has made herself an expert in sound processing and an essential part of creating the multimedia training experiences, as well as tackling other problems. Thanks for your perseverance when faced with intractable computer problems. The folks at Amaio in Prague have helped me out with several projects. Daniel Will-Harris was the original work-by-Internet inspiration, and he is of course fundamental to all my graphic design solutions.

Over the years, through his conferences and workshops, Gerald Weinberg has become my unofficial coach and mentor, for which I thank him.

Ervin Varga was exceptionally helpful with technical corrections on the 4th edition—although other people helped on various chapters and examples, Ervin was my primary technical reviewer for the book, and he also took on the task of rewriting the solution guide for the 4th edition. Ervin found errors and made improvements to the book that were invaluable additions to this text. His thoroughness and attention to detail are amazing, and he's far and away the best technical reader I've ever had. Thanks, Ervin.

My weblog on Bill Venners' *www.Artima.com* has been a source of assistance when I've needed to bounce ideas around. Thanks to the readers that have helped me clarify concepts by submitting comments, including James Watson, Howard Lovatt, Michael Barker, and others, in particular those who helped with generics.

Thanks to Mark Welsh for his continuing assistance.

Evan Cofsky continues to be very supportive by knowing off the top of his head all the arcane details of setting up and maintaining Linux-based Web servers, and keeping the MindView server tuned and secure.

A special thanks to my new friend, coffee, who generated nearly boundless enthusiasm for this project. Camp4 Coffee in Crested Butte, Colorado, has become the standard hangout when people have come up to take MindView seminars, and during seminar breaks it is the best catering I've ever had. Thanks to my buddy Al Smith for creating it and making it such a great place, and for being such an interesting and entertaining part of the Crested Butte experience. And to all the Camp4 barristas who so cheerfully dole out beverages.

Thanks to the folks at Prentice Hall for continuing to give me what I want, putting up with all my special requirements, and for going out of their way to make things run smoothly for me.

Certain tools have proved invaluable during my development process and I am very grateful to the creators every time I use these. Cygwin (*www.cygwin.com*) has solved innumerable problems for me that Windows can't/won't and I become more attached to it each day (if I only had this 15 years ago when my brain was still hard-wired with Gnu Emacs). IBM's Eclipse (*www.eclipse.org*) is a truly wonderful contribution to the development community, and I expect

to see great things from it as it continues to evolve (how did IBM become hip? I must have missed a memo). JetBrains IntelliJ Idea continues to forge creative new paths in development tools.

I began using Enterprise Architect from Sparxsystems on this book, and it has rapidly become my UML tool of choice. Marco Hunsicker's Jalopy code formatter (*www.triemax.com*) came in handy on numerous occasions, and Marco was very helpful in configuring it to my particular needs. I've also found Slava Pestov's JEdit and plug-ins to be helpful at times (*www.jedit.org*) and it's quite a reasonable beginner's editor for seminars.

And of course, if I don't say it enough everywhere else, I use Python (*www.Python.org*) constantly to solve problems, the brainchild of my buddy Guido Van Rossum and the gang of goofy geniuses with whom I spent a few great days sprinting (Tim Peters, I've now framed that mouse you borrowed, officially named the "TimBotMouse"). You guys need to find healthier places to eat lunch. (Also, thanks to the entire Python community, an amazing bunch of people.)

Lots of people sent in corrections and I am indebted to them all, but particular thanks go to (for the 1st edition): Kevin Raulerson (found tons of great bugs), Bob Resendes (simply incredible), John Pinto, Joe Dante, Joe Sharp (all three were fabulous), David Combs (many grammar and clarification corrections), Dr. Robert Stephenson, John Cook, Franklin Chen, Zev Griner, David Karr, Leander A. Stroschein, Steve Clark, Charles A. Lee, Austin Maher, Dennis P. Roth, Roque Oliveira, Douglas Dunn, Dejan Ristic, Neil Galarneau, David B. Malkovsky, Steve Wilkinson, and a host of others. Prof. Ir. Marc Meurrens put in a great deal of effort to publicize and make the electronic version of the 1st edition of the book available in Europe.

Thanks to those who helped me rewrite the examples to use the Swing library (for the 2nd edition), and for other assistance: Jon Shvarts, Thomas Kirsch, Rahim Adatia, Rajesh Jain, Ravi Manthena, Banu Rajamani, Jens Brandt, Nitin Shivaram, Malcolm Davis, and everyone who expressed support.

In the 4th edition, Chris Grindstaff was very helpful during the development of the SWT section, and Sean Neville wrote the first draft of the Flex section for me.

Kraig Brockschmidt and Gen Kiyooka have been some of the smart technical people in my life who have become friends and have also been both influential and unusual in that they do yoga and practice other forms of spiritual enhancement, which I find quite inspirational and instructional.

It's not that much of a surprise to me that understanding Delphi helped me understand Java, since there are many concepts and language design decisions in common. My Delphi friends provided assistance by helping me gain insight into that marvelous programming environment. They are Marco Cantu (another Italian—perhaps being steeped in Latin gives one aptitude for programming languages?), Neil Rubenking (who used to do the yoga/vegetarian/Zen thing until he discovered computers), and of course Zack Urlocker (the original Delphi product manager), a long-time pal whom I've traveled the world with. We're all indebted to the brilliance of Anders Hejlsberg, who continues to toil away at C# (which, as you'll learn in this book, was a major inspiration for Java SE5).

My friend Richard Hale Shaw's insights and support have been very helpful (and Kim's, too). Richard and I spent many months giving seminars together and trying to work out the perfect learning experience for the attendees.

The book design, cover design, and cover photo were created by my friend Daniel Will-Harris, noted author and designer (*www.Will-Harris.com*), who used to play with rub-on letters in

junior high school while he awaited the invention of computers and desktop publishing, and complained of me mumbling over my algebra problems. However, I produced the camera-ready pages myself, so the typesetting errors are mine. Microsoft® Word XP for Windows was used to write the book and to create camera-ready pages in Adobe Acrobat; the book was created directly from the Acrobat PDF files. As a tribute to the electronic age, I happened to be overseas when I produced the final versions of the 1st and 2nd editions of the book—the 1st edition was sent from Cape Town, South Africa, and the 2nd edition was posted from Prague. The 3rd and 4th came from Crested Butte, Colorado. The body typeface is *Georgia* and the headlines are in *Verdana*. The cover typeface is *ITC Rennie Mackintosh*.

A special thanks to all my teachers and all my students (who are my teachers as well).

Molly the cat often sat in my lap while I worked on this edition, and thus offered her own kind of warm, furry support.

The supporting cast of friends includes, but is not limited to: Patty Gast (Masseuse extraordinaire), Andrew Binstock, Steve Sinofsky, JD Hildebrandt, Tom Keffer, Brian McElhinney, Brinkley Barr, Bill Gates at *Midnight Engineering Magazine*, Larry Constantine and Lucy Lockwood, Gene Wang, Dave Mayer, David Intersimone, Chris and Laura Strand, the Almquists, Brad Jerbic, Marilyn Cvitanic, Mark Mabry, the Robbins families, the Moelter families (and the McMillans), Michael Wilk, Dave Stoner, the Cranstons, Larry Fogg, Mike Sequeira, Gary Entsminger, Kevin and Sonda Donovan, Joe Lordi, Dave and Brenda Bartlett, Patti Gast, Blake, Annette & Jade, the Rentschlers, the Sudeks, Dick, Patty, and Lee Eckel, Lynn and Todd, and their families. And of course, Mom and Dad.

Introduction

"He gave man speech, and speech created thought, Which is the measure of the Universe"—*Prometheus Unbound*, Shelley

> *Human beings ... are very much at the mercy of the particular language which has become the medium of expression for their society. It is quite an illusion to imagine that one adjusts to reality essentially without the use of language and that language is merely an incidental means of solving specific problems of communication and reflection. The fact of the matter is that the "real world" is to a large extent unconsciously built up on the language habits of the group.*
>
> *The Status of Linguistics as a Science*, 1929, Edward Sapir

Like any human language, Java provides a way to express concepts. If successful, this medium of expression will be significantly easier and more flexible than the alternatives as problems grow larger and more complex.

You can't look at Java as just a collection of features—some of the features make no sense in isolation. You can use the sum of the parts only if you are thinking about *design*, not simply coding. And to understand Java in this way, you must understand the problems with the language and with programming in general. This book discusses programming problems, why they are problems, and the approach Java has taken to solve them. Thus, the set of features that I explain in each chapter are based on the way I see a particular type of problem being solved with the language. In this way I hope to move you, a little at a time, to the point where the Java mindset becomes your native tongue.

Throughout, I'll be taking the attitude that you want to build a model in your head that allows you to develop a deep understanding of the language; if you encounter a puzzle, you'll feed it to your model and deduce the answer.

Prerequisites

This book assumes that you have some programming familiarity: You understand that a program is a collection of statements, the idea of a subroutine/function/macro, control statements such as "if" and looping constructs such as "while," etc. However, you might have learned this in many places, such as programming with a macro language or working with a tool like Perl. As long as you've programmed to the point where you feel comfortable with the basic ideas of programming, you'll be able to work through this book. Of course, the book will be *easier* for C programmers and more so for C++ programmers, but don't count yourself out if you're not experienced with those languages—however, come willing to work hard. Also, the *Thinking in C* multimedia seminar that you can download from *www.MindView.net* will bring you up to speed in the fundamentals necessary to learn Java. However, I will be introducing the concepts of object-oriented programming (OOP) and Java's basic control mechanisms.

Although references may be made to C and C++ language features, these are not intended to be insider comments, but instead to help all programmers put Java in perspective with those languages, from which, after all, Java is descended. I will attempt to make these references simple and to explain anything that I think a non-C/C++ programmer would not be familiar with.

Learning Java

At about the same time that my first book, *Using C++* (Osborne/McGraw-Hill, 1989), came out, I began teaching that language. Teaching programming ideas has become my profession; I've seen nodding heads, blank faces, and puzzled expressions in audiences all over the world since 1987. As I began giving in-house training with smaller groups of people, I discovered something during the exercises. Even those people who were smiling and nodding were confused about many issues. I found out, by creating and chairing the C++ track at the Software Development Conference for a number of years (and later creating and chairing the Java track), that I and other speakers tended to give the typical audience too many topics too quickly. So eventually, through both variety in the audience level and the way that I presented the material, I would end up losing some portion of the audience. Maybe it's asking too much, but because I am one of those people resistant to traditional lecturing (and for most people, I believe, such resistance results from boredom), I wanted to try to keep everyone up to speed.

For a time, I was creating a number of different presentations in fairly short order. Thus, I ended up learning by experiment and iteration (a technique that also works well in program design). Eventually, I developed a course using everything I had learned from my teaching experience. My company, MindView, Inc., now gives this as the public and in-house *Thinking in Java* seminar; this is our main introductory seminar that provides the foundation for our more advanced seminars. You can find details at *www.MindView.net*. (The introductory seminar is also available as the *Hands-On Java* CD ROM. Information is available at the same Web site.)

The feedback that I get from each seminar helps me change and refocus the material until I think it works well as a teaching medium. But this book isn't just seminar notes; I tried to pack as much information as I could within these pages, and structured it to draw you through into the next subject. More than anything, the book is designed to serve the solitary reader who is struggling with a new programming language.

Goals

Like my previous book, *Thinking in C++*, this book was designed with one thing in mind: the way people learn a language. When I think of a chapter in the book, I think in terms of what makes a good lesson during a seminar. Seminar audience feedback helped me understand the difficult parts that needed illumination. In the areas where I got ambitious and included too many features all at once, I came to know—through the process of presenting the material—that if you include a lot of new features, you need to explain them all, and this easily compounds the student's confusion.

Each chapter tries to teach a single feature, or a small group of associated features, without relying on concepts that haven't been introduced yet. That way you can digest each piece in the context of your current knowledge before moving on.

My goals in this book are to:

1. Present the material one simple step at a time so that you can easily digest each idea before moving on. Carefully sequence the presentation of features so that you're exposed to a topic before you see it in use. Of course, this isn't always possible; in those situations, a brief introductory description is given.
2. Use examples that are as simple and short as possible. This sometimes prevents me from tackling "real world" problems, but I've found that beginners are usually happier when they can understand every detail of an example rather than being impressed by

the scope of the problem it solves. Also, there's a severe limit to the amount of code that can be absorbed in a classroom situation. For this I will no doubt receive criticism for using "toy examples," but I'm willing to accept that in favor of producing something pedagogically useful.

3. Give you what I think is important for you to understand about the language, rather than everything that I know. I believe there is an information importance hierarchy, and that there are some facts that 95 percent of programmers will never need to know—details that just confuse people and increase their perception of the complexity of the language. To take an example from C, if you memorize the operator precedence table (I never did), you can write clever code. But if you need to think about it, it will also confuse the reader/maintainer of that code. So forget about precedence, and use parentheses when things aren't clear.

4. Keep each section focused enough so that the lecture time—and the time between exercise periods—is small. Not only does this keep the audience's minds more active and involved during a hands-on seminar, but it gives the reader a greater sense of accomplishment.

5. Provide you with a solid foundation so that you can understand the issues well enough to move on to more difficult coursework and books.

Teaching from this book

The original edition of this book evolved from a one-week seminar which was, when Java was in its infancy, enough time to cover the language. As Java grew and continued to encompass more and more features and libraries, I stubbornly tried to teach it all in one week. At one point, a customer asked me to teach "just the fundamentals," and in doing so I discovered that trying to cram everything into a single week had become painful for both myself and for seminarians. Java was no longer a "simple" language that could be taught in a week.

That experience and realization drove much of the reorganization of this book, which is now designed to support a two-week seminar or a two-term college course. The introductory portion ends with the *Error Handling with Exceptions* chapter, but you may also want to supplement this with an introduction to JDBC, Servlets and JSPs. This provides a foundation course, and is the core of the *Hands-On Java* CD ROM. The remainder of the book comprises an intermediate-level course, and is the material covered in the *Intermediate Thinking in Java* CD ROM. Both of these CD ROMs are for sale at *www.MindView.net*.

Contact Prentice-Hall at *www.prenhallprofessional.com* for information about professor support materials for this book.

JDK HTML documentation

The Java language and libraries from Sun Microsystems (a free download from *http://java.sun.com*) come with documentation in electronic form, readable using a Web browser. Many books published on Java have duplicated this documentation. So you either already have it or you can download it, and unless necessary, this book will not repeat that documentation, because it's usually much faster if you find the class descriptions with your Web browser than if you look them up in a book (and the online documentation is probably more up-to-date). You'll simply be referred to "the JDK documentation." This book will provide extra descriptions of the classes only when it's necessary to supplement that documentation so you can understand a particular example.

Exercises

I've discovered that simple exercises are exceptionally useful to complete a student's understanding during a seminar, so you'll find a set at the end of each chapter.

Most exercises are designed to be easy enough that they can be finished in a reasonable amount of time in a classroom situation while the instructor observes, making sure that all the students are absorbing the material. Some are more challenging, but none present major challenges.

Solutions to selected exercises can be found in the electronic document *The Thinking in Java Annotated Solution Guide*, available for sale from *www.MindView.net*.

Foundations for Java

Another bonus with this edition is the free multimedia seminar that you can download from *www.MindView.net*. This is the *Thinking in C* seminar that gives you an introduction to the C syntax, operators, and functions that Java syntax is based upon. In previous editions of the book this was in the *Foundations for Java* CD that was packaged with the book, but now the seminar may be freely downloaded.

I originally commissioned Chuck Allison to create *Thinking in C* as a standalone product, but decided to include it with the 2nd edition of *Thinking in C++* and 2nd and 3rd editions of *Thinking in Java* because of the consistent experience of having people come to seminars without an adequate background in basic C syntax. The thinking apparently goes "I'm a smart programmer and I don't want to learn C, but rather C++ or Java, so I'll just skip C and go directly to C++/Java." After arriving at the seminar, it slowly dawns on folks that the prerequisite of understanding C syntax is there for a very good reason.

Technologies have changed, and it made more sense to rework *Thinking in C* as a downloadable Flash presentation rather than including it as a CD. By providing this seminar online, I can ensure that everyone can begin with adequate preparation.

The *Thinking in C* seminar also allows the book to appeal to a wider audience. Even though the *Operators* and *Controlling Execution* chapters do cover the fundamental parts of Java that come from C, the online seminar is a gentler introduction, and assumes even less about the student's programming background than does the book.

Source code

All the source code for this book is available as copyrighted freeware, distributed as a single package, by visiting the Web site *www.MindView.net*. To make sure that you get the most current version, this is the official code distribution site. You may distribute the code in classroom and other educational situations.

The primary goal of the copyright is to ensure that the source of the code is properly cited, and to prevent you from republishing the code in print media without permission. (As long as the source is cited, using examples from the book in most media is generally not a problem.)

In each source-code file you will find a reference to the following copyright notice:

```
//:! Copyright.txt
This computer source code is Copyright ©2006 MindView, Inc.
All Rights Reserved.
```

```
OBLIGATIONS TO PROVIDE MAINTENANCE, SUPPORT, UPDATES,
ENHANCEMENTS, OR MODIFICATIONS.

Please note that MindView, Inc. maintains a Web site which
is the sole distribution point for electronic copies of the Source Code,
http://www.MindView.net (and official mirror
sites), where it is freely available under the terms stated above.

If you think you've found an error in the Source Code,
please submit a correction using the feedback system that you will find at
http://www.MindView.net.
///:~
```

You may use the code in your projects and in the classroom (including your presentation materials) as long as the copyright notice that appears in each source file is retained.

Coding standards

In the text of this book, identifiers (methods, variables, and class names) are set in **bold**. Most keywords are also set in bold, except for those keywords that are used so much that the bolding can become tedious, such as "class."

I use a particular coding style for the examples in this book. As much as possible, this follows the style that Sun itself uses in virtually all of the code you will find at its site (see *http://java.sun.com/docs/codeconv/index.html*), and seems to be supported by most Java development environments. If you've read my other works, you'll also notice that Sun's coding style coincides with mine—this pleases me, although I had nothing (that I know of) to do with it. The subject of formatting style is good for hours of hot debate, so I'll just say I'm not trying to dictate correct style via my examples; I have my own motivation for using the style that I do. Because Java is a free-form programming language, you can continue to use whatever style you're comfortable with. One solution to the coding style issue is to use a tool like *Jalopy* (*www.triemax.com*), which assisted me in developing this book, to change formatting to that which suits you.

The code files printed in the book are tested with an automated system, and should all work without compiler errors.

This book focuses on and is tested with Java SE5/6. If you need to learn about earlier releases of the language that are not covered in this edition, the 1st through 3rd editions of the book are freely downloadable at *www.MindView.net*.

Errors

No matter how many tools a writer uses to detect errors, some always creep in and these often leap off the page for a fresh reader. If you discover anything you believe to be an error, please use the link you will find for this book at *www.MindView.net* to submit the error along with your suggested correction. Your help is appreciated.

Introduction to Objects

"We cut nature up, organize it into concepts, and ascribe significances as we do, largely because we are parties to an agreement that holds throughout our speech community and is codified in the patterns of our language ... we cannot talk at all except by subscribing to the organization and classification of data which the agreement decrees." Benjamin Lee Whorf (1897-1941)

The genesis of the computer revolution was in a machine. The genesis of our programming languages thus tends to look like that machine.

But computers are not so much machines as they are mind amplification tools ("bicycles for the mind," as Steve Jobs is fond of saying) and a different kind of expressive medium. As a result, the tools are beginning to look less like machines and more like parts of our minds, and also like other forms of expression such as writing, painting, sculpture, animation, and filmmaking. Object-oriented programming (OOP) is part of this movement toward using the computer as an expressive medium.

This chapter will introduce you to the basic concepts of OOP, including an overview of development methods. This chapter, and this book, assumes that you have some programming experience, although not necessarily in C. If you think you need more preparation in programming before tackling this book, you should work through the *Thinking in C* multimedia seminar, downloadable from *www.MindView.net*.

This chapter is background and supplementary material. Many people do not feel comfortable wading into object-oriented programming without understanding the big picture first. Thus, there are many concepts that are introduced here to give you a solid overview of OOP. However, other people may not get the big picture concepts until they've seen some of the mechanics first; these people may become bogged down and lost without some code to get their hands on. If you're part of this latter group and are eager to get to the specifics of the language, feel free to jump past this chapter—skipping it at this point will not prevent you from writing programs or learning the language. However, you will want to come back here eventually to fill in your knowledge so you can understand why objects are important and how to design with them.

The progress of abstraction

All programming languages provide abstractions. It can be argued that the complexity of the problems you're able to solve is directly related to the kind and quality of abstraction. By "kind" I mean, "What is it that you are abstracting?" Assembly language is a small abstraction of the underlying machine. Many so-called "imperative" languages that followed (such as FORTRAN, BASIC, and C) were abstractions of assembly language. These languages are big improvements over assembly language, but their primary abstraction still requires you to think in terms of the structure of the computer rather than the structure of the problem you are trying to solve. The programmer must establish the association between the machine model (in the "solution space,"

which is the place where you're implementing that solution, such as a computer) and the model of the problem that is actually being solved (in the "problem space," which is the place where the problem exists, such as a business). The effort required to perform this mapping, and the fact that it is extrinsic to the programming language, produces programs that are difficult to write and expensive to maintain, and as a side effect created the entire "programming methods" industry.

The alternative to modeling the machine is to model the problem you're trying to solve. Early languages such as LISP and APL chose particular views of the world ("All problems are ultimately lists" or "All problems are algorithmic," respectively). Prolog casts all problems into chains of decisions. Languages have been created for constraint-based programming and for programming exclusively by manipulating graphical symbols. (The latter proved to be too restrictive.) Each of these approaches may be a good solution to the particular class of problem they're designed to solve, but when you step outside of that domain they become awkward.

The object-oriented approach goes a step further by providing tools for the programmer to represent elements in the problem space. This representation is general enough that the programmer is not constrained to any particular type of problem. We refer to the elements in the problem space and their representations in the solution space as "objects." (You will also need other objects that don't have problem-space analogs.) The idea is that the program is allowed to adapt itself to the lingo of the problem by adding new types of objects, so when you read the code describing the solution, you're reading words that also express the problem. This is a more flexible and powerful language abstraction than what we've had before.[1] Thus, OOP allows you to describe the problem in terms of the problem, rather than in terms of the computer where the solution will run. There's still a connection back to the computer: Each object looks quite a bit like a little computer—it has a state, and it has operations that you can ask it to perform. However, this doesn't seem like such a bad analogy to objects in the real world—they all have characteristics and behaviors.

Alan Kay summarized five basic characteristics of Smalltalk, the first successful object-oriented language and one of the languages upon which Java is based. These characteristics represent a pure approach to object-oriented programming:

1. **Everything is an object.** Think of an object as a fancy variable; it stores data, but you can "make requests" to that object, asking it to perform operations on itself. In theory, you can take any conceptual component in the problem you're trying to solve (dogs, buildings, services, etc.) and represent it as an object in your program.
2. **A program is a bunch of objects telling each other what to do by sending messages**. To make a request of an object, you "send a message" to that object. More concretely, you can think of a message as a request to call a method that belongs to a particular object.
3. **Each object has its own memory made up of other objects**. Put another way, you create a new kind of object by making a package containing existing objects. Thus, you can build complexity into a program while hiding it behind the simplicity of objects.

[1] Some language designers have decided that object-oriented programming by itself is not adequate to easily solve all programming problems, and advocate the combination of various approaches into *multiparadigm* programming languages. See *Multiparadigm Programming in Leda* by Timothy Budd (Addison-Wesley, 1995).

4. **Every object has a type**. Using the parlance, each object is an *instance* of a *class*, in which "class" is synonymous with "type." The most important distinguishing characteristic of a class is "What messages can you send to it?"

5. **All objects of a particular type can receive the same messages**. This is actually a loaded statement, as you will see later. Because an object of type "circle" is also an object of type "shape," a circle is guaranteed to accept shape messages. This means you can write code that talks to shapes and automatically handle anything that fits the description of a shape. This *substitutability* is one of the powerful concepts in OOP.

Booch offers an even more succinct description of an object:

> *An object has state, behavior and identity*.

This means that an object can have internal data (which gives it state), methods (to produce behavior), and each object can be uniquely distinguished from every other object—to put this in a concrete sense, each object has a unique address in memory.[2]

An object has an interface

Aristotle was probably the first to begin a careful study of the concept of *type;* he spoke of "the class of fishes and the class of birds." The idea that all objects, while being unique, are also part of a class of objects that have characteristics and behaviors in common was used directly in the first object-oriented language, Simula-67, with its fundamental keyword **class** that introduces a new type into a program.

Simula, as its name implies, was created for developing simulations such as the classic "bank teller problem." In this, you have numerous tellers, customers, accounts, transactions, and units of money—a lot of "objects." Objects that are identical except for their state during a program's execution are grouped together into "classes of objects," and that's where the keyword **class** came from. Creating abstract data types (classes) is a fundamental concept in object-oriented programming. Abstract data types work almost exactly like built-in types: You can create variables of a type (called *objects* or *instances* in object-oriented parlance) and manipulate those variables (called *sending messages* or *requests;* you send a message and the object figures out what to do with it). The members (elements) of each class share some commonality: Every account has a balance, every teller can accept a deposit, etc. At the same time, each member has its own state: Each account has a different balance, each teller has a name. Thus, the tellers, customers, accounts, transactions, etc., can each be represented with a unique entity in the computer program. This entity is the object, and each object belongs to a particular class that defines its characteristics and behaviors.

So, although what we really do in object-oriented programming is create new data types, virtually all object-oriented programming languages use the "class" keyword. When you see the word "type" think "class" and vice versa.[3]

---

[2] This is actually a bit restrictive, since objects can conceivably exist in different machines and address spaces, and they can also be stored on disk. In these cases, the identity of the object must be determined by something other than memory address.

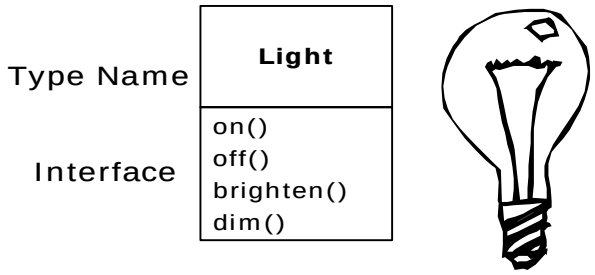
[3] Some people make a distinction, stating that type determines the interface while class is a particular implementation of that interface.

Since a class describes a set of objects that have identical characteristics (data elements) and behaviors (functionality), a class is really a data type because a floating point number, for example, also has a set of characteristics and behaviors. The difference is that a programmer defines a class to fit a problem rather than being forced to use an existing data type that was designed to represent a unit of storage in a machine. You extend the programming language by adding new data types specific to your needs. The programming system welcomes the new classes and gives them all the care and type checking that it gives to built-in types.

The object-oriented approach is not limited to building simulations. Whether or not you agree that any program is a simulation of the system you're designing, the use of OOP techniques can easily reduce a large set of problems to a simple solution.

Once a class is established, you can make as many objects of that class as you like, and then manipulate those objects as if they are the elements that exist in the problem you are trying to solve. Indeed, one of the challenges of object-oriented programming is to create a one-to-one mapping between the elements in the problem space and objects in the solution space.

But how do you get an object to do useful work for you? There needs to be a way to make a request of the object so that it will do something, such as complete a transaction, draw something on the screen, or turn on a switch. And each object can satisfy only certain requests. The requests you can make of an object are defined by its *interface,* and the type is what determines the interface. A simple example might be a representation of a light bulb:




```
Light lt = new Light();
lt.on();
```

The interface determines the requests that you can make for a particular object. However, there must be code somewhere to satisfy that request. This, along with the hidden data, comprises the *implementation*. From a procedural programming standpoint, it's not that complicated. A type has a method associated with each possible request, and when you make a particular request to an object, that method is called. This process is usually summarized by saying that you "send a message" (make a request) to an object, and the object figures out what to do with that message (it executes code).

Here, the name of the type/class is **Light**, the name of this particular **Light** object is **lt**, and the requests that you can make of a **Light** object are to turn it on, turn it off, make it brighter, or make it dimmer. You create a **Light** object by defining a "reference" (**lt**) for that object and calling **new** to request a new object of that type. To send a message to the object, you state the name of the object and connect it to the message request with a period (dot). From the standpoint of the user of a predefined class, that's pretty much all there is to programming with objects.

The preceding diagram follows the format of the *Unified Modeling Language* (UML). Each class is represented by a box, with the type name in the top portion of the box, any *data members* that you care to describe in the middle portion of the box, and the *methods* (the functions that belong

to this object, which receive any messages you send to that object) in the bottom portion of the box. Often, only the name of the class and the public methods are shown in UML design diagrams, so the middle portion is not shown, as in this case. If you're interested only in the class name, then the bottom portion doesn't need to be shown, either.

An object provides services

While you're trying to develop or understand a program design, one of the best ways to think about objects is as "service providers." Your program itself will provide services to the user, and it will accomplish this by using the services offered by other objects. Your goal is to produce (or even better, locate in existing code libraries) a set of objects that provide the ideal services to solve your problem.

A way to start doing this is to ask, "If I could magically pull them out of a hat, what objects would solve my problem right away?" For example, suppose you are creating a bookkeeping program. You might imagine some objects that contain pre-defined bookkeeping input screens, another set of objects that perform bookkeeping calculations, and an object that handles printing of checks and invoices on all different kinds of printers. Maybe some of these objects already exist, and for the ones that don't, what would they look like? What services would *those* objects provide, and what objects would *they* need to fulfill their obligations? If you keep doing this, you will eventually reach a point where you can say either, "That object seems simple enough to sit down and write" or "I'm sure that object must exist already." This is a reasonable way to decompose a problem into a set of objects.

Thinking of an object as a service provider has an additional benefit: It helps to improve the cohesiveness of the object. *High cohesion* is a fundamental quality of software design: It means that the various aspects of a software component (such as an object, although this could also apply to a method or a library of objects) "fit together" well. One problem people have when designing objects is cramming too much functionality into one object. For example, in your check printing module, you may decide you need an object that knows all about formatting and printing. You'll probably discover that this is too much for one object, and that what you need is three or more objects. One object might be a catalog of all the possible check layouts, which can be queried for information about how to print a check. One object or set of objects can be a generic printing interface that knows all about different kinds of printers (but nothing about bookkeeping—this one is a candidate for buying rather than writing yourself). And a third object could use the services of the other two to accomplish the task. Thus, each object has a cohesive set of services it offers. In a good object-oriented design, each object does one thing well, but doesn't try to do too much. This not only allows the discovery of objects that might be purchased (the printer interface object), but it also produces new objects that might be reused somewhere else (the catalog of check layouts).

Treating objects as service providers is a great simplifying tool. This is useful not only during the design process, but also when someone else is trying to understand your code or reuse an object. If they can see the value of the object based on what service it provides, it makes it much easier to fit it into the design.

The hidden implementation

It is helpful to break up the playing field into *class creators* (those who create new data types) and *client programmers*[4] (the class consumers who use the data types in their applications). The

[4] I'm indebted to my friend Scott Meyers for this term.

goal of the client programmer is to collect a toolbox full of classes to use for rapid application development. The goal of the class creator is to build a class that exposes only what's necessary to the client programmer and keeps everything else hidden. Why? Because if it's hidden, the client programmer can't access it, which means that the class creator can change the hidden portion at will without worrying about the impact on anyone else. The hidden portion usually represents the tender insides of an object that could easily be corrupted by a careless or uninformed client programmer, so hiding the implementation reduces program bugs.

In any relationship it's important to have boundaries that are respected by all parties involved. When you create a library, you establish a relationship with the client programmer, who is also a programmer, but one who is putting together an application by using your library, possibly to build a bigger library. If all the members of a class are available to everyone, then the client programmer can do anything with that class and there's no way to enforce rules. Even though you might really prefer that the client programmer not directly manipulate some of the members of your class, without access control there's no way to prevent it. Everything's naked to the world.

So the first reason for access control is to keep client programmers' hands off portions they shouldn't touch—parts that are necessary for the internal operation of the data type but not part of the interface that users need in order to solve their particular problems. This is actually a service to client programmers because they can easily see what's important to them and what they can ignore.

The second reason for access control is to allow the library designer to change the internal workings of the class without worrying about how it will affect the client programmer. For example, you might implement a particular class in a simple fashion to ease development, and then later discover that you need to rewrite it in order to make it run faster. If the interface and implementation are clearly separated and protected, you can accomplish this easily.

Java uses three explicit keywords to set the boundaries in a class: **public**, **private**, and **protected**. These *access specifiers* determine who can use the definitions that follow. **public** means the following element is available to everyone. The **private** keyword, on the other hand, means that no one can access that element except you, the creator of the type, inside methods of that type. **private** is a brick wall between you and the client programmer. Someone who tries to access a **private** member will get a compile-time error. The **protected** keyword acts like **private**, with the exception that an inheriting class has access to **protected** members, but not **private** members. Inheritance will be introduced shortly.

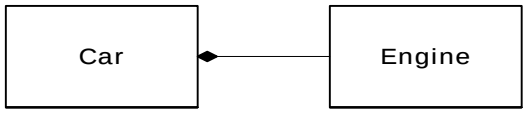
Java also has a "default" access, which comes into play if you don't use one of the aforementioned specifiers. This is usually called *package access* because classes can access the members of other classes in the same *package* (library component), but outside of the package those same members appear to be **private**.

Reusing the implementation

Once a class has been created and tested, it should (ideally) represent a useful unit of code. It turns out that this reusability is not nearly so easy to achieve as many would hope; it takes experience and insight to produce a reusable object design. But once you have such a design, it begs to be reused. Code reuse is one of the greatest advantages that object-oriented programming languages provide.

The simplest way to reuse a class is to just use an object of that class directly, but you can also place an object of that class inside a new class. We call this "creating a member object." Your new class can be made up of any number and type of other objects, in any combination that you need

to achieve the functionality desired in your new class. Because you are composing a new class from existing classes, this concept is called *composition* (if the composition happens dynamically, it's usually called *aggregation*). Composition is often referred to as a "has-a" relationship, as in "A car has an engine."



(This UML diagram indicates composition with the filled diamond, which states there is one car. I will typically use a simpler form: just a line, without the diamond, to indicate an association.[5])

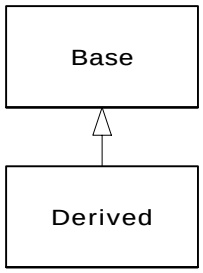
Composition comes with a great deal of flexibility. The member objects of your new class are typically private, making them inaccessible to the client programmers who are using the class. This allows you to change those members without disturbing existing client code. You can also change the member objects at run time, to dynamically change the behavior of your program. Inheritance, which is described next, does not have this flexibility since the compiler must place compile-time restrictions on classes created with inheritance.

Because inheritance is so important in object-oriented programming, it is often highly emphasized, and the new programmer can get the idea that inheritance should be used everywhere. This can result in awkward and overly complicated designs. Instead, you should first look to composition when creating new classes, since it is simpler and more flexible. If you take this approach, your designs will be cleaner. Once you've had some experience, it will be reasonably obvious when you need inheritance.

Inheritance

By itself, the idea of an object is a convenient tool. It allows you to package data and functionality together by *concept*, so you can represent an appropriate problem-space idea rather than being forced to use the idioms of the underlying machine. These concepts are expressed as fundamental units in the programming language by using the **class** keyword.

It seems a pity, however, to go to all the trouble to create a class and then be forced to create a brand new one that might have similar functionality. It's nicer if we can take the existing class, clone it, and then make additions and modifications to the clone. This is effectively what you get with *inheritance*, with the exception that if the original class (called the *base class* or *superclass* or *parent class*) is changed, the modified "clone" (called the *derived class* or *inherited class* or *subclass* or *child class*) also reflects those changes.



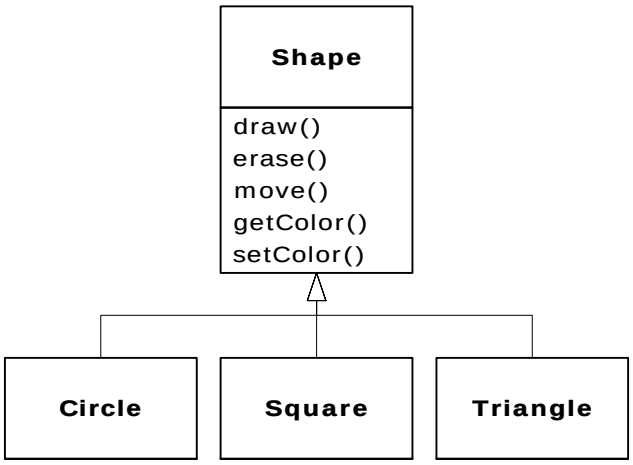
[5] This is usually enough detail for most diagrams, and you don't need to get specific about whether you're using aggregation or composition.

(The arrow in this UML diagram points from the derived class to the base class. As you will see, there is commonly more than one derived class.)

A type does more than describe the constraints on a set of objects; it also has a relationship with other types. Two types can have characteristics and behaviors in common, but one type may contain more characteristics than another and may also handle more messages (or handle them differently). Inheritance expresses this similarity between types by using the concept of base types and derived types. A base type contains all of the characteristics and behaviors that are shared among the types derived from it. You create a base type to represent the core of your ideas about some objects in your system. From the base type, you derive other types to express the different ways that this core can be realized.

For example, a trash-recycling machine sorts pieces of trash. The base type is "trash," and each piece of trash has a weight, a value, and so on, and can be shredded, melted, or decomposed. From this, more specific types of trash are derived that may have additional characteristics (a bottle has a color) or behaviors (an aluminum can may be crushed, a steel can is magnetic). In addition, some behaviors may be different (the value of paper depends on its type and condition). Using inheritance, you can build a type hierarchy that expresses the problem you're trying to solve in terms of its types.

A second example is the classic "shape" example, perhaps used in a computer-aided design system or game simulation. The base type is "shape," and each shape has a size, a color, a position, and so on. Each shape can be drawn, erased, moved, colored, etc. From this, specific types of shapes are derived (inherited)—circle, square, triangle, and so on—each of which may have additional characteristics and behaviors. Certain shapes can be flipped, for example. Some behaviors may be different, such as when you want to calculate the area of a shape. The type hierarchy embodies both the similarities and differences between the shapes.

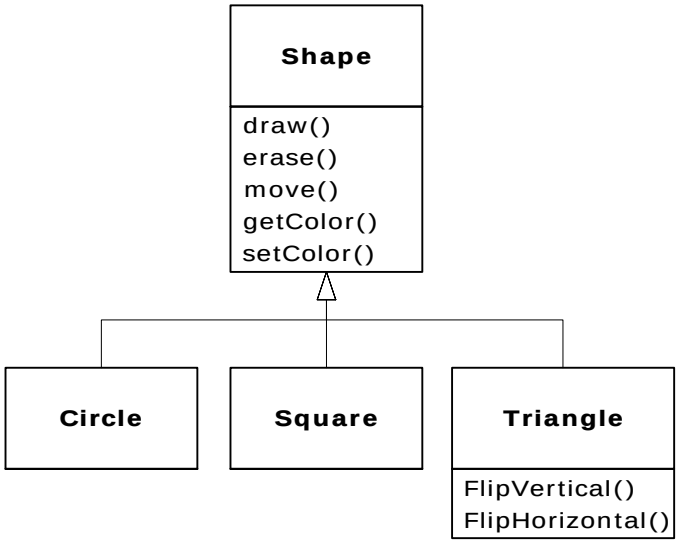


Casting the solution in the same terms as the problem is very useful because you don't need a lot of intermediate models to get from a description of the problem to a description of the solution. With objects, the type hierarchy is the primary model, so you go directly from the description of the system in the real world to the description of the system in code. Indeed, one of the difficulties people have with object-oriented design is that it's too simple to get from the beginning to the end. A mind trained to look for complex solutions can initially be stumped by this simplicity.

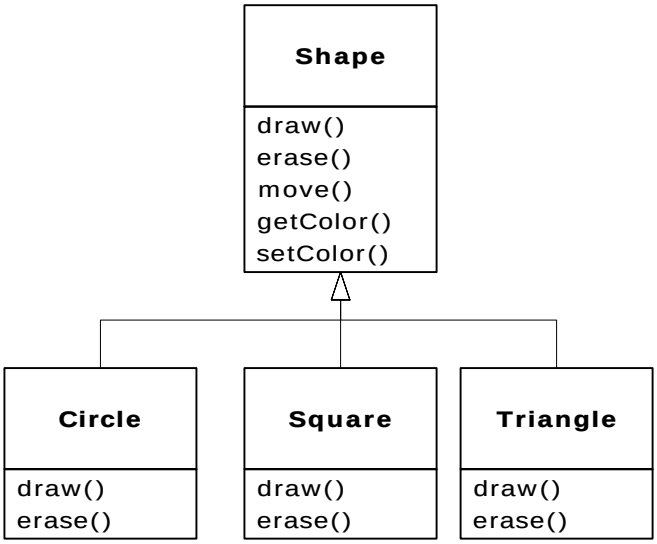
When you inherit from an existing type, you create a new type. This new type contains not only all the members of the existing type (although the **private** ones are hidden away and inaccessible), but more importantly it duplicates the interface of the base class. That is, all the messages you can send to objects of the base class you can also send to objects of the derived class. Since we know the type of a class by the messages we can send to it, this means that the derived class *is the same type as the base class*. In the previous example, "A circle is a shape." This type equivalence via inheritance is one of the fundamental gateways in understanding the meaning of object-oriented programming.

Since both the base class and derived class have the same fundamental interface, there must be some implementation to go along with that interface. That is, there must be some code to execute when an object receives a particular message. If you simply inherit a class and don't do anything else, the methods from the base-class interface come right along into the derived class. That means objects of the derived class have not only the same type, they also have the same behavior, which isn't particularly interesting.

You have two ways to differentiate your new derived class from the original base class. The first is quite straightforward: You simply add brand new methods to the derived class. These new methods are not part of the base-class interface. This means that the base class simply didn't do as much as you wanted it to, so you added more methods. This simple and primitive use for inheritance is, at times, the perfect solution to your problem. However, you should look closely for the possibility that your base class might also need these additional methods. This process of discovery and iteration of your design happens regularly in object-oriented programming.



Although inheritance may sometimes imply (especially in Java, where the keyword for inheritance is **extends**) that you are going to add new methods to the interface, that's not necessarily true. The second and more important way to differentiate your new class is to *change* the behavior of an existing base-class method. This is referred to as *overriding* that method.

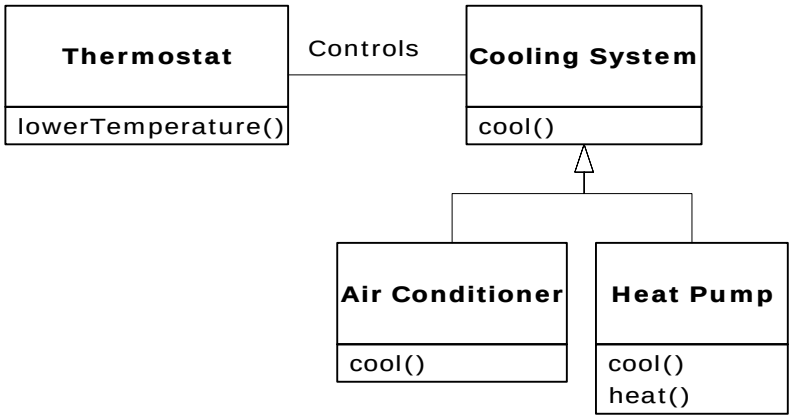


To override a method, you simply create a new definition for the method in the derived class. You're saying, "I'm using the same interface method here, but I want it to do something different for my new type."

Is-a vs. is-like-a relationships

There's a certain debate that can occur about inheritance: Should inheritance override *only* base-class methods (and not add new methods that aren't in the base class)? This would mean that the derived class is *exactly* the same type as the base class since it has exactly the same interface. As a result, you can exactly substitute an object of the derived class for an object of the base class. This can be thought of as *pure substitution*, and it's often referred to as the *substitution principle*. In a sense, this is the ideal way to treat inheritance. We often refer to the relationship between the base class and derived classes in this case as an *is-a* relationship, because you can say, "A circle *is a* shape." A test for inheritance is to determine whether you can state the is-a relationship about the classes and have it make sense.

There are times when you must add new interface elements to a derived type, thus extending the interface. The new type can still be substituted for the base type, but the substitution isn't perfect because your new methods are not accessible from the base type. This can be described as an *is-like-a* relationship (my term). The new type has the interface of the old type but it also contains other methods, so you can't really say it's exactly the same. For example, consider an air conditioner. Suppose your house is wired with all the controls for cooling; that is, it has an interface that allows you to control cooling. Imagine that the air conditioner breaks down and you replace it with a heat pump, which can both heat and cool. The heat pump *is-like-an* air conditioner, but it can do more. Because the control system of your house is designed only to control cooling, it is restricted to communication with the cooling part of the new object. The interface of the new object has been extended, and the existing system doesn't know about anything except the original interface.



Of course, once you see this design it becomes clear that the base class "cooling system" is not general enough, and should be renamed to "temperature control system" so that it can also include heating—at which point the substitution principle will work. However, this diagram is an example of what can happen with design in the real world.

When you see the substitution principle it's easy to feel like this approach (pure substitution) is the only way to do things, and in fact it *is* nice if your design works out that way. But you'll find that there are times when it's equally clear that you must add new methods to the interface of a derived class. With inspection both cases should be reasonably obvious.

Interchangeable objects with polymorphism

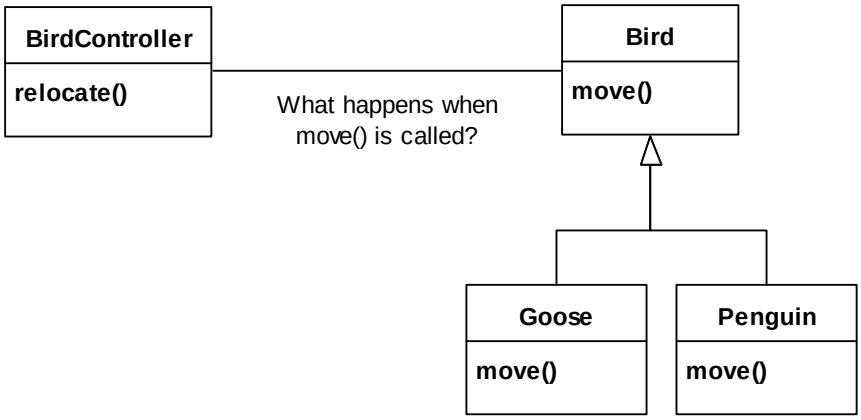
When dealing with type hierarchies, you often want to treat an object not as the specific type that it is, but instead as its base type. This allows you to write code that doesn't depend on specific types. In the shape example, methods manipulate generic shapes, unconcerned about whether they're circles, squares, triangles, or some shape that hasn't even been defined yet. All shapes can be drawn, erased, and moved, so these methods simply send a message to a shape object; they don't worry about how the object copes with the message.

Such code is unaffected by the addition of new types, and adding new types is the most common way to extend an object-oriented program to handle new situations. For example, you can derive a new subtype of shape called pentagon without modifying the methods that deal only with generic shapes. This ability to easily extend a design by deriving new subtypes is one of the essential ways to encapsulate change. This greatly improves designs while reducing the cost of software maintenance.

There's a problem, however, with attempting to treat derived-type objects as their generic base types (circles as shapes, bicycles as vehicles, cormorants as birds, etc.). If a method is going to tell a generic shape to draw itself, or a generic vehicle to steer, or a generic bird to move, the compiler cannot know at compile time precisely what piece of code will be executed. That's the whole point—when the message is sent, the programmer doesn't *want* to know what piece of code will be executed; the draw method can be applied equally to a circle, a square, or a triangle, and the object will execute the proper code depending on its specific type.

If you don't have to know what piece of code will be executed, then when you add a new subtype, the code it executes can be different without requiring changes to the method that calls it. Therefore, the compiler cannot know precisely what piece of code is executed, so what does it do?

For example, in the following diagram the **BirdController** object just works with generic **Bird** objects and does not know what exact type they are. This is convenient from **BirdController**'s perspective because it doesn't have to write special code to determine the exact type of **Bird** it's working with or that **Bird**'s behavior. So how does it happen that, when **move( )** is called while ignoring the specific type of **Bird**, the right behavior will occur (a **Goose** walks, flies, or swims, and a **Penguin** walks or swims)?



The answer is the primary twist in object-oriented programming: The compiler cannot make a function call in the traditional sense. The function call generated by a non-OOP compiler causes what is called *early binding*, a term you may not have heard before because you've never thought about it any other way. It means the compiler generates a call to a specific function name, and the runtime system resolves this call to the absolute address of the code to be executed. In OOP, the program cannot determine the address of the code until run time, so some other scheme is necessary when a message is sent to a generic object.

To solve the problem, object-oriented languages use the concept of *late binding*. When you send a message to an object, the code being called isn't determined until run time. The compiler does ensure that the method exists and performs type checking on the arguments and return value, but it doesn't know the exact code to execute.

To perform late binding, Java uses a special bit of code in lieu of the absolute call. This code calculates the address of the method body, using information stored in the object (this process is covered in great detail in the *Polymorphism* chapter). Thus, each object can behave differently according to the contents of that special bit of code. When you send a message to an object, the object actually does figure out what to do with that message.

In some languages you must explicitly state that you want a method to have the flexibility of late-binding properties (C++ uses the **virtual** keyword to do this). In these languages, by default, methods are *not* dynamically bound. In Java, dynamic binding is the default behavior and you don't need to remember to add any extra keywords in order to get polymorphism.

Consider the shape example. The family of classes (all based on the same uniform interface) was diagrammed earlier in this chapter. To demonstrate polymorphism, we want to write a single piece of code that ignores the specific details of type and talks only to the base class. That code is *decoupled* from type-specific information and thus is simpler to write and easier to understand. And, if a new type—a **Hexagon**, for example—is added through inheritance, the code you write will work just as well for the new type of **Shape** as it did on the existing types. Thus, the program is *extensible*.

If you write a method in Java (as you will soon learn how to do):

```
void doSomething(Shape shape) {
  shape.erase();
  // ...
  shape.draw();
}
```

This method speaks to any **Shape**, so it is independent of the specific type of object that it's drawing and erasing. If some other part of the program uses the **doSomething( )** method:

```
Circle circle = new Circle();
Triangle triangle = new Triangle();
Line line = new Line();
doSomething(circle);
doSomething(triangle);
doSomething(line);
```

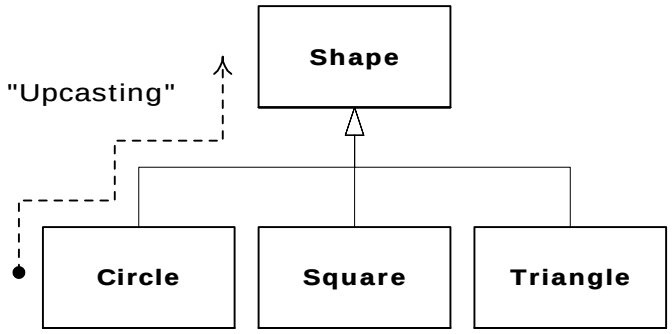
The calls to **doSomething( )** automatically work correctly, regardless of the exact type of the object.

This is a rather amazing trick. Consider the line:

```
doSomething(circle);
```

What's happening here is that a **Circle** is being passed into a method that's expecting a **Shape**. Since a **Circle** *is* a **Shape** it can be treated as one by **doSomething( )**. That is, any message that **doSomething( )** can send to a **Shape**, a **Circle** can accept. So it is a completely safe and logical thing to do.

We call this process of treating a derived type as though it were its base type *upcasting*. The name *cast* is used in the sense of casting into a mold and the *up* comes from the way the inheritance diagram is typically arranged, with the base type at the top and the derived classes fanning out downward. Thus, casting to a base type is moving up the inheritance diagram: "upcasting."



An object-oriented program contains some upcasting somewhere, because that's how you decouple yourself from knowing about the exact type you're working with. Look at the code in **doSomething( )**:

```
  shape.erase();
  // ...
  shape.draw();
```

Notice that it doesn't say, "If you're a **Circle**, do this, if you're a **Square**, do that, etc." If you write that kind of code, which checks for all the possible types that a **Shape** can actually be, it's

messy and you need to change it every time you add a new kind of **Shape**. Here, you just say, "You're a shape, I know you can **erase( )** and **draw( )** yourself, do it, and take care of the details correctly."

What's impressive about the code in **doSomething( )** is that, somehow, the right thing happens. Calling **draw( )** for **Circle** causes different code to be executed than when calling **draw( )** for a **Square** or a **Line**, but when the **draw( )** message is sent to an anonymous **Shape**, the correct behavior occurs based on the actual type of the **Shape**. This is amazing because, as mentioned earlier, when the Java compiler is compiling the code for **doSomething( )**, it cannot know exactly what types it is dealing with. So ordinarily, you'd expect it to end up calling the version of **erase( )** and **draw( )** for the base class **Shape**, and not for the specific **Circle**, **Square**, or **Line**. And yet the right thing happens because of polymorphism. The compiler and runtime system handle the details; all you need to know right now is that it does happen, and more importantly, how to design with it. When you send a message to an object, the object will do the right thing, even when upcasting is involved.

The singly rooted hierarchy

One of the issues in OOP that has become especially prominent since the introduction of C++ is whether all classes should ultimately be inherited from a single base class. In Java (as with virtually all other OOP languages *except* for C++) the answer is yes, and the name of this ultimate base class is simply **Object**. It turns out that the benefits of the singly rooted hierarchy are many.

All objects in a singly rooted hierarchy have an interface in common, so they are all ultimately the same fundamental type. The alternative (provided by C++) is that you don't know that everything is the same basic type. From a backward-compatibility standpoint this fits the model of C better and can be thought of as less restrictive, but when you want to do full-on object-oriented programming you must then build your own hierarchy to provide the same convenience that's built into other OOP languages. And in any new class library you acquire, some other incompatible interface will be used. It requires effort (and possibly multiple inheritance) to work the new interface into your design. Is the extra "flexibility" of C++ worth it? If you need it—if you have a large investment in C—it's quite valuable. If you're starting from scratch, other alternatives such as Java can often be more productive.

All objects in a singly rooted hierarchy can be guaranteed to have certain functionality. You know you can perform certain basic operations on every object in your system. All objects can easily be created on the heap, and argument passing is greatly simplified.

A singly rooted hierarchy makes it much easier to implement a *garbage collector*, which is one of the fundamental improvements of Java over C++. And since information about the type of an object is guaranteed to be in all objects, you'll never end up with an object whose type you cannot determine. This is especially important with system-level operations, such as exception handling, and to allow greater flexibility in programming.

Containers

In general, you don't know how many objects you're going to need to solve a particular problem, or how long they will last. You also don't know how to store those objects. How can you know how much space to create if that information isn't known until run time?

The solution to most problems in object-oriented design seems flippant: You create another type of object. The new type of object that solves this particular problem holds references to other objects. Of course, you can do the same thing with an *array*, which is available in most

languages. But this new object, generally called a *container* (also called a *collection*, but the Java library uses that term in a different sense so this book will use "container"), will expand itself whenever necessary to accommodate everything you place inside it. So you don't need to know how many objects you're going to hold in a container. Just create a container object and let it take care of the details.

Fortunately, a good OOP language comes with a set of containers as part of the package. In C++, it's part of the Standard C++ Library and is often called the *Standard Template Library* (STL). Smalltalk has a very complete set of containers. Java also has numerous containers in its standard library. In some libraries, one or two generic containers is considered good enough for all needs, and in others (Java, for example) the library has different types of containers for different needs: several different kinds of **List** classes (to hold sequences), **Map**s (also known as *associative arrays*, to associate objects with other objects), **Set**s (to hold one of each type of object), and more components such as queues, trees, stacks, etc.

From a design standpoint, all you really want is a container that can be manipulated to solve your problem. If a single type of container satisfied all of your needs, there'd be no reason to have different kinds. There are two reasons that you need a choice of containers. First, containers provide different types of interfaces and external behavior. A stack has a different interface and behavior than a queue, which is different from a set or a list. One of these might provide a more flexible solution to your problem than the other. Second, different containers have different efficiencies for certain operations. For example, there are two basic types of **List**: **ArrayList** and **LinkedList**. Both are simple sequences that can have identical interfaces and external behaviors. But certain operations can have significantly different costs. Randomly accessing elements in an **ArrayList** is a constant-time operation; it takes the same amount of time regardless of the element you select. However, in a **LinkedList** it is expensive to move through the list to randomly select an element, and it takes longer to find an element that is farther down the list. On the other hand, if you want to insert an element in the middle of a sequence, it's cheaper in a **LinkedList** than in an **ArrayList**. These and other operations have different efficiencies depending on the underlying structure of the sequence. You might start building your program with a **LinkedList** and, when tuning for performance, change to an **ArrayList**. Because of the abstraction via the interface **List**, you can change from one to the other with minimal impact on your code.

Parameterized types (generics)

Before Java SE5, containers held the one universal type in Java: **Object**. The singly rooted hierarchy means that everything is an **Object**, so a container that holds **Object**s can hold anything.[6] This made containers easy to reuse.

To use such a container, you simply add object references to it and later ask for them back. But, since the container held only **Object**s, when you added an object reference into the container it was upcast to **Object**, thus losing its character. When fetching it back, you got an **Object** reference, and not a reference to the type that you put in. So how do you turn it back into something that has the specific type of the object that you put into the container?

Here, the cast is used again, but this time you're not casting up the inheritance hierarchy to a more general type. Instead, you cast down the hierarchy to a more specific type. This manner of casting is called *downcasting*. With upcasting, you know, for example, that a **Circle** is a type of

[6] They do not hold primitives, but Java SE5 *autoboxing* makes this restriction almost a non-issue. This is discussed in detail later in the book.

Shape so it's safe to upcast, but you don't know that an **Object** is necessarily a **Circle** or a **Shape** so it's hardly safe to downcast unless you know exactly what you're dealing with.

It's not completely dangerous, however, because if you downcast to the wrong thing you'll get a runtime error called an *exception,* which will be described shortly. When you fetch object references from a container, though, you must have some way to remember exactly what they are so you can perform a proper downcast.

Downcasting and the runtime checks require extra time for the running program and extra effort from the programmer. Wouldn't it make sense to somehow create the container so that it knows the types that it holds, eliminating the need for the downcast and a possible mistake? The solution is called a *parameterized type* mechanism. A parameterized type is a class that the compiler can automatically customize to work with particular types. For example, with a parameterized container, the compiler could customize that container so that it would accept only **Shape**s and fetch only **Shape**s.

One of the big changes in Java SE5 is the addition of parameterized types, called *generics* in Java. You'll recognize the use of generics by the angle brackets with types inside; for example, an **ArrayList** that holds **Shape** can be created like this:

```
ArrayList<Shape> shapes = new ArrayList<Shape>();
```

There have also been changes to many of the standard library components in order to take advantage of generics. As you will see, generics have an impact on much of the code in this book.

Object creation & lifetime

One critical issue when working with objects is the way they are created and destroyed. Each object requires resources, most notably memory, in order to exist. When an object is no longer needed it must be cleaned up so that these resources are released for reuse. In simple programming situations the question of how an object is cleaned up doesn't seem too challenging: You create the object, use it for as long as it's needed, and then it should be destroyed. However, it's not hard to encounter situations that are more complex.

Suppose, for example, you are designing a system to manage air traffic for an airport. (The same model might also work for managing crates in a warehouse, or a video rental system, or a kennel for boarding pets.) At first it seems simple: Make a container to hold airplanes, then create a new airplane and place it in the container for each airplane that enters the air-traffic-control zone. For cleanup, simply clean up the appropriate airplane object when a plane leaves the zone.

But perhaps you have some other system to record data about the planes; perhaps data that doesn't require such immediate attention as the main controller function. Maybe it's a record of the flight plans of all the small planes that leave the airport. So you have a second container of small planes, and whenever you create a plane object you also put it in this second container if it's a small plane. Then some background process performs operations on the objects in this container during idle moments.

Now the problem is more difficult: How can you possibly know when to destroy the objects? When you're done with the object, some other part of the system might not be. This same problem can arise in a number of other situations, and in programming systems (such as C++) in which you must explicitly delete an object when you're done with it this can become quite complex.

Where is the data for an object and how is the lifetime of the object controlled? C++ takes the approach that control of efficiency is the most important issue, so it gives the programmer a choice. For maximum runtime speed, the storage and lifetime can be determined while the program is being written, by placing the objects on the stack (these are sometimes called *automatic* or *scoped* variables) or in the static storage area. This places a priority on the speed of storage allocation and release, and this control can be very valuable in some situations. However, you sacrifice flexibility because you must know the exact quantity, lifetime, and type of objects while you're writing the program. If you are trying to solve a more general problem such as computer-aided design, warehouse management, or air-traffic control, this is too restrictive.

The second approach is to create objects dynamically in a pool of memory called the heap. In this approach, you don't know until run time how many objects you need, what their lifetime is, or what their exact type is. Those are determined at the spur of the moment while the program is running. If you need a new object, you simply make it on the heap at the point that you need it. Because the storage is managed dynamically, at run time, the amount of time required to allocate storage on the heap can be noticeably longer than the time to create storage on the stack. Creating storage on the stack is often a single assembly instruction to move the stack pointer down and another to move it back up. The time to create heap storage depends on the design of the storage mechanism.

The dynamic approach makes the generally logical assumption that objects tend to be complicated, so the extra overhead of finding storage and releasing that storage will not have an important impact on the creation of an object. In addition, the greater flexibility is essential to solve the general programming problem.

Java uses dynamic memory allocation, exclusively.[7] Every time you want to create an object, you use the **new** operator to build a dynamic instance of that object.

There's another issue, however, and that's the lifetime of an object. With languages that allow objects to be created on the stack, the compiler determines how long the object lasts and can automatically destroy it. However, if you create it on the heap the compiler has no knowledge of its lifetime. In a language like C++, you must determine programmatically when to destroy the object, which can lead to memory leaks if you don't do it correctly (and this is a common problem in C++ programs). Java provides a feature called a *garbage collector* that automatically discovers when an object is no longer in use and destroys it. A garbage collector is much more convenient because it reduces the number of issues that you must track and the code you must write. More importantly, the garbage collector provides a much higher level of insurance against the insidious problem of memory leaks, which has brought many a C++ project to its knees.

With Java, the garbage collector is designed to take care of the problem of releasing the memory (although this doesn't include other aspects of cleaning up an object). The garbage collector "knows" when an object is no longer in use, and it then automatically releases the memory for that object. This, combined with the fact that all objects are inherited from the single root class **Object** and that you can create objects only one way—on the heap—makes the process of programming in Java much simpler than programming in C++. You have far fewer decisions to make and hurdles to overcome.

[7] Primitive types, which you'll learn about later, are a special case.

Exception handling: dealing with errors

Ever since the beginning of programming languages, error handling has been a particularly difficult issue. Because it's so hard to design a good error-handling scheme, many languages simply ignore the issue, passing the problem on to library designers who come up with halfway measures that work in many situations but that can easily be circumvented, generally by just ignoring them. A major problem with most error-handling schemes is that they rely on programmer vigilance in following an agreed-upon convention that is not enforced by the language. If the programmer is not vigilant—often the case if they are in a hurry—these schemes can easily be forgotten.

Exception handling wires error handling directly into the programming language and sometimes even the operating system. An exception is an object that is "thrown" from the site of the error and can be "caught" by an appropriate exception handler designed to handle that particular type of error. It's as if exception handling is a different, parallel path of execution that can be taken when things go wrong. And because it uses a separate execution path, it doesn't need to interfere with your normally executing code. This tends to make that code simpler to write because you aren't constantly forced to check for errors. In addition, a thrown exception is unlike an error value that's returned from a method or a flag that's set by a method in order to indicate an error condition—these can be ignored. An exception cannot be ignored, so it's guaranteed to be dealt with at some point. Finally, exceptions provide a way to reliably recover from a bad situation. Instead of just exiting the program, you are often able to set things right and restore execution, which produces much more robust programs.

Java's exception handling stands out among programming languages, because in Java, exception handling was wired in from the beginning and you're forced to use it. It is the single acceptable way to report errors. If you don't write your code to properly handle exceptions, you'll get a compile-time error message. This guaranteed consistency can sometimes make error handling much easier.

It's worth noting that exception handling isn't an object-oriented feature, although in object-oriented languages the exception is normally represented by an object. Exception handling existed before object-oriented languages.

Concurrent programming

A fundamental concept in computer programming is the idea of handling more than one task at a time. Many programming problems require that the program stop what it's doing, deal with some other problem, and then return to the main process. The solution has been approached in many ways. Initially, programmers with low-level knowledge of the machine wrote interrupt service routines, and the suspension of the main process was initiated through a hardware interrupt. Although this worked well, it was difficult and non-portable, so it made moving a program to a new type of machine slow and expensive.

Sometimes, interrupts are necessary for handling time-critical tasks, but there's a large class of problems in which you're simply trying to partition the problem into separately running pieces (tasks) so that the whole program can be more responsive. Within a program, these separately running pieces are called *threads*, and the general concept is called *concurrency*. A common example of concurrency is the user interface. By using tasks, a user can press a button and get a quick response rather than being forced to wait until the program finishes its current task.

Ordinarily, tasks are just a way to allocate the time of a single processor. But if the operating system supports multiple processors, each task can be assigned to a different processor, and they can truly run in parallel. One of the convenient features of concurrency at the language level is that the programmer doesn't need to worry about whether there are many processors or just one. The program is logically divided into tasks, and if the machine has more than one processor, then the program runs faster, without any special adjustments.

All this makes concurrency sound pretty simple. There is a catch: shared resources. If you have more than one task running that's expecting to access the same resource, you have a problem. For example, two processes can't simultaneously send information to a printer. To solve the problem, resources that can be shared, such as the printer, must be locked while they are being used. So a task locks a resource, completes its task, and then releases the lock so that someone else can use the resource.

Java's concurrency is built into the language, and Java SE5 has added significant additional library support.

Java and the Internet

If Java is, in fact, yet another computer programming language, you may question why it is so important and why it is being promoted as a revolutionary step in computer programming. The answer isn't immediately obvious if you're coming from a traditional programming perspective. Although Java is very useful for solving traditional standalone programming problems, it is also important because it solves programming problems for the World Wide Web.

What is the Web?

The Web can seem a bit of a mystery at first, with all this talk of "surfing," "presence," and "home pages." It's helpful to step back and see what it really is, but to do this you must understand client/server systems, another aspect of computing that's full of confusing issues.

Client/server computing

The primary idea of a client/server system is that you have a central repository of information—some kind of data, usually in a database—that you want to distribute on demand to some set of people or machines. A key to the client/server concept is that the repository of information is centrally located so that it can be changed and so that those changes will propagate out to the information consumers. Taken together, the information repository, the software that distributes the information, and the machine(s) where the information and software reside are called "the server." The software that resides on the consumer machine, communicates with the server, fetches the information, processes it, and then displays it on the consumer machine is called the *client*.

The basic concept of client/server computing, then, is not so complicated. The problems arise because you have a single server trying to serve many clients at once. Generally, a database management system is involved, so the designer "balances" the layout of data into tables for optimal use. In addition, systems often allow a client to insert new information into a server. This means you must ensure that one client's new data doesn't walk over another client's new data, or that data isn't lost in the process of adding it to the database (this is called transaction processing). As client software changes, it must be built, debugged, and installed on the client machines, which turns out to be more complicated and expensive than you might think. It's especially problematic to support multiple types of computers and operating systems. Finally, there's the all-important performance issue: You might have hundreds of clients making requests

of your server at any moment, so a small delay can be critical. To minimize latency, programmers work hard to offload processing tasks, often to the client machine, but sometimes to other machines at the server site, using so-called *middleware*. (Middleware is also used to improve maintainability.)

The simple idea of distributing information has so many layers of complexity that the whole problem can seem hopelessly enigmatic. And yet it's crucial: Client/server computing accounts for roughly half of all programming activities. It's responsible for everything from taking orders and credit-card transactions to the distribution of any kind of data—stock market, scientific, government, you name it. What we've come up with in the past is individual solutions to individual problems, inventing a new solution each time. These were hard to create and hard to use, and the user had to learn a new interface for each one. The entire client/server problem needed to be solved in a big way.

The Web as a giant server

The Web is actually one giant client/server system. It's a bit worse than that, since you have all the servers and clients coexisting on a single network at once. You don't need to know that, because all you care about is connecting to and interacting with one server at a time (even though you might be hopping around the world in your search for the correct server).

Initially it was a simple one-way process. You made a request of a server and it handed you a file, which your machine's browser software (i.e., the client) would interpret by formatting onto your local machine. But in short order people began wanting to do more than just deliver pages from a server. They wanted full client/server capability so that the client could feed information back to the server, for example, to do database lookups on the server, to add new information to the server, or to place an order (which requires special security measures). These are the changes we've been seeing in the development of the Web.

The Web browser was a big step forward: the concept that one piece of information can be displayed on any type of computer without change. However, the original browsers were still rather primitive and rapidly bogged down by the demands placed on them. They weren't particularly interactive, and tended to clog up both the server and the Internet because whenever you needed to do something that required programming you had to send information back to the server to be processed. It could take many seconds or minutes to find out you had misspelled something in your request. Since the browser was just a viewer it couldn't perform even the simplest computing tasks. (On the other hand, it was safe, because it couldn't execute any programs on your local machine that might contain bugs or viruses.)

To solve this problem, different approaches have been taken. To begin with, graphics standards have been enhanced to allow better animation and video within browsers. The remainder of the problem can be solved only by incorporating the ability to run programs on the client end, under the browser. This is called *client-side programming*.

Client-side programming

The Web's initial server-browser design provided for interactive content, but the interactivity was completely provided by the server. The server produced static pages for the client browser, which would simply interpret and display them. Basic *HyperText Markup Language* (HTML) contains simple mechanisms for data gathering: text-entry boxes, check boxes, radio boxes, lists and drop-down lists, as well as a button that could only be programmed to reset the data on the form or "submit" the data on the form back to the server. This submission passes through the *Common Gateway Interface* (CGI) provided on all Web servers. The text within the submission tells CGI

what to do with it. The most common action is to run a program located on the server in a directory that's typically called "cgi-bin." (If you watch the address window at the top of your browser when you push a button on a Web page, you can sometimes see "cgi-bin" within all the gobbledygook there.) These programs can be written in most languages. Perl has been a common choice because it is designed for text manipulation and is interpreted, so it can be installed on any server regardless of processor or operating system. However, Python (*www.Python.org*) has been making inroads because of its greater power and simplicity.

Many powerful Web sites today are built strictly on CGI, and you can in fact do nearly anything with CGI. However, Web sites built on CGI programs can rapidly become overly complicated to maintain, and there is also the problem of response time. The response of a CGI program depends on how much data must be sent, as well as the load on both the server and the Internet. (On top of this, starting a CGI program tends to be slow.) The initial designers of the Web did not foresee how rapidly this bandwidth would be exhausted for the kinds of applications people developed. For example, any sort of dynamic graphing is nearly impossible to perform with consistency because a *Graphics Interchange Format* (GIF) file must be created and moved from the server to the client for each version of the graph. In addition, you've no doubt experienced the process of data validation for a Web input form. You press the submit button on a page; the data is shipped back to the server; the server starts a CGI program that discovers an error, formats an HTML page informing you of the error, and then sends the page back to you; you must then back up a page and try again. Not only is this slow, it's inelegant.

The solution is client-side programming. Most desktop computers that run Web browsers are powerful engines capable of doing vast work, and with the original static HTML approach they are sitting there, just idly waiting for the server to dish up the next page. Client-side programming means that the Web browser is harnessed to do whatever work it can, and the result for the user is a much speedier and more interactive experience at your Web site.

The problem with discussions of client-side programming is that they aren't very different from discussions of programming in general. The parameters are almost the same, but the platform is different; a Web browser is like a limited operating system. In the end, you must still program, and this accounts for the dizzying array of problems and solutions produced by client-side programming. The rest of this section provides an overview of the issues and approaches in client-side programming.

Plug-ins

One of the most significant steps forward in client-side programming is the development of the plug-in. This is a way for a programmer to add new functionality to the browser by downloading a piece of code that plugs itself into the appropriate spot in the browser. It tells the browser, "From now on you can perform this new activity." (You need to download the plug-in only once.) Some fast and powerful behavior is added to browsers via plug-ins, but writing a plug-in is not a trivial task, and isn't something you'd want to do as part of the process of building a particular site. The value of the plug-in for client-side programming is that it allows an expert programmer to develop extensions and add those extensions to a browser without the permission of the browser manufacturer. Thus, plug-ins provide a "back door" that allows the creation of new client-side programming languages (although not all languages are implemented as plug-ins).

Scripting languages

Plug-ins resulted in the development of browser scripting languages. With a scripting language, you embed the source code for your client-side program directly into the HTML page, and the plug-in that interprets that language is automatically activated while the HTML page is being

displayed. Scripting languages tend to be reasonably easy to understand and, because they are simply text that is part of an HTML page, they load very quickly as part of the single server hit required to procure that page. The trade-off is that your code is exposed for everyone to see (and steal). Generally, however, you aren't doing amazingly sophisticated things with scripting languages, so this is not too much of a hardship.

One scripting language that you can expect a Web browser to support *without* a plug-in is JavaScript (this has only a passing resemblance to Java and you'll have to climb an additional learning curve to use it. It was named that way just to grab some of Java's marketing momentum). Unfortunately, most Web browsers originally implemented JavaScript in a different way from the other Web browsers, and even from other versions of themselves. The standardization of JavaScript in the form of *ECMAScript* has helped, but it has taken a long time for the various browsers to catch up (and it didn't help that Microsoft was pushing its own agenda in the form of VBScript, which also had vague similarities to JavaScript). In general, you must program in a kind of least-common-denominator form of JavaScript in order to be able to run on all browsers. Dealing with errors and debugging JavaScript can only be described as a mess. As proof of its difficulty, only recently has anyone created a truly complex piece of JavaScript (Google, in GMail), and that required excessive dedication and expertise.

This points out that the scripting languages used inside Web browsers are really intended to solve specific types of problems, primarily the creation of richer and more interactive graphical user interfaces (GUIs). However, a scripting language might solve 80 percent of the problems encountered in client-side programming. Your problems might very well fit completely within that 80 percent, and since scripting languages can allow easier and faster development, you should probably consider a scripting language before looking at a more involved solution such as Java programming.

Java

If a scripting language can solve 80 percent of the client-side programming problems, what about the other 20 percent—the "really hard stuff"? Java is a popular solution for this. Not only is it a powerful programming language built to be secure, cross-platform, and international, but Java is being continually extended to provide language features and libraries that elegantly handle problems that are difficult in traditional programming languages, such as concurrency, database access, network programming, and distributed computing. Java allows client-side programming via the *applet* and with *Java Web Start*.

An applet is a mini-program that will run only under a Web browser. The applet is downloaded automatically as part of a Web page (just as, for example, a graphic is automatically downloaded). When the applet is activated, it executes a program. This is part of its beauty—it provides you with a way to automatically distribute the client software from the server at the time the user needs the client software, and no sooner. The user gets the latest version of the client software without fail and without difficult reinstallation. Because of the way Java is designed, the programmer needs to create only a single program, and that program automatically works with all computers that have browsers with built-in Java interpreters. (This safely includes the vast majority of machines.) Since Java is a full-fledged programming language, you can do as much work as possible on the client before and after making requests of the server. For example, you won't need to send a request form across the Internet to discover that you've gotten a date or some other parameter wrong, and your client computer can quickly do the work of plotting data instead of waiting for the server to make a plot and ship a graphic image back to you. Not only do you get the immediate win of speed and responsiveness, but the general network traffic and load on servers can be reduced, preventing the entire Internet from slowing down.

Alternatives

To be honest, Java applets have not particularly lived up to their initial fanfare. When Java first appeared, what everyone seemed most excited about was applets, because these would finally allow serious client-side programmability, to increase responsiveness and decrease bandwidth requirements for Internet-based applications. People envisioned vast possibilities.

Indeed, you can find some very clever applets on the Web. But the overwhelming move to applets never happened. The biggest problem was probably that the 10 MB download necessary to install the Java Runtime Environment (JRE) was too scary for the average user. The fact that Microsoft chose not to include the JRE with Internet Explorer may have sealed its fate. In any event, Java applets didn't happen on a large scale.

Nonetheless, applets and *Java Web Start* applications are still valuable in some situations. Anytime you have control over user machines, for example within a corporation, it is reasonable to distribute and update client applications using these technologies, and this can save considerable time, effort, and money, especially if you need to do frequent updates.

In the *Graphical User Interfaces* chapter, we will look at one promising new technology, Macromedia's *Flex*, which allows you to create Flash-based applet-equivalents. Because the Flash Player is available on upwards of 98 percent of all Web browsers (including Windows, Linux and the Mac) it can be considered an accepted standard. Installing or upgrading the Flash Player is quick and easy. The ActionScript language is based on ECMAScript so it is reasonably familiar, but Flex allows you to program without worrying about browser specifics—thus it is far more attractive than JavaScript. For client-side programming, this is an alternative worth considering.

.NET and C#

For a while, the main competitor to Java applets was Microsoft's ActiveX, although that required that the client be running Windows. Since then, Microsoft has produced a full competitor to Java in the form of the .NET platform and the C# programming language. The .NET platform is roughly the same as the *Java Virtual Machine* (JVM; the software platform on which Java programs execute) and Java libraries, and C# bears unmistakable similarities to Java. This is certainly the best work that Microsoft has done in the arena of programming languages and programming environments. Of course, they had the considerable advantage of being able to see what worked well and what didn't work so well in Java, and build upon that, but build they have. This is the first time since its inception that Java has had any real competition. As a result, the Java designers at Sun have taken a hard look at C# and why programmers might want to move to it, and have responded by making fundamental improvements to Java in Java SE5.

Currently, the main vulnerability and important question concerning .NET is whether Microsoft will allow it to be *completely* ported to other platforms. They claim there's no problem doing this, and the Mono project (*www.go-mono.com*) has a partial implementation of .NET working on Linux, but until the implementation is complete and Microsoft has not decided to squash any part of it, .NET as a cross-platform solution is still a risky bet.

Internet vs. intranet

The Web is the most general solution to the client/server problem, so it makes sense to use the same technology to solve a subset of the problem, in particular the classic client/server problem *within* a company. With traditional client/server approaches you have the problem of multiple types of client computers, as well as the difficulty of installing new client software, both of which are handily solved with Web browsers and client-side programming. When Web technology is used for an information network that is restricted to a particular company, it is referred to as an

intranet. Intranets provide much greater security than the Internet, since you can physically control access to the servers within your company. In terms of training, it seems that once people understand the general concept of a browser it's much easier for them to deal with differences in the way pages and applets look, so the learning curve for new kinds of systems seems to be reduced.

The security problem brings us to one of the divisions that seems to be automatically forming in the world of client-side programming. If your program is running on the Internet, you don't know what platform it will be working under, and you want to be extra careful that you don't disseminate buggy code. You need something cross-platform and secure, like a scripting language or Java.

If you're running on an intranet, you might have a different set of constraints. It's not uncommon that your machines could all be Intel/Windows platforms. On an intranet, you're responsible for the quality of your own code and can repair bugs when they're discovered. In addition, you might already have a body of legacy code that you've been using in a more traditional client/server approach, whereby you must physically install client programs every time you do an upgrade. The time wasted in installing upgrades is the most compelling reason to move to browsers, because upgrades are invisible and automatic (Java Web Start is also a solution to this problem). If you are involved in such an intranet, the most sensible approach to take is the shortest path that allows you to use your existing code base, rather than trying to recode your programs in a new language.

When faced with this bewildering array of solutions to the client-side programming problem, the best plan of attack is a cost-benefit analysis. Consider the constraints of your problem and what would be the shortest path to your solution. Since client-side programming is still programming, it's always a good idea to take the fastest development approach for your particular situation. This is an aggressive stance to prepare for inevitable encounters with the problems of program development.

Server-side programming

This whole discussion has ignored the issue of server-side programming, which is arguably where Java has had its greatest success. What happens when you make a request of a server? Most of the time the request is simply "Send me this file." Your browser then interprets the file in some appropriate fashion: as an HTML page, a graphic image, a Java applet, a script program, etc.

A more complicated request to a server generally involves a database transaction. A common scenario involves a request for a complex database search, which the server then formats into an HTML page and sends to you as the result. (Of course, if the client has more intelligence via Java or a scripting language, the raw data can be sent and formatted at the client end, which will be faster and less load on the server.) Or you might want to register your name in a database when you join a group or place an order, which will involve changes to that database. These database requests must be processed via some code on the server side, which is generally referred to as server-side programming. Traditionally, server-side programming has been performed using Perl, Python, C++, or some other language to create CGI programs, but more sophisticated systems have since appeared. These include Java-based Web servers that allow you to perform all your server-side programming in Java by writing what are called *servlets*. Servlets and their offspring, JSPs, are two of the most compelling reasons that companies that develop Web sites are moving to Java, especially because they eliminate the problems of dealing with differently abled browsers. Server-side programming topics are covered in *Thinking in Enterprise Java* at *www.MindView.net*.

Despite all this talk about Java on the Internet, it is a general-purpose programming language that can solve the kinds of problems that you can solve with other languages. Here, Java's strength is not only in its portability, but also its programmability, its robustness, its large, standard library and the numerous third-party libraries that are available and that continue to be developed.

Summary

You know what a procedural program looks like: data definitions and function calls. To find the meaning of such a program, you must work at it, looking through the function calls and low-level concepts to create a model in your mind. This is the reason we need intermediate representations when designing procedural programs—by themselves, these programs tend to be confusing because the terms of expression are oriented more toward the computer than to the problem you're solving.

Because OOP adds many new concepts on top of what you find in a procedural language, your natural assumption may be that the resulting Java program will be far more complicated than the equivalent procedural program. Here, you'll be pleasantly surprised: A well-written Java program is generally far simpler and much easier to understand than a procedural program. What you'll see are the definitions of the objects that represent concepts in your problem space (rather than the issues of the computer representation) and messages sent to those objects to represent the activities in that space. One of the delights of object-oriented programming is that, with a well-designed program, it's easy to understand the code by reading it. Usually, there's a lot less code as well, because many of your problems will be solved by reusing existing library code.

OOP and Java may not be for everyone. It's important to evaluate your own needs and decide whether Java will optimally satisfy those needs, or if you might be better off with another programming system (including the one you're currently using). If you know that your needs will be very specialized for the foreseeable future and if you have specific constraints that may not be satisfied by Java, then you owe it to yourself to investigate the alternatives (in particular, I recommend looking at Python; see *www.Python.org*). If you still choose Java as your language, you'll at least understand what the options were and have a clear vision of why you took that direction.

Everything Is an Object

"If we spoke a different language, we would perceive a somewhat different world."

Ludwig Wittgenstein (1889-1951)

Although it is based on C++, Java is more of a "pure" object-oriented language.

Both C++ and Java are hybrid languages, but in Java the designers felt that the hybridization was not as important as it was in C++. A hybrid language allows multiple programming styles; the reason C++ is hybrid is to support backward compatibility with the C language. Because C++ is a superset of the C language, it includes many of that language's undesirable features, which can make some aspects of C++ overly complicated.

The Java language assumes that you want to do only object-oriented programming. This means that before you can begin you must shift your mindset into an object-oriented world (unless it's already there). The benefit of this initial effort is the ability to program in a language that is simpler to learn and to use than many other OOP languages. In this chapter you'll see the basic components of a Java program and learn that (almost) everything in Java is an object.

You manipulate objects with references

Each programming language has its own means of manipulating elements in memory. Sometimes the programmer must be constantly aware of what type of manipulation is going on. Are you manipulating the element directly, or are you dealing with some kind of indirect representation (a pointer in C or C++) that must be treated with a special syntax?

All this is simplified in Java. You treat everything as an object, using a single consistent syntax. Although you *treat* everything as an object, the identifier you manipulate is actually a "reference" to an object.[1] You might imagine a television (the object) and a remote control (the reference). As

---

[1] This can be a flashpoint. There are those who say, "Clearly, it's a pointer," but this presumes an underlying implementation. Also, Java references are much more akin to C++ references than to pointers in their syntax. In the 1st edition of this book, I chose to invent a new term, "handle," because C++ references and Java references have some important differences. I was coming out of C++ and did not want to confuse the C++ programmers whom I assumed would be the largest audience for Java. In the 2nd edition, I decided that "reference" was the more commonly used term, and that anyone changing from C++ would have a lot more to cope with than the terminology of references, so they might as well jump in with both feet. However, there are people who disagree even with the term "reference." I read in one book where it was "completely wrong to say that Java supports pass by reference," because Java object identifiers (according to that author) are *actually* "object references." And (he goes on) everything is *actually* pass by value. So you're not passing by reference, you're "passing an object reference by value." One could argue for the precision of such convoluted explanations, but I think my approach simplifies the understanding of the concept without hurting anything (well, the language lawyers may claim that I'm lying to you, but I'll say that I'm providing an appropriate abstraction).

long as you're holding this reference, you have a connection to the television, but when someone says, "Change the channel" or "Lower the volume," what you're manipulating is the reference, which in turn modifies the object. If you want to move around the room and still control the television, you take the remote/reference with you, not the television.

Also, the remote control can stand on its own, with no television. That is, just because you have a reference doesn't mean there's necessarily an object connected to it. So if you want to hold a word or sentence, you create a **String** reference:

```
String s;
```

But here you've created *only* the reference, not an object. If you decided to send a message to **s** at this point, you'll get an error because **s** isn't actually attached to anything (there's no television). A safer practice, then, is always to initialize a reference when you create it:

```
String s = "asdf";
```

However, this uses a special Java feature: Strings can be initialized with quoted text. Normally, you must use a more general type of initialization for objects.

You must create all the objects

When you create a reference, you want to connect it with a new object. You do so, in general, with the **new** operator. The keyword **new** says, "Make me a new one of these objects." So in the preceding example, you can say:

```
String s = new String("asdf");
```

Not only does this mean "Make me a new **String**," but it also gives information about *how* to make the **String** by supplying an initial character string.

Of course, Java comes with a plethora of ready-made types in addition to **String**. What's more important is that you can create your own types. In fact, creating new types is the fundamental activity in Java programming, and it's what you'll be learning about in the rest of this book.

Where storage lives

It's useful to visualize some aspects of how things are laid out while the program is running—in particular how memory is arranged. There are five different places to store data:

1. **Registers**. This is the fastest storage because it exists in a place different from that of other storage: inside the processor. However, the number of registers is severely limited, so registers are allocated as they are needed. You don't have direct control, nor do you see any evidence in your programs that registers even exist (C & C++, on the other hand, allow you to suggest register allocation to the compiler).

2. **The stack**. This lives in the general random-access memory (RAM) area, but has direct support from the processor via its *stack pointer*. The stack pointer is moved down to create new memory and moved up to release that memory. This is an extremely fast and efficient way to allocate storage, second only to registers. The Java system must know, while it is creating the program, the exact lifetime of all the items that are stored on the stack. This constraint places limits on the flexibility of your programs, so while some Java storage exists on the stack—in particular, object references—Java objects themselves are not placed on the stack.

3. **The heap**. This is a general-purpose pool of memory (also in the RAM area) where all Java objects live. The nice thing about the heap is that, unlike the stack, the compiler doesn't need to know how long that storage must stay on the heap. Thus, there's a great deal of flexibility in using storage on the heap. Whenever you need an object, you simply write the code to create it by using **new**, and the storage is allocated on the heap when that code is executed. Of course there's a price you pay for this flexibility: It may take more time to allocate and clean up heap storage than stack storage (if you even *could* create objects on the stack in Java, as you can in C++).

4. **Constant storage**. Constant values are often placed directly in the program code, which is safe since they can never change. Sometimes constants are cordoned off by themselves so that they can be optionally placed in read-only memory (ROM), in embedded systems.[2]

5. **Non-RAM storage**. If data lives completely outside a program, it can exist while the program is not running, outside the control of the program. The two primary examples of this are *streamed objects,* in which objects are turned into streams of bytes, generally to be sent to another machine, and *persistent objects,* in which the objects are placed on disk so they will hold their state even when the program is terminated. The trick with these types of storage is turning the objects into something that can exist on the other medium, and yet can be resurrected into a regular RAM-based object when necessary. Java provides support for *lightweight persistence*, and mechanisms such as JDBC and Hibernate provide more sophisticated support for storing and retrieving object information in databases.

Special case: primitive types

One group of types, which you'll use quite often in your programming, gets special treatment. You can think of these as "primitive" types. The reason for the special treatment is that to create an object with **new**—especially a small, simple variable—isn't very efficient, because **new** places objects on the heap. For these types Java falls back on the approach taken by C and C++. That is, instead of creating the variable by using **new**, an "automatic" variable is created that is *not a reference*. The variable holds the value directly, and it's placed on the stack, so it's much more efficient.

Java determines the size of each primitive type. These sizes don't change from one machine architecture to another as they do in most languages. This size invariance is one reason Java programs are more portable than programs in most other languages.

Primitive type	Size	Minimum	Maximum	Wrapper type
boolean	—	—	—	**Boolean**
char	16 bits	Unicode 0	Unicode 2^{16}- 1	**Character**
byte	8 bits	-128	+127	**Byte**
short	16 bits	-2^{15}	$+2^{15}$-1	**Short**
int	32 bits	-2^{31}	$+2^{31}$-1	**Integer**
long	64 bits	-2^{63}	$+2^{63}$-1	**Long**

[2] An example of this is the string pool. All literal strings and string-valued constant expressions are interned automatically and put into special static storage.

Primitive type	Size	Minimum	Maximum	Wrapper type
float	32 bits	IEEE754	IEEE754	**Float**
double	64 bits	IEEE754	IEEE754	**Double**
void	—	—	—	**Void**

All numeric types are signed, so don't look for unsigned types.

The size of the **boolean** type is not explicitly specified; it is only defined to be able to take the literal values **true** or **false**.

The "wrapper" classes for the primitive data types allow you to make a non-primitive object on the heap to represent that primitive type. For example:

```
char c = 'x';
Character ch = new Character(c);
```

Or you could also use:

```
Character ch = new Character('x');
```

Java SE5 *autoboxing* will automatically convert from a primitive to a wrapper type:

```
Character ch = 'x';
```

and back:

```
char c = ch;
```

The reasons for wrapping primitives will be shown in a later chapter.

High-precision numbers

Java includes two classes for performing high-precision arithmetic: **BigInteger** and **BigDecimal**. Although these approximately fit into the same category as the "wrapper" classes, neither one has a primitive analogue.

Both classes have methods that provide analogues for the operations that you perform on primitive types. That is, you can do anything with a **BigInteger** or **BigDecimal** that you can with an **int** or **float**, it's just that you must use method calls instead of operators. Also, since there's more involved, the operations will be slower. You're exchanging speed for accuracy.

BigInteger supports arbitrary-precision integers. This means that you can accurately represent integral values of any size without losing any information during operations.

BigDecimal is for arbitrary-precision fixed-point numbers; you can use these for accurate monetary calculations, for example.

Consult the JDK documentation for details about the constructors and methods you can call for these two classes.

Arrays in Java

Virtually all programming languages support some kind of arrays. Using arrays in C and C++ is perilous because those arrays are only blocks of memory. If a program accesses the array outside of its memory block or uses the memory before initialization (common programming errors), there will be unpredictable results.

One of the primary goals of Java is safety, so many of the problems that plague programmers in C and C++ are not repeated in Java. A Java array is guaranteed to be initialized and cannot be accessed outside of its range. The range checking comes at the price of having a small amount of memory overhead on each array as well as verifying the index at run time, but the assumption is that the safety and increased productivity are worth the expense (and Java can sometimes optimize these operations).

When you create an array of objects, you are really creating an array of references, and each of those references is automatically initialized to a special value with its own keyword: **null**. When Java sees **null**, it recognizes that the reference in question isn't pointing to an object. You must assign an object to each reference before you use it, and if you try to use a reference that's still **null**, the problem will be reported at run time. Thus, typical array errors are prevented in Java.

You can also create an array of primitives. Again, the compiler guarantees initialization because it zeroes the memory for that array.

Arrays will be covered in detail in later chapters.

You never need to destroy an object

In most programming languages, the concept of the lifetime of a variable occupies a significant portion of the programming effort. How long does the variable last? If you are supposed to destroy it, when should you? Confusion over variable lifetimes can lead to a lot of bugs, and this section shows how Java greatly simplifies the issue by doing all the cleanup work for you.

Scoping

Most procedural languages have the concept of *scope*. This determines both the visibility and lifetime of the names defined within that scope. In C, C++, and Java, scope is determined by the placement of curly braces **{}**. So for example:

```
{
  int x = 12;
  // Only x available
  {
    int q = 96;
    // Both x & q available
  }
  // Only x available
  // q is "out of scope"
}
```

A variable defined within a scope is available only to the end of that scope.

Any text after a '//' to the end of a line is a comment.

Indentation makes Java code easier to read. Since Java is a free-form language, the extra spaces, tabs, and carriage returns do not affect the resulting program.

You *cannot* do the following, even though it is legal in C and C++:

```
{
  int x = 12;
  {
```

```
    int x = 96; // Illegal
  }
}
```

The compiler will announce that the variable **x** has already been defined. Thus the C and C++ ability to "hide" a variable in a larger scope is not allowed, because the Java designers thought that it led to confusing programs.

Scope of objects

Java objects do not have the same lifetimes as primitives. When you create a Java object using **new**, it hangs around past the end of the scope. Thus if you use:

```
{
  String s = new String("a string");
} // End of scope
```

the reference **s** vanishes at the end of the scope. However, the **String** object that **s** was pointing to is still occupying memory. In this bit of code, there is no way to access the object after the end of the scope, because the only reference to it is out of scope. In later chapters you'll see how the reference to the object can be passed around and duplicated during the course of a program.

It turns out that because objects created with **new** stay around for as long as you want them, a whole slew of C++ programming problems simply vanish in Java. In C++ you must not only make sure that the objects stay around for as long as you need them, you must also destroy the objects when you're done with them.

That brings up an interesting question. If Java leaves the objects lying around, what keeps them from filling up memory and halting your program? This is exactly the kind of problem that would occur in C++. This is where a bit of magic happens. Java has a *garbage collector*, which looks at all the objects that were created with **new** and figures out which ones are not being referenced anymore. Then it releases the memory for those objects, so the memory can be used for new objects. This means that you never need to worry about reclaiming memory yourself. You simply create objects, and when you no longer need them, they will go away by themselves. This eliminates a certain class of programming problem: the so-called "memory leak," in which a programmer forgets to release memory.

Creating new data types: **class**

If everything is an object, what determines how a particular class of object looks and behaves? Put another way, what establishes the *type* of an object? You might expect there to be a keyword called "type," and that certainly would have made sense. Historically, however, most object-oriented languages have used the keyword **class** to mean "I'm about to tell you what a new type of object looks like." The **class** keyword (which is so common that it will not usually be bold-faced throughout this book) is followed by the name of the new type. For example:

```
class ATypeName { /* Class body goes here */ }
```

This introduces a new type, although the class body consists only of a comment (the stars and slashes and what is inside, which will be discussed later in this chapter), so there is not too much that you can do with it. However, you can create an object of this type using **new**:

```
ATypeName a = new ATypeName();
```

But you cannot tell it to do much of anything (that is, you cannot send it any interesting messages) until you define some methods for it.

Fields and methods

When you define a class (and all you do in Java is define classes, make objects of those classes, and send messages to those objects), you can put two types of elements in your class: *fields* (sometimes called *data members*), and *methods* (sometimes called *member functions*). A field is an object of any type that you can talk to via its reference, or a primitive type. If it is a reference to an object, you must initialize that reference to connect it to an actual object (using **new**, as seen earlier).

Each object keeps its own storage for its fields; ordinary fields are not shared among objects. Here is an example of a class with some fields:

```
class DataOnly {
  int i;
  double d;
  boolean b;
}
```

This class doesn't *do* anything except hold data. But you can create an object like this:

```
DataOnly data = new DataOnly();
```

You can assign values to the fields, but you must first know how to refer to a member of an object. This is accomplished by stating the name of the object reference, followed by a period (dot), followed by the name of the member inside the object:

```
objectReference.member
```

For example:

```
data.i = 47;
data.d = 1.1;
data.b = false;
```

It is also possible that your object might contain other objects that contain data you'd like to modify. For this, you just keep "connecting the dots." For example:

```
myPlane.leftTank.capacity = 100;
```

The **DataOnly** class cannot do much of anything except hold data, because it has no methods. To understand how those work, you must first understand *arguments* and *return values*, which will be described shortly.

Default values for primitive members

When a primitive data type is a member of a class, it is guaranteed to get a default value if you do not initialize it:

Primitive type	Default
boolean	false
char	'\u0000' (null)
byte	(byte)0
short	(short)0
int	0
long	0L

Primitive type	Default
float	0.0f
double	0.0d

The default values are only what Java guarantees when the variable is used *as a member of a class*. This ensures that member variables of primitive types will always be initialized (something C++ doesn't do), reducing a source of bugs. However, this initial value may not be correct or even legal for the program you are writing. It's best to always explicitly initialize your variables.

This guarantee doesn't apply to *local variables*—those that are not fields of a class. Thus, if within a method definition you have:

```
int x;
```

Then **x** will get some arbitrary value (as in C and C++); it will not automatically be initialized to zero. You are responsible for assigning an appropriate value before you use **x**. If you forget, Java definitely improves on C++: You get a compile-time error telling you the variable might not have been initialized. (Many C++ compilers will warn you about uninitialized variables, but in Java these are errors.)

Methods, arguments, and return values

In many languages (like C and C++), the term *function* is used to describe a named subroutine. The term that is more commonly used in Java is *method*, as in "a way to do something." If you want, you can continue thinking in terms of functions. It's really only a syntactic difference, but this book follows the common Java usage of the term "method."

Methods in Java determine the messages an object can receive. The fundamental parts of a method are the name, the arguments, the return type, and the body. Here is the basic form:

```
ReturnType methodName( /* Argument list */ ) {
  /* Method body */
}
```

The return type describes the value that comes back from the method after you call it. The argument list gives the types and names for the information that you want to pass into the method. The method name and argument list (which is called the *signature* of the method) uniquely identify that method.

Methods in Java can be created only as part of a class. A method can be called only for an object,[3] and that object must be able to perform that method call. If you try to call the wrong method for an object, you'll get an error message at compile time. You call a method for an object by naming the object followed by a period (dot), followed by the name of the method and its argument list, like this:

```
objectName.methodName(arg1, arg2, arg3);
```

For example, suppose you have a method **f( )** that takes no arguments and returns a value of type **int**. Then, if you have an object called **a** for which **f( )** can be called, you can say this:

[3] **static** methods, which you'll learn about soon, can be called *for the class*, without an object.

```
int x = a.f();
```

The type of the return value must be compatible with the type of **x**.

This act of calling a method is commonly referred to as *sending a message to an object*. In the preceding example, the message is **f( )** and the object is **a**. Object-oriented programming is often summarized as simply "sending messages to objects."

The argument list

The method argument list specifies what information you pass into the method. As you might guess, this information—like everything else in Java—takes the form of objects. So, what you must specify in the argument list are the types of the objects to pass in and the name to use for each one. As in any situation in Java where you seem to be handing objects around, you are actually passing references.[4] The type of the reference must be correct, however. If the argument is supposed to be a **String**, you must pass in a **String** or the compiler will give an error.

Consider a method that takes a **String** as its argument. Here is the definition, which must be placed within a class definition for it to be compiled:

```
int storage(String s) {
  return s.length() * 2;
}
```

This method tells you how many bytes are required to hold the information in a particular **String**. (The size of each **char** in a **String** is 16 bits, or two bytes, to support Unicode characters.) The argument is of type **String** and is called **s**. Once **s** is passed into the method, you can treat it just like any other object. (You can send messages to it.) Here, the **length( )** method is called, which is one of the methods for **String**s; it returns the number of characters in a string.

You can also see the use of the **return** keyword, which does two things. First, it means "Leave the method, I'm done." Second, if the method produces a value, that value is placed right after the **return** statement. In this case, the return value is produced by evaluating the expression **s.length( ) * 2**.

You can return any type you want, but if you don't want to return anything at all, you do so by indicating that the method returns **void**. Here are some examples:

```
boolean flag() { return true; }
double naturalLogBase() { return 2.718; }
void nothing() { return; }
void nothing2() {}
```

When the return type is **void**, then the **return** keyword is used only to exit the method, and is therefore unnecessary when you reach the end of the method. You can return from a method at any point, but if you've given a non-**void** return type, then the compiler will force you (with error messages) to return the appropriate type of value regardless of where you return.

At this point, it can look like a program is just a bunch of objects with methods that take other objects as arguments and send messages to those other objects. That is indeed much of what goes

[4] With the usual exception of the aforementioned "special" data types **boolean**, **char**, **byte**, **short**, **int**, **long**, **float**, and **double**. In general, though, you pass objects, which really means you pass references to objects.

on, but in the following chapter you'll learn how to do the detailed low-level work by making decisions within a method. For this chapter, sending messages will suffice.

Building a Java program

There are several other issues you must understand before seeing your first Java program.

Name visibility

A problem in any programming language is the control of names. If you use a name in one module of the program, and another programmer uses the same name in another module, how do you distinguish one name from another and prevent the two names from "clashing?" In C this is a particular problem because a program is often an unmanageable sea of names. C++ classes (on which Java classes are based) nest functions within classes so they cannot clash with function names nested within other classes. However, C++ still allows global data and global functions, so clashing is still possible. To solve this problem, C++ introduced *namespaces* using additional keywords.

Java was able to avoid all of this by taking a fresh approach. To produce an unambiguous name for a library, the Java creators want you to use your Internet domain name in reverse since domain names are guaranteed to be unique. Since my domain name is **MindView.net**, my utility library of foibles would be named **net.mindview.utility.foibles**. After your reversed domain name, the dots are intended to represent subdirectories.

In Java 1.0 and Java 1.1 the domain extensions **com**, **edu**, **org**, **net**, etc., were capitalized by convention, so the library would appear: **NET.mindview.utility.foibles**. Partway through the development of Java 2, however, it was discovered that this caused problems, so now the entire package name is lowercase.

This mechanism means that all of your files automatically live in their own namespaces, and each class within a file must have a unique identifier—the language prevents name clashes for you.

Using other components

Whenever you want to use a predefined class in your program, the compiler must know how to locate it. Of course, the class might already exist in the same source-code file that it's being called from. In that case, you simply use the class—even if the class doesn't get defined until later in the file (Java eliminates the so-called "forward referencing" problem).

What about a class that exists in some other file? You might think that the compiler should be smart enough to simply go and find it, but there is a problem. Imagine that you want to use a class with a particular name, but more than one definition for that class exists (presumably these are different definitions). Or worse, imagine that you're writing a program, and as you're building it you add a new class to your library that conflicts with the name of an existing class.

To solve this problem, you must eliminate all potential ambiguities. This is accomplished by telling the Java compiler exactly what classes you want by using the **import** keyword. **import** tells the compiler to bring in a package, which is a library of classes. (In other languages, a library could consist of functions and data as well as classes, but remember that all code in Java must be written inside a class.)

Most of the time you'll be using components from the standard Java libraries that come with your compiler. With these, you don't need to worry about long, reversed domain names; you just say, for example:

```
import java.util.ArrayList;
```

to tell the compiler that you want to use Java's **ArrayList** class. However, **util** contains a number of classes, and you might want to use several of them without declaring them all explicitly. This is easily accomplished by using '*****' to indicate a wild card:

```
import java.util.*;
```

It is more common to import a collection of classes in this manner than to import classes individually.

The **static** keyword

Ordinarily, when you create a class you are describing how objects of that class look and how they will behave. You don't actually get an object until you create one using **new**, and at that point storage is allocated and methods become available.

There are two situations in which this approach is not sufficient. One is if you want to have only a single piece of storage for a particular field, regardless of how many objects of that class are created, or even if no objects are created. The other is if you need a method that isn't associated with any particular object of this class. That is, you need a method that you can call even if no objects are created.

You can achieve both of these effects with the **static** keyword. When you say something is **static**, it means that particular field or method is not tied to any particular object instance of that class. So even if you've never created an object of that class you can call a **static** method or access a **static** field. With ordinary, non-**static** fields and methods, you must create an object and use that object to access the field or method, since non-**static** fields and methods must know the particular object they are working with.[5]

Some object-oriented languages use the terms *class data* and *class methods*, meaning that the data and methods exist only for the class as a whole, and not for any particular objects of the class. Sometimes the Java literature uses these terms too.

To make a field or method **static**, you simply place the keyword before the definition. For example, the following produces a **static** field and initializes it:

```
class StaticTest {
  static int i = 47;
}
```

Now even if you make two **StaticTest** objects, there will still be only one piece of storage for **StaticTest.i.** Both objects will share the same **i**. Consider:

```
StaticTest st1 = new StaticTest();
StaticTest st2 = new StaticTest();
```

At this point, both **st1.i** and **st2.i** have the same value of 47 since they refer to the same piece of memory.

---

[5] Of course, since **static** methods don't need any objects to be created before they are used, they cannot *directly* access non-**static** members or methods by simply calling those other members without referring to a named object (since non-**static** members and methods must be tied to a particular object).

There are two ways to refer to a **static** variable. As the preceding example indicates, you can name it via an object, by saying, for example, **st2.i**. You can also refer to it directly through its class name, something you cannot do with a non-**static** member.

```
StaticTest.i++;
```

The **++** operator adds one to the variable. At this point, both **st1.i** and **st2.i** will have the value 48.

Using the class name is the preferred way to refer to a **static** variable. Not only does it emphasize that variable's **static** nature, but in some cases it gives the compiler better opportunities for optimization.

Similar logic applies to **static** methods. You can refer to a **static** method either through an object as you can with any method, or with the special additional syntax **ClassName.method( )**. You define a **static** method in a similar way:

```
class Incrementable {
  static void increment() { StaticTest.i++; }
}
```

You can see that the **Incrementable** method **increment( )** increments the **static** data **i** using the **++** operator. You can call **increment( )** in the typical way, through an object:

```
Incrementable sf = new Incrementable();
sf.increment();
```

Or, because **increment( )** is a **static** method, you can call it directly through its class:

```
Incrementable.increment();
```

Although **static**, when applied to a field, definitely changes the way the data is created (one for each class versus the non-**static** one for each object), when applied to a method it's not so dramatic. An important use of **static** for methods is to allow you to call that method without creating an object. This is essential, as you will see, in defining the **main( )** method that is the entry point for running an application.

Your first Java program

Finally, here's the first complete program. It starts by printing a string, and then the date, using the **Date** class from the Java standard library.

```
// HelloDate.java
import java.util.*;

public class HelloDate {
  public static void main(String[] args) {
    System.out.println("Hello, it's: ");
    System.out.println(new Date());
  }
}
```

At the beginning of each program file, you must place any necessary **import** statements to bring in extra classes you'll need for the code in that file. Note that I say "extra." That's because there's a certain library of classes that are automatically brought into every Java file: **java.lang**. Start up your Web browser and look at the documentation from Sun. (If you haven't downloaded the JDK

documentation from *http://java.sun.com*, do so now.[6] Note that this documentation doesn't come packed with the JDK; you must do a separate download to get it.) If you look at the list of the packages, you'll see all the different class libraries that come with Java. Select **java.lang**. This will bring up a list of all the classes that are part of that library. Since **java.lang** is implicitly included in every Java code file, these classes are automatically available. There's no **Date** class listed in **java.lang**, which means you must import another library to use that. If you don't know the library where a particular class is, or if you want to see all of the classes, you can select "Tree" in the Java documentation. Now you can find every single class that comes with Java. Then you can use the browser's "find" function to find **Date**. When you do you'll see it listed as **java.util.Date**, which lets you know that it's in the **util** library and that you must **import java.util.*** in order to use **Date**.

If you go back to the beginning, select **java.lang** and then **System**, you'll see that the **System** class has several fields, and if you select **out**, you'll discover that it's a **static PrintStream** object. Since it's **static**, you don't need to create anything with **new**. The **out** object is always there, and you can just use it. What you can do with this **out** object is determined by its type: **PrintStream**. Conveniently, **PrintStream** is shown in the description as a hyperlink, so if you click on that, you'll see a list of all the methods you can call for **PrintStream**. There are quite a few, and these will be covered later in the book. For now all we're interested in is **println( )**, which in effect means "Print what I'm giving you out to the console and end with a newline." Thus, in any Java program you write you can write something like this:

```
System.out.println("A String of things");
```

whenever you want to display information to the console.

The name of the class is the same as the name of the file. When you're creating a standalone program such as this one, one of the classes in the file must have the same name as the file. (The compiler complains if you don't do this.) That class must contain a method called **main( )** with this signature and return type:

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

The **public** keyword means that the method is available to the outside world (described in detail in the *Access Control* chapter). The argument to **main( )** is an array of **String** objects. The **args** won't be used in this program, but the Java compiler insists that they be there because they hold the arguments from the command line.

The line that prints the date is quite interesting:

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

The argument is a **Date** object that is being created just to send its value (which is automatically converted to a **String**) to **println( )**. As soon as this statement is finished, that **Date** is unnecessary, and the garbage collector can come along and get it anytime. We don't need to worry about cleaning it up.

When you look at the JDK documentation from *http://java.sun.com*, you will see that **System** has many other methods that allow you to produce interesting effects (one of Java's most powerful assets is its large set of standard libraries). For example:

---

[6] The Java compiler and documentation from Sun tend to change regularly, and the best place to get them is directly from Sun. By downloading it yourself, you will get the most recent version.

```
//: object/ShowProperties.java

public class ShowProperties {
  public static void main(String[] args) {
    System.getProperties().list(System.out);
    System.out.println(System.getProperty("user.name"));
    System.out.println(
      System.getProperty("java.library.path"));
  }
} ///:~
```

The first line in **main( )** displays all of the "properties" from the system where you are running the program, so it gives you environment information. The **list( )** method sends the results to its argument, **System.out**. You will see later in the book that you can send the results elsewhere, to a file, for example. You can also ask for a specific property—in this case, the user name and **java.library.path**. (The unusual comments at the beginning and end will be explained a little later.)

Compiling and running

To compile and run this program, and all the other programs in this book, you must first have a Java programming environment. There are a number of third-party development environments, but in this book I will assume that you are using the Java Developer's Kit (JDK) from Sun, which is free. If you are using another development system,[7] you will need to look in the documentation for that system to determine how to compile and run programs.

Get on the Internet and go to *http://java.sun.com*. There you will find information and links that will lead you through the process of downloading and installing the JDK for your particular platform.

Once the JDK is installed, and you've set up your computer's path information so that it will find **javac** and **java**, download and unpack the source code for this book (you can find it at *www.MindView.net*). This will create a subdirectory for each chapter in this book. Move to the subdirectory named **objects** and type:

```
javac HelloDate.java
```

This command should produce no response. If you get any kind of an error message, it means you haven't installed the JDK properly and you need to investigate those problems.

On the other hand, if you just get your command prompt back, you can type:

```
java HelloDate
```

and you'll get the message and the date as output.

This is the process you can use to compile and run each of the programs in this book. However, you will see that the source code for this book also has a file called **build.xml** in each chapter, and this contains "Ant" commands for automatically building the files for that chapter. Buildfiles and Ant (including where to download it) are described more fully in the supplement you will find at *http://MindView.net/Books/BetterJava*, but once you have Ant installed (from

[7] IBM's "jikes" compiler is a common alternative, as it is significantly faster than Sun's javac (although if you're building groups of files using *Ant*, there's not too much of a difference). There are also open-source projects to create Java compilers, runtime environments, and libraries.

http://jakarta.apache.org/ant) you can just type '**ant**' at the command prompt to compile and run the programs in each chapter. If you haven't installed Ant yet, you can just type the **javac** and **java** commands by hand.

Comments and embedded documentation

There are two types of comments in Java. The first is the traditional C-style comment that was inherited by C++. These comments begin with a **/*** and continue, possibly across many lines, until a ***/**. Note that many programmers will begin each line of a continued comment with a *******, so you'll often see:

```
/* This is a comment
 * that continues
 * across lines
 */
```

Remember, however, that everything inside the **/*** and ***/** is ignored, so there's no difference in saying:

```
/* This is a comment that
continues across lines */
```

The second form of comment comes from C++. It is the single-line comment, which starts with a **//** and continues until the end of the line. This type of comment is convenient and commonly used because it's easy. You don't need to hunt on the keyboard to find **/** and then ***** (instead, you just press the same key twice), and you don't need to close the comment. So you will often see:

```
// This is a one-line comment
```

Comment documentation

Possibly the biggest problem with documenting code has been maintaining that documentation. If the documentation and the code are separate, it becomes tedious to change the documentation every time you change the code. The solution seems simple: Link the code to the documentation. The easiest way to do this is to put everything in the same file. To complete the picture, however, you need a special comment syntax to mark the documentation and a tool to extract those comments and put them in a useful form. This is what Java has done.

The tool to extract the comments is called *Javadoc*, and it is part of the JDK installation. It uses some of the technology from the Java compiler to look for special comment tags that you put in your programs. It not only extracts the information marked by these tags, but it also pulls out the class name or method name that adjoins the comment. This way you can get away with the minimal amount of work to generate decent program documentation.

The output of Javadoc is an HTML file that you can view with your Web browser. Thus, Javadoc allows you to create and maintain a single source file and automatically generate useful documentation. Because of Javadoc, you have a straightforward standard for creating documentation, so you can expect or even demand documentation with all Java libraries.

In addition, you can write your own Javadoc handlers, called *doclets*, if you want to perform special operations on the information processed by Javadoc (to produce output in a different format, for example). Doclets are introduced in the supplement at *http://MindView.net/Books/BetterJava*.

What follows is only an introduction and overview of the basics of Javadoc. A thorough description can be found in the JDK documentation. When you unpack the documentation, look in the "tooldocs" subdirectory (or follow the "tooldocs" link).

Syntax

All of the Javadoc commands occur only within **/**** comments. The comments end with ***/** as usual. There are two primary ways to use Javadoc: Embed HTML or use "doc tags." *Standalone doc tags* are commands that start with an '**@**' and are placed at the beginning of a comment line. (A leading '*****', however, is ignored.) *Inline doc tags* can appear anywhere within a Javadoc comment and also start with an '**@**' but are surrounded by curly braces.

There are three "types" of comment documentation, which correspond to the element the comment precedes: class, field, or method. That is, a class comment appears right before the definition of a class, a field comment appears right in front of the definition of a field, and a method comment appears right in front of the definition of a method. As a simple example:

```
//: object/Documentation1.java
/** A class comment */
public class Documentation1 {
  /** A field comment */
  public int i;
  /** A method comment */
  public void f() {}
} ///:~
```

Note that Javadoc will process comment documentation for only **public** and **protected** members. Comments for **private** and package-access members (see the *Access Control* chapter) are ignored, and you'll see no output. (However, you can use the **-private** flag to include **private** members as well.) This makes sense, since only **public** and **protected** members are available outside the file, which is the client programmer's perspective.

The output for the preceding code is an HTML file that has the same standard format as all the rest of the Java documentation, so users will be comfortable with the format and can easily navigate your classes. It's worth entering the preceding code, sending it through Javadoc, and viewing the resulting HTML file to see the results.

Embedded HTML

Javadoc passes HTML commands through to the generated HTML document. This allows you full use of HTML; however, the primary motive is to let you format code, such as:

```
//: object/Documentation2.java
/**
* <pre>
* System.out.println(new Date());
* </pre>
*/
///:~
```

You can also use HTML just as you would in any other Web document to format the regular text in your descriptions:

```
//: object/Documentation3.java
/**
* You can <em>even</em> insert a list:
```

```
* <ol>
* <li> Item one
* <li> Item two
* <li> Item three
* </ol>
*/
///:~
```

Note that within the documentation comment, asterisks at the beginning of a line are thrown away by Javadoc, along with leading spaces. Javadoc reformats everything so that it conforms to the standard documentation appearance. Don't use headings such as **<h1>** or **<hr>** as embedded HTML, because Javadoc inserts its own headings and yours will interfere with them.

All types of comment documentation—class, field, and method—can support embedded HTML.

Some example tags

Here are some of the Javadoc tags available for code documentation. Before trying to do anything serious using Javadoc, you should consult the Javadoc reference in the JDK documentation to learn all the different ways that you can use Javadoc.

@see

This tag allows you to refer to the documentation in other classes. Javadoc will generate HTML with the **@see** tags hyperlinked to the other documentation. The forms are:

```
@see classname
@see fully-qualified-classname
@see fully-qualified-classname#method-name
```

Each one adds a hyperlinked "See Also" entry to the generated documentation. Javadoc will not check the hyperlinks you give it to make sure they are valid.

{@link *package.class#member label*}

Very similar to **@see**, except that it can be used inline and uses the *label* as the hyperlink text rather than "See Also."

{@docRoot}

Produces the relative path to the documentation root directory. Useful for explicit hyperlinking to pages in the documentation tree.

{@inheritDoc}

Inherits the documentation from the nearest base class of this class into the current doc comment.

@version

This is of the form:

```
@version version-information
```

in which **version-information** is any significant information you see fit to include. When the **-version** flag is placed on the Javadoc command line, the version information will be called out specially in the generated HTML documentation.

@author

This is of the form:

```
@author author-information
```

in which **author-information** is, presumably, your name, but it could also include your email address or any other appropriate information. When the **-author** flag is placed on the Javadoc command line, the author information will be called out specially in the generated HTML documentation.

You can have multiple author tags for a list of authors, but they must be placed consecutively. All the author information will be lumped together into a single paragraph in the generated HTML.

@since

This tag allows you to indicate the version of this code that began using a particular feature. You'll see it appearing in the HTML Java documentation to indicate what version of the JDK is used.

@param

This is used for method documentation, and is of the form:

```
@param parameter-name description
```

in which **parameter-name** is the identifier in the method parameter list, and **description** is text that can continue on subsequent lines. The description is considered finished when a new documentation tag is encountered. You can have any number of these, presumably one for each parameter.

@return

This is used for method documentation, and looks like this:

```
@return description
```

in which **description** gives you the meaning of the return value. It can continue on subsequent lines.

@throws

Exceptions will be demonstrated in the *Error Handling with Exceptions* chapter. Briefly, they are objects that can be "thrown" out of a method if that method fails. Although only one exception object can emerge when you call a method, a particular method might produce any number of different types of exceptions, all of which need descriptions. So the form for the exception tag is:

```
@throws fully-qualified-class-name description
```

in which *fully-qualified-class-name* gives an unambiguous name of an exception class that's defined somewhere, and *description* (which can continue on subsequent lines) tells you why this particular type of exception can emerge from the method call.

@deprecated

This is used to indicate features that were superseded by an improved feature. The deprecated tag is a suggestion that you no longer use this particular feature, since sometime in the future it is likely to be removed. A method that is marked **@deprecated** causes the compiler to issue a

warning if it is used. In Java SE5, the **@deprecated** Javadoc tag has been superseded by the **@Deprecated** *annotation* (you'll learn about these in the *Annotations* chapter).

Documentation example

Here is the first Java program again, this time with documentation comments added:

```
//: object/HelloDate.java
import java.util.*;

/** The first Thinking in Java example program.
 * Displays a string and today's date.
 * @author Bruce Eckel
 * @author www.MindView.net
 * @version 4.0
*/
public class HelloDate {
  /** Entry point to class & application.
   * @param args array of string arguments
   * @throws exceptions No exceptions thrown
  */
  public static void main(String[] args) {
    System.out.println("Hello, it's: ");
    System.out.println(new Date());
  }
} /* Output: (55% match)
Hello, it's:
Wed Oct 05 14:39:36 MDT 2005
*///:~
```

The first line of the file uses my own technique of putting a '**//:**' as a special marker for the comment line containing the source file name. That line contains the path information to the file (**object** indicates this chapter) followed by the file name. The last line also finishes with a comment, and this one ('**///:~**') indicates the end of the source code listing, which allows it to be automatically updated into the text of this book after being checked with a compiler and executed.

The **/* Output:** tag indicates the beginning of the output that will be generated by this file. In this form, it can be automatically tested to verify its accuracy. In this case, the **(55% match)** indicates to the testing system that the output will be fairly different from one run to the next so it should only expect a 55 percent correlation with the output shown here. Most examples in this book that produce output will contain the output in this commented form, so you can see the output and know that it is correct.

Coding style

The style described in the *Code Conventions for the Java Programming Language*[8] is to capitalize the first letter of a class name. If the class name consists of several words, they are run together (that is, you don't use underscores to separate the names), and the first letter of each embedded word is capitalized, such as:

---

[8] *http://java.sun.com/docs/codeconv/index.html*. To preserve space in this book and seminar presentations, not all of these guidelines could be followed, but you'll see that the style I use here matches the Java standard as much as possible.

```
class AllTheColorsOfTheRainbow { // ...
```

This is sometimes called "camel-casing." For almost everything else—methods, fields (member variables), and object reference names—the accepted style is just as it is for classes *except* that the first letter of the identifier is lowercase. For example:

```
class AllTheColorsOfTheRainbow {
  int anIntegerRepresentingColors;
  void changeTheHueOfTheColor(int newHue) {
    // ...
  }
  // ...
}
```

The user must also type all these long names, so be merciful.

The Java code you will see in the Sun libraries also follows the placement of open-and-close curly braces that you see used in this book.

Summary

The goal of this chapter is just enough Java to understand how to write a simple program. You've also gotten an overview of the language and some of its basic ideas. However, the examples so far have all been of the form "Do this, then do that, then do something else." The next two chapters will introduce the basic operators used in Java programming, and then show you how to control the flow of your program.

Exercises

Normally, exercises will be distributed throughout the chapters, but in this chapter you were learning how to write basic programs so all the exercises were delayed until the end.

The number in parentheses after each exercise number is an indicator of how difficult the exercise is, in a ranking from 1-10.

Solutions to selected exercises can be found in the electronic document *The Thinking in Java Annotated Solution Guide*, available for sale from *www.MindView.net*.

Exercise 1: (2) Create a class containing an **int** and a **char** that are not initialized, and print their values to verify that Java performs default initialization.

Exercise 2: (1) Following the **HelloDate.java** example in this chapter, create a "hello, world" program that simply displays that statement. You need only a single method in your class (the "main" one that gets executed when the program starts). Remember to make it **static** and to include the argument list, even though you don't use the argument list. Compile the program with **javac** and run it using **java**. If you are using a different development environment than the JDK, learn how to compile and run programs in that environment.

Exercise 3: (1) Find the code fragments involving **ATypeName** and turn them into a program that compiles and runs.

Exercise 4: (1) Turn the **DataOnly** code fragments into a program that compiles and runs.

Exercise 5: (1) Modify the previous exercise so that the values of the data in **DataOnly** are assigned to and printed in **main( )**.

Exercise 6: (2) Write a program that includes and calls the **storage( )** method defined as a code fragment in this chapter.

Exercise 7: (1) Turn the **Incrementable** code fragments into a working program.

Exercise 8: (3) Write a program that demonstrates that, no matter how many objects you create of a particular class, there is only one instance of a particular **static** field in that class.

Exercise 9: (2) Write a program that demonstrates that autoboxing works for all the primitive types and their wrappers.

Exercise 10: (2) Write a program that prints three arguments taken from the command line. To do this, you'll need to index into the command-line array of **String**s.

Exercise 11: (1) Turn the **AllTheColorsOfTheRainbow** example into a program that compiles and runs.

Exercise 12: (2) Find the code for the second version of **HelloDate.java**, which is the simple comment documentation example. Execute **Javadoc** on the file and view the results with your Web browser.

Exercise 13: (1) Run **Documentation1.java**, **Documentation2.java**, and **Documentation3.java** through **Javadoc**. Verify the resulting documentation with your Web browser.

Exercise 14: (1) Add an HTML list of items to the documentation in the previous exercise.

Exercise 15: (1) Take the program in Exercise 2 and add comment documentation to it. Extract this comment documentation into an HTML file using **Javadoc** and view it with your Web browser.

Exercise 16: (1) In the *Initialization & Cleanup* chapter, locate the **Overloading.java** example and add Javadoc documentation. Extract this comment documentation into an HTML file using **Javadoc** and view it with your Web browser.

Operators

At the lowest level, data in Java is manipulated using operators.

Because Java was inherited from C++, most of these operators will be familiar to C and C++ programmers. Java has also added some improvements and simplifications.

If you're familiar with C or C++ syntax, you can skim through this chapter and the next, looking for places where Java is different from those languages. However, if you find yourself floundering a bit in these two chapters, make sure you go through the multimedia seminar *Thinking in C*, freely downloadable from *www.MindView.net*. It contains audio lectures, slides, exercises, and solutions specifically designed to bring you up to speed with the fundamentals necessary to learn Java.

Simpler print statements

In the previous chapter, you were introduced to the Java print statement:

```
System.out.println("Rather a lot to type");
```

You may observe that this is not only a lot to type (and thus many redundant tendon hits), but also rather noisy to read. Most languages before and after Java have taken a much simpler approach to such a commonly used statement.

The *Access Control* chapter introduces the concept of the *static import* that was added to Java SE5, and creates a tiny library to simplify writing print statements. However, you don't need to know those details in order to begin using that library. We can rewrite the program from the last chapter using this new library:

```
//: operators/HelloDate.java
import java.util.*;
import static net.mindview.util.Print.*;

public class HelloDate {
  public static void main(String[] args) {
    print("Hello, it's: ");
    print(new Date());
  }
} /* Output: (55% match)
Hello, it's:
Wed Oct 05 14:39:05 MDT 2005
*///:~
```

The results are much cleaner. Notice the insertion of the **static** keyword in the second **import** statement.

In order to use this library, you must download this book's code package from *www.MindView.net* or one of its mirrors. Unzip the code tree and add the root directory of that code tree to your computer's CLASSPATH environment variable. (You'll eventually get a full introduction to the classpath, but you might as well get used to struggling with it early. Alas, it is one of the more common battles you will have with Java.)

Although the use of **net.mindview.util.Print** nicely simplifies most code, it is not justifiable everywhere. If there are only a small number of print statements in a program, I forego the **import** and write out the full **System.out.println( )**.

Exercise 1: (1) Write a program that uses the "short" and normal form of print statement.

Using Java operators

An operator takes one or more arguments and produces a new value. The arguments are in a different form than ordinary method calls, but the effect is the same. Addition and unary plus (**+**), subtraction and unary minus (**-**), multiplication (*****), division (**/**), and assignment (**=**) all work much the same in any programming language.

All operators produce a value from their operands. In addition, some operators change the value of an operand. This is called a *side effect*. The most common use for operators that modify their operands is to generate the side effect, but you should keep in mind that the value produced is available for your use, just as in operators without side effects.

Almost all operators work only with primitives. The exceptions are '**=**', '**==**' and '**!=**', which work with all objects (and are a point of confusion for objects). In addition, the **String** class supports '**+**' and '**+=**'.

Precedence

Operator precedence defines how an expression evaluates when several operators are present. Java has specific rules that determine the order of evaluation. The easiest one to remember is that multiplication and division happen before addition and subtraction. Programmers often forget the other precedence rules, so you should use parentheses to make the order of evaluation explicit. For example, look at statements **(1)** and **(2)**:

```
//: operators/Precedence.java

public class Precedence {
  public static void main(String[] args) {
    int x = 1, y = 2, z = 3;
    int a = x + y - 2/2 + z;           // (1)
    int b = x + (y - 2)/(2 + z);       // (2)
    System.out.println("a = " + a + " b = " + b);
  }
} /* Output:
a = 5 b = 1
*///:~
```

These statements look roughly the same, but from the output you can see that they have very different meanings which depend on the use of parentheses.

Notice that the **System.out.println( )** statement involves the '**+**' operator. In this context, '**+**' means "string concatenation" and, if necessary, "string conversion." When the compiler sees a **String** followed by a '**+**' followed by a non-**String**, it attempts to convert the non-**String** into a **String**. As you can see from the output, it successfully converts from **int** into **String** for **a** and **b**.

Assignment

Assignment is performed with the operator =. It means "Take the value of the right-hand side (often called the *rvalue*) and copy it into the left-hand side (often called the *lvalue*)." An rvalue is any constant, variable, or expression that produces a value, but an lvalue must be a distinct, named variable. (That is, there must be a physical space to store the value.) For instance, you can assign a constant value to a variable:

```
a = 4;
```

but you cannot assign anything to a constant value—it cannot be an lvalue. (You can't say **4 = a;**.)

Assignment of primitives is quite straightforward. Since the primitive holds the actual value and not a reference to an object, when you assign primitives, you copy the contents from one place to another. For example, if you say **a = b** for primitives, then the contents of **b** are copied into **a**. If you then go on to modify **a**, **b** is naturally unaffected by this modification. As a programmer, this is what you can expect for most situations.

When you assign objects, however, things change. Whenever you manipulate an object, what you're manipulating is the reference, so when you assign "from one object to another," you're actually copying a reference from one place to another. This means that if you say **c = d** for objects, you end up with both **c** and **d** pointing to the object that, originally, only **d** pointed to. Here's an example that demonstrates this behavior:

```
//: operators/Assignment.java
// Assignment with objects is a bit tricky.
import static net.mindview.util.Print.*;

class Tank {
  int level;
}

public class Assignment {
  public static void main(String[] args) {
    Tank t1 = new Tank();
    Tank t2 = new Tank();
    t1.level = 9;
    t2.level = 47;
    print("1: t1.level: " + t1.level +
          ", t2.level: " + t2.level);
    t1 = t2;
    print("2: t1.level: " + t1.level +
          ", t2.level: " + t2.level);
    t1.level = 27;
    print("3: t1.level: " + t1.level +
          ", t2.level: " + t2.level);
  }
} /* Output:
1: t1.level: 9, t2.level: 47
2: t1.level: 47, t2.level: 47
3: t1.level: 27, t2.level: 27
*///:~
```

The **Tank** class is simple, and two instances (**t1** and **t2**) are created within **main( )**. The **level** field within each **Tank** is given a different value, and then **t2** is assigned to **t1**, and **t1** is changed. In many programming languages you expect **t1** and **t2** to be independent at all times, but

because you've assigned a reference, changing the **t1** object appears to change the **t2** object as well! This is because both **t1** and **t2** contain the same reference, which is pointing to the same object. (The original reference that was in **t1**, that pointed to the object holding a value of 9, was overwritten during the assignment and effectively lost; its object will be cleaned up by the garbage collector.)

This phenomenon is often called *aliasing*, and it's a fundamental way that Java works with objects. But what if you don't want aliasing to occur in this case? You could forego the assignment and say:

```
t1.level = t2.level;
```

This retains the two separate objects instead of discarding one and tying **t1** and **t2** to the same object. You'll soon realize that manipulating the fields within objects is messy and goes against good object-oriented design principles. This is a nontrivial topic, so you should keep in mind that assignment for objects can add surprises.

Exercise 2: (1) Create a class containing a **float** and use it to demonstrate aliasing.

Aliasing during method calls

Aliasing will also occur when you pass an object into a method:

```
//: operators/PassObject.java
// Passing objects to methods may not be
// what you're used to.
import static net.mindview.util.Print.*;

class Letter {
  char c;
}

public class PassObject {
  static void f(Letter y) {
    y.c = 'z';
  }
  public static void main(String[] args) {
    Letter x = new Letter();
    x.c = 'a';
    print("1: x.c: " + x.c);
    f(x);
    print("2: x.c: " + x.c);
  }
} /* Output:
1: x.c: a
2: x.c: z
*///:~
```

In many programming languages, the method **f( )** would appear to be making a copy of its argument **Letter y** inside the scope of the method. But once again a reference is being passed, so the line

```
y.c = 'z';
```

is actually changing the object outside of **f( )**.

Aliasing and its solution is a complex issue which is covered in one of the online supplements for this book. However, you should be aware of it at this point so you can watch for pitfalls.

Exercise 3: (1) Create a class containing a **float** and use it to demonstrate aliasing during method calls.

Mathematical operators

The basic mathematical operators are the same as the ones available in most programming languages: addition (**+**), subtraction (**-**), division (**/**), multiplication (*****) and modulus (**%**, which produces the remainder from integer division). Integer division truncates, rather than rounds, the result.

Java also uses the shorthand notation from C/C++ that performs an operation and an assignment at the same time. This is denoted by an operator followed by an equal sign, and is consistent with all the operators in the language (whenever it makes sense). For example, to add 4 to the variable **x** and assign the result to **x**, use: **x += 4**.

This example shows the use of the mathematical operators:

```
//: operators/MathOps.java
// Demonstrates the mathematical operators.
import java.util.*;
import static net.mindview.util.Print.*;

public class MathOps {
  public static void main(String[] args) {
    // Create a seeded random number generator:
    Random rand = new Random(47);
    int i, j, k;
    // Choose value from 1 to 100:
    j = rand.nextInt(100) + 1;
    print("j : " + j);
    k = rand.nextInt(100) + 1;
    print("k : " + k);
    i = j + k;
    print("j + k : " + i);
    i = j - k;
    print("j - k : " + i);
    i = k / j;
    print("k / j : " + i);
    i = k * j;
    print("k * j : " + i);
    i = k % j;
    print("k % j : " + i);
    j %= k;
    print("j %= k : " + j);
    // Floating-point number tests:
    float u, v, w; // Applies to doubles, too
    v = rand.nextFloat();
    print("v : " + v);
    w = rand.nextFloat();
    print("w : " + w);
    u = v + w;
    print("v + w : " + u);
    u = v - w;
    print("v - w : " + u);
    u = v * w;
    print("v * w : " + u);
    u = v / w;
```

```
    print("v / w : " + u);
    // The following also works for char,
    // byte, short, int, long, and double:
    u += v;
    print("u += v : " + u);
    u -= v;
    print("u -= v : " + u);
    u *= v;
    print("u *= v : " + u);
    u /= v;
    print("u /= v : " + u);
  }
} /* Output:
j : 59
k : 56
j + k : 115
j - k : 3
k / j : 0
k * j : 3304
k % j : 56
j %= k : 3
v : 0.5309454
w : 0.0534122
v + w : 0.5843576
v - w : 0.47753322
v * w : 0.028358962
v / w : 9.940527
u += v : 10.471473
u -= v : 9.940527
u *= v : 5.2778773
u /= v : 9.940527
*///:~
```

To generate numbers, the program first creates a **Random** object. If you create a **Random** object with no arguments, Java uses the current time as a seed for the random number generator, and will thus produce different output for each execution of the program. However, in the examples in this book, it is important that the output shown at the end of the examples be as consistent as possible, so that this output can be verified with external tools. By providing a *seed* (an initialization value for the random number generator that will always produce the same sequence for a particular seed value) when creating the **Random** object, the same random numbers will be generated each time the program is executed, so the output is verifiable.[1] To generate more varying output, feel free to remove the seed in the examples in the book.

The program generates a number of different types of random numbers with the **Random** object simply by calling the methods **nextInt( )** and **nextFloat( )** (you can also call **nextLong( )** or **nextDouble( )**). The argument to **nextInt( )** sets the upper bound on the generated number. The lower bound is zero, which we don't want because of the possibility of a divide-by-zero, so the result is offset by one.

Exercise 4: (2) Write a program that calculates velocity using a constant distance and a constant time.

[1] The number 47 was considered a "magic number" at a college I attended, and it stuck.

Unary minus and plus operators

The unary minus (**-**) and unary plus (**+**) are the same operators as binary minus and plus. The compiler figures out which use is intended by the way you write the expression. For instance, the statement

```
x = -a;
```

has an obvious meaning. The compiler is able to figure out:

```
x = a * -b;
```

but the reader might get confused, so it is sometimes clearer to say:

```
x = a * (-b);
```

Unary minus inverts the sign on the data. Unary plus provides symmetry with unary minus, although it doesn't have any effect.

Auto increment and decrement

Java, like C, has a number of shortcuts. Shortcuts can make code much easier to type, and either easier or harder to read.

Two of the nicer shortcuts are the increment and decrement operators (often referred to as the auto-increment and auto-decrement operators). The decrement operator is **--** and means "decrease by one unit." The increment operator is **++** and means "increase by one unit." If **a** is an **int**, for example, the expression **++a** is equivalent to (**a = a + 1**). Increment and decrement operators not only modify the variable, but also produce the value of the variable as a result.

There are two versions of each type of operator, often called the *prefix* and *postfix* versions. *Pre-increment* means the **++** operator appears before the variable, and *post-increment* means the **++** operator appears after the variable. Similarly, *pre-decrement* means the **--** operator appears before the variable, and *post-decrement* means the **--** operator appears after the variable. For pre-increment and pre-decrement (i.e., **++a** or **--a**), the operation is performed and the value is produced. For post-increment and post-decrement (i.e., **a++** or **a--**), the value is produced, then the operation is performed. As an example:

```
//: operators/AutoInc.java
// Demonstrates the ++ and -- operators.
import static net.mindview.util.Print.*;

public class AutoInc {
  public static void main(String[] args) {
    int i = 1;
    print("i : " + i);
    print("++i : " + ++i); // Pre-increment
    print("i++ : " + i++); // Post-increment
    print("i : " + i);
    print("--i : " + --i); // Pre-decrement
    print("i-- : " + i--); // Post-decrement
    print("i : " + i);
  }
} /* Output:
i : 1
++i : 2
i++ : 2
```

```
i : 3
--i : 2
i-- : 2
i : 1
*///:~
```

You can see that for the prefix form, you get the value after the operation has been performed, but with the postfix form, you get the value before the operation is performed. These are the only operators, other than those involving assignment, that have side effects—they change the operand rather than using just its value.

The increment operator is one explanation for the name C++, implying "one step beyond C." In an early Java speech, Bill Joy (one of the Java creators), said that "Java=C++--" (C plus plus minus minus), suggesting that Java is C++ with the unnecessary hard parts removed, and therefore a much simpler language. As you progress in this book, you'll see that many parts are simpler, and yet in other ways Java isn't much easier than C++.

Relational operators

Relational operators generate a **boolean** result. They evaluate the relationship between the values of the operands. A relational expression produces **true** if the relationship is true, and **false** if the relationship is untrue. The relational operators are less than (<), greater than (>), less than or equal to (<=), greater than or equal to (>=), equivalent (==) and not equivalent (!=). Equivalence and nonequivalence work with all primitives, but the other comparisons won't work with type **boolean**. Because **boolean** values can only be **true** or **false**, "greater than" and "less than" doesn't make sense.

Testing object equivalence

The relational operators == and != also work with all objects, but their meaning often confuses the first-time Java programmer. Here's an example:

```
//: operators/Equivalence.java

public class Equivalence {
  public static void main(String[] args) {
    Integer n1 = new Integer(47);
    Integer n2 = new Integer(47);
    System.out.println(n1 == n2);
    System.out.println(n1 != n2);
  }
} /* Output:
false
true
*///:~
```

The statement **System.out.println(n1 == n2)** will print the result of the **boolean** comparison within it. Surely the output should be "true" and then "false," since both **Integer** objects are the same. But while the *contents* of the objects are the same, the references are not the same. The operators == and != compare object references, so the output is actually "false" and then "true." Naturally, this surprises people at first.

What if you want to compare the actual contents of an object for equivalence? You must use the special method **equals( )** that exists for all objects (not primitives, which work fine with == and !=). Here's how it's used:

```
//: operators/EqualsMethod.java

public class EqualsMethod {
  public static void main(String[] args) {
    Integer n1 = new Integer(47);
    Integer n2 = new Integer(47);
    System.out.println(n1.equals(n2));
  }
} /* Output:
true
*///:~
```

The result is now what you expect. Ah, but it's not as simple as that. If you create your own class, like this:

```
//: operators/EqualsMethod2.java
// Default equals() does not compare contents.

class Value {
  int i;
}

public class EqualsMethod2 {
  public static void main(String[] args) {
    Value v1 = new Value();
    Value v2 = new Value();
    v1.i = v2.i = 100;
    System.out.println(v1.equals(v2));
  }
} /* Output:
false
*///:~
```

things are confusing again: The result is **false**. This is because the default behavior of **equals( )** is to compare references. So unless you *override* **equals( )** in your new class you won't get the desired behavior. Unfortunately, you won't learn about overriding until the *Reusing Classes* chapter and about the proper way to define **equals( )** until the *Containers in Depth* chapter, but being aware of the way **equals( )** behaves might save you some grief in the meantime.

Most of the Java library classes implement **equals( )** so that it compares the contents of objects instead of their references.

Exercise 5: (2) Create a class called **Dog** containing two **String**s: **name** and **says**. In **main( )**, create two dog objects with names "spot" (who says, "Ruff!") and "scruffy" (who says, "Wurf!"). Then display their names and what they say.

Exercise 6: (3) Following Exercise 5, create a new **Dog** reference and assign it to spot's object. Test for comparison using **==** and **equals( )** for all references.

Logical operators

Each of the logical operators AND (**&&**), OR (**||**) and NOT (**!**) produces a **boolean** value of **true** or **false** based on the logical relationship of its arguments. This example uses the relational and logical operators:

```
//: operators/Bool.java
// Relational and logical operators.
```

```
import java.util.*;
import static net.mindview.util.Print.*;

public class Bool {
  public static void main(String[] args) {
    Random rand = new Random(47);
    int i = rand.nextInt(100);
    int j = rand.nextInt(100);
    print("i = " + i);
    print("j = " + j);
    print("i > j is " + (i > j));
    print("i < j is " + (i < j));
    print("i >= j is " + (i >= j));
    print("i <= j is " + (i <= j));
    print("i == j is " + (i == j));
    print("i != j is " + (i != j));
    // Treating an int as a boolean is not legal Java:
//! print("i && j is " + (i && j));
//! print("i || j is " + (i || j));
//! print("!i is " + !i);
    print("(i < 10) && (j < 10) is "
       + ((i < 10) && (j < 10)) );
    print("(i < 10) || (j < 10) is "
       + ((i < 10) || (j < 10)) );
  }
} /* Output:
i = 58
j = 55
i > j is true
i < j is false
i >= j is true
i <= j is false
i == j is false
i != j is true
(i < 10) && (j < 10) is false
(i < 10) || (j < 10) is false
*///:~
```

You can apply AND, OR, or NOT to **boolean** values only. You can't use a non-**boolean** as if it were a **boolean** in a logical expression as you can in C and C++. You can see the failed attempts at doing this commented out with a '**//!**' (this comment syntax enables automatic removal of comments to facilitate testing). The subsequent expressions, however, produce **boolean** values using relational comparisons, then use logical operations on the results.

Note that a **boolean** value is automatically converted to an appropriate text form if it is used where a **String** is expected.

You can replace the definition for **int** in the preceding program with any other primitive data type except **boolean**. Be aware, however, that the comparison of floating point numbers is very strict. A number that is the tiniest fraction different from another number is still "not equal." A number that is the tiniest bit above zero is still nonzero.

Exercise 7: (3) Write a program that simulates coin-flipping.

Short-circuiting

When dealing with logical operators, you run into a phenomenon called "short-circuiting." This means that the expression will be evaluated only *until* the truth or falsehood of the entire expression can be unambiguously determined. As a result, the latter parts of a logical expression might not be evaluated. Here's an example that demonstrates short-circuiting:

```
//: operators/ShortCircuit.java
// Demonstrates short-circuiting behavior
// with logical operators.
import static net.mindview.util.Print.*;

public class ShortCircuit {
  static boolean test1(int val) {
    print("test1(" + val + ")");
    print("result: " + (val < 1));
    return val < 1;
  }
  static boolean test2(int val) {
    print("test2(" + val + ")");
    print("result: " + (val < 2));
    return val < 2;
  }
  static boolean test3(int val) {
    print("test3(" + val + ")");
    print("result: " + (val < 3));
    return val < 3;
  }
  public static void main(String[] args) {
    boolean b = test1(0) && test2(2) && test3(2);
    print("expression is " + b);
  }
} /* Output:
test1(0)
result: true
test2(2)
result: false
expression is false
*///:~
```

Each test performs a comparison against the argument and returns **`true`** or **`false`**. It also prints information to show you that it's being called. The tests are used in the expression:

```
test1(0) && test2(2) && test3(2)
```

You might naturally think that all three tests would be executed, but the output shows otherwise. The first test produced a **`true`** result, so the expression evaluation continues. However, the second test produced a **`false`** result. Since this means that the whole expression must be **`false`**, why continue evaluating the rest of the expression? It might be expensive. The reason for short-circuiting, in fact, is that you can get a potential performance increase if all the parts of a logical expression do not need to be evaluated.

Literals

Ordinarily, when you insert a literal value into a program, the compiler knows exactly what type to make it. Sometimes, however, the type is ambiguous. When this happens, you must guide the

compiler by adding some extra information in the form of characters associated with the literal value. The following code shows these characters:

```
//: operators/Literals.java
import static net.mindview.util.Print.*;

public class Literals {
  public static void main(String[] args) {
    int i1 = 0x2f; // Hexadecimal (lowercase)
    print("i1: " + Integer.toBinaryString(i1));
    int i2 = 0X2F; // Hexadecimal (uppercase)
    print("i2: " + Integer.toBinaryString(i2));
    int i3 = 0177; // Octal (leading zero)
    print("i3: " + Integer.toBinaryString(i3));
    char c = 0xffff; // max char hex value
    print("c: " + Integer.toBinaryString(c));
    byte b = 0x7f; // max byte hex value
    print("b: " + Integer.toBinaryString(b));
    short s = 0x7fff; // max short hex value
    print("s: " + Integer.toBinaryString(s));
    long n1 = 200L; // long suffix
    long n2 = 200l; // long suffix (but can be confusing)
    long n3 = 200;
    float f1 = 1;
    float f2 = 1F; // float suffix
    float f3 = 1f; // float suffix
    double d1 = 1d; // double suffix
    double d2 = 1D; // double suffix
    // (Hex and Octal also work with long)
  }
} /* Output:
i1: 101111
i2: 101111
i3: 1111111
c: 1111111111111111
b: 1111111
s: 111111111111111
*///:~
```

A trailing character after a literal value establishes its type. Uppercase or lowercase **L** means **long** (however, using a lowercase **l** is confusing because it can look like the number one). Uppercase or lowercase **F** means **float**. Uppercase or lowercase **D** means **double**.

Hexadecimal (base 16), which works with all the integral data types, is denoted by a leading **0x** or **0X** followed by **0-9** or **a-f** either in uppercase or lowercase. If you try to initialize a variable with a value bigger than it can hold (regardless of the numerical form of the value), the compiler will give you an error message. Notice in the preceding code the maximum possible hexadecimal values for **char**, **byte**, and **short**. If you exceed these, the compiler will automatically make the value an **int** and tell you that you need a narrowing *cast* for the assignment (casts are defined later in this chapter). You'll know you've stepped over the line.

Octal (base 8) is denoted by a leading zero in the number and digits from 0-7.

There is no literal representation for binary numbers in C, C++, or Java. However, when working with hexadecimal and octal notation, it's useful to display the binary form of the results. This is easily accomplished with the **static toBinaryString( )** methods from the **Integer** and **Long**

classes. Notice that when passing smaller types to **Integer.toBinaryString( )**, the type is automatically converted to an **int**.

Exercise 8: (2) Show that hex and octal notations work with long values. Use **Long.toBinaryString( )** to display the results.

Exponential notation

Exponents use a notation that I've always found rather dismaying:

```
//: operators/Exponents.java
// "e" means "10 to the power."

public class Exponents {
  public static void main(String[] args) {
    // Uppercase and lowercase 'e' are the same:
    float expFloat = 1.39e-43f;
    expFloat = 1.39E-43f;
    System.out.println(expFloat);
    double expDouble = 47e47d; // 'd' is optional
    double expDouble2 = 47e47; // Automatically double
    System.out.println(expDouble);
  }
} /* Output:
1.39E-43
4.7E48
*///:~
```

In science and engineering, 'e' refers to the base of natural logarithms, approximately 2.718. (A more precise **double** value is available in Java as **Math.E**.) This is used in exponentiation expressions such as $1.39 \times e^{-43}$, which means 1.39×2.718^{-43}. However, when the FORTRAN programming language was invented, they decided that **e** would mean "ten to the power," which is an odd decision because FORTRAN was designed for science and engineering, and one would think its designers would be sensitive about introducing such an ambiguity.[2] At any rate, this custom was followed in C, C++ and now Java. So if you're used to thinking in terms of **e** as the base of natural logarithms, you must do a mental translation when you see an expression such as **1.39 e-43f** in Java; it means 1.39×10^{-43}.

Note that you don't need to use the trailing character when the compiler can figure out the appropriate type. With

```
long n3 = 200;
```

there's no ambiguity, so an **L** after the 200 would be superfluous. However, with

```
float f4 = 1e-43f; // 10 to the power
```

[2] John Kirkham writes, "I started computing in 1962 using FORTRAN II on an IBM 1620. At that time, and throughout the 1960s and into the 1970s, FORTRAN was an all uppercase language. This probably started because many of the early input devices were old teletype units that used 5 bit Baudot code, which had no lowercase capability. The 'E' in the exponential notation was also always uppercase and was never confused with the natural logarithm base 'e', which is always lowercase. The 'E' simply stood for exponential, which was for the base of the number system used—usually 10. At the time octal was also widely used by programmers. Although I never saw it used, if I had seen an octal number in exponential notation I would have considered it to be base 8. The first time I remember seeing an exponential using a lowercase 'e' was in the late 1970s and I also found it confusing. The problem arose as lowercase crept into FORTRAN, not at its beginning. We actually had functions to use if you really wanted to use the natural logarithm base, but they were all uppercase."

the compiler normally takes exponential numbers as doubles, so without the trailing **f**, it will give you an error telling you that you must use a cast to convert **double** to **float**.

Exercise 9: (1) Display the largest and smallest numbers for both **float** and **double** exponential notation.

Bitwise operators

The bitwise operators allow you to manipulate individual bits in an integral primitive data type. Bitwise operators perform Boolean algebra on the corresponding bits in the two arguments to produce the result.

The bitwise operators come from C's low-level orientation, where you often manipulate hardware directly and must set the bits in hardware registers. Java was originally designed to be embedded in TV set-top boxes, so this low-level orientation still made sense. However, you probably won't use the bitwise operators much.

The bitwise AND operator (**&**) produces a one in the output bit if both input bits are one; otherwise, it produces a zero. The bitwise OR operator (|) produces a one in the output bit if either input bit is a one and produces a zero only if both input bits are zero. The bitwise EXCLUSIVE OR, or XOR (^), produces a one in the output bit if one or the other input bit is a one, but not both. The bitwise NOT (~, also called the *ones complement* operator) is a unary operator; it takes only one argument. (All other bitwise operators are binary operators.) Bitwise NOT produces the opposite of the input bit—a one if the input bit is zero, a zero if the input bit is one.

The bitwise operators and logical operators use the same characters, so it is helpful to have a mnemonic device to help you remember the meanings: Because bits are "small," there is only one character in the bitwise operators.

Bitwise operators can be combined with the = sign to unite the operation and assignment: **&=**, **|=** and **^=** are all legitimate. (Since ~ is a unary operator, it cannot be combined with the = sign.)

The **boolean** type is treated as a one-bit value, so it is somewhat different. You can perform a bitwise AND, OR, and XOR, but you can't perform a bitwise NOT (presumably to prevent confusion with the logical NOT). For **boolean**s, the bitwise operators have the same effect as the logical operators except that they do not short circuit. Also, bitwise operations on **boolean**s include an XOR logical operator that is not included under the list of "logical" operators. You cannot use **boolean**s in shift expressions, which are described next.

Exercise 10: (3) Write a program with two constant values, one with alternating binary ones and zeroes, with a zero in the least-significant digit, and the second, also alternating, with a one in the least-significant digit (hint: It's easiest to use hexadecimal constants for this). Take these two values and combine them in all possible ways using the bitwise operators, and display the results using **Integer.toBinaryString( )**.

Shift operators

The shift operators also manipulate bits. They can be used solely with primitive, integral types. The left-shift operator (<<) produces the operand to the left of the operator after it has been shifted to the left by the number of bits specified to the right of the operator (inserting zeroes at the lower-order bits). The signed right-shift operator (>>) produces the operand to the left of the operator after it has been shifted to the right by the number of bits specified to the right of the

operator. The signed right shift >> uses *sign extension*: If the value is positive, zeroes are inserted at the higher-order bits; if the value is negative, ones are inserted at the higher-order bits. Java has also added the unsigned right shift >>>, which uses *zero extension*: Regardless of the sign, zeroes are inserted at the higher-order bits. This operator does not exist in C or C++.

If you shift a **char**, **byte**, or **short**, it will be promoted to **int** before the shift takes place, and the result will be an **int**. Only the five low-order bits of the right-hand side will be used. This prevents you from shifting more than the number of bits in an **int**. If you're operating on a **long**, you'll get a **long** result. Only the six low-order bits of the right-hand side will be used, so you can't shift more than the number of bits in a **long**.

Shifts can be combined with the equal sign (<<= or >>= or >>>=). The lvalue is replaced by the lvalue shifted by the rvalue. There is a problem, however, with the unsigned right shift combined with assignment. If you use it with **byte** or **short**, you don't get the correct results. Instead, these are promoted to **int** and right shifted, but then truncated as they are assigned back into their variables, so you get **-1** in those cases. The following example demonstrates this:

```
//: operators/URShift.java
// Test of unsigned right shift.
import static net.mindview.util.Print.*;

public class URShift {
  public static void main(String[] args) {
    int i = -1;
    print(Integer.toBinaryString(i));
    i >>>= 10;
    print(Integer.toBinaryString(i));
    long l = -1;
    print(Long.toBinaryString(l));
    l >>>= 10;
    print(Long.toBinaryString(l));
    short s = -1;
    print(Integer.toBinaryString(s));
    s >>>= 10;
    print(Integer.toBinaryString(s));
    byte b = -1;
    print(Integer.toBinaryString(b));
    b >>>= 10;
    print(Integer.toBinaryString(b));
    b = -1;
    print(Integer.toBinaryString(b));
    print(Integer.toBinaryString(b>>>10));
  }
} /* Output:
11111111111111111111111111111111
11111111111111111111
11111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111
*///:~
```

In the last shift, the resulting value is not assigned back into **b**, but is printed directly, so the correct behavior occurs.

Here's an example that demonstrates the use of all the operators involving bits:

```
//: operators/BitManipulation.java
// Using the bitwise operators.
import java.util.*;
import static net.mindview.util.Print.*;

public class BitManipulation {
  public static void main(String[] args) {
    Random rand = new Random(47);
    int i = rand.nextInt();
    int j = rand.nextInt();
    printBinaryInt("-1", -1);
    printBinaryInt("+1", +1);
    int maxpos = 2147483647;
    printBinaryInt("maxpos", maxpos);
    int maxneg = -2147483648;
    printBinaryInt("maxneg", maxneg);
    printBinaryInt("i", i);
    printBinaryInt("~i", ~i);
    printBinaryInt("-i", -i);
    printBinaryInt("j", j);
    printBinaryInt("i & j", i & j);
    printBinaryInt("i | j", i | j);
    printBinaryInt("i ^ j", i ^ j);
    printBinaryInt("i << 5", i << 5);
    printBinaryInt("i >> 5", i >> 5);
    printBinaryInt("(~i) >> 5", (~i) >> 5);
    printBinaryInt("i >>> 5", i >>> 5);
    printBinaryInt("(~i) >>> 5", (~i) >>> 5);

    long l = rand.nextLong();
    long m = rand.nextLong();
    printBinaryLong("-1L", -1L);
    printBinaryLong("+1L", +1L);
    long ll = 9223372036854775807L;
    printBinaryLong("maxpos", ll);
    long lln = -9223372036854775808L;
    printBinaryLong("maxneg", lln);
    printBinaryLong("l", l);
    printBinaryLong("~l", ~l);
    printBinaryLong("-l", -l);
    printBinaryLong("m", m);
    printBinaryLong("l & m", l & m);
    printBinaryLong("l | m", l | m);
    printBinaryLong("l ^ m", l ^ m);
    printBinaryLong("l << 5", l << 5);
    printBinaryLong("l >> 5", l >> 5);
    printBinaryLong("(~l) >> 5", (~l) >> 5);
    printBinaryLong("l >>> 5", l >>> 5);
    printBinaryLong("(~l) >>> 5", (~l) >>> 5);
  }
  static void printBinaryInt(String s, int i) {
    print(s + ", int: " + i + ", binary:\n   " +
```

```
      Integer.toBinaryString(i));
  }
  static void printBinaryLong(String s, long l) {
    print(s + ", long: " + l + ", binary:\n    " +
      Long.toBinaryString(l));
  }
} /* Output:
-1, int: -1, binary:
   11111111111111111111111111111111
+1, int: 1, binary:
   1
maxpos, int: 2147483647, binary:
   01111111111111111111111111111111
maxneg, int: -2147483648, binary:
   10000000000000000000000000000000
i, int: -1172028779, binary:
   10111010001001000100001010010101
~i, int: 1172028778, binary:
   01000101110110111011110101101010
-i, int: 1172028779, binary:
   01000101110110111011110101101011
j, int: 1717241110, binary:
   01100110010110110000010100010110
i & j, int: 570425364, binary:
   00010000000000000000000000010100
i | j, int: -25213033, binary:
   11111110011111111010001110010111
i ^ j, int: -595638397, binary:
   11011100011111111010001111000011
i << 5, int: 1149784736, binary:
   01000100100010000101001010100000
i >> 5, int: -36625900, binary:
   11111101111010001001000010000100
(~i) >> 5, int: 36625899, binary:
   00000101110110111011101011
i >>> 5, int: 97591828, binary:
   00000101110110111011101011
(~i) >>> 5, int: 36625899, binary:
   00000101110110111011101011
...
*///:~
```

The two methods at the end, **printBinaryInt( )** and **printBinaryLong( )**, take an **int** or a **long**, respectively, and print it out in binary format along with a descriptive string. As well as demonstrating the effect of all the bitwise operators for **int** and **long**, this example also shows the minimum, maximum, +1, and -1 values for **int** and **long** so you can see what they look like. Note that the high bit represents the sign: 0 means positive and 1 means negative. The output for the **int** portion is displayed above.

The binary representation of the numbers is referred to as *signed twos complement*.

Exercise 11: (3) Start with a number that has a binary one in the most significant position (hint: Use a hexadecimal constant). Using the signed right-shift operator, right shift it all the way through all of its binary positions, each time displaying the result using **Integer.toBinaryString( )**.

Exercise 12: (3) Start with a number that is all binary ones. Left shift it, then use the unsigned right-shift operator to right shift through all of its binary positions, each time displaying the result using **Integer.toBinaryString( )**.

Exercise 13: (1) Write a method that displays **char** values in binary form. Demonstrate it using several different characters.

Ternary **if-else** operator

The *ternary* operator, also called the *conditional* operator, is unusual because it has three operands. It is truly an operator because it produces a value, unlike the ordinary **if-else** statement that you'll see in the next section of this chapter. The expression is of the form:

```
boolean-exp ? value0 : value1
```

If *boolean-exp* evaluates to **true**, *value0* is evaluated, and its result becomes the value produced by the operator. If *boolean-exp* is **false**, *value1* is evaluated and its result becomes the value produced by the operator.

Of course, you could use an ordinary **if-else** statement (described later), but the ternary operator is much terser. Although C (where this operator originated) prides itself on being a terse language, and the ternary operator might have been introduced partly for efficiency, you should be somewhat wary of using it on an everyday basis—it's easy to produce unreadable code.

The conditional operator is different from **if-else** because it produces a value. Here's an example comparing the two:

```
//: operators/TernaryIfElse.java
import static net.mindview.util.Print.*;

public class TernaryIfElse {
  static int ternary(int i) {
    return i < 10 ? i * 100 : i * 10;
  }
  static int standardIfElse(int i) {
    if(i < 10)
      return i * 100;
    else
      return i * 10;
  }
  public static void main(String[] args) {
    print(ternary(9));
    print(ternary(10));
    print(standardIfElse(9));
    print(standardIfElse(10));
  }
} /* Output:
900
100
900
100
*///:~
```

You can see that this code in **ternary( )** is more compact than what you'd need to write without the ternary operator, in **standardIfElse( )**. However, **standardIfElse( )** is easier to understand, and doesn't require a lot more typing. So be sure to ponder your reasons when

choosing the ternary operator—it's generally warranted when you're setting a variable to one of two values.

String operator + and +=

There's one special usage of an operator in Java: The **+** and **+=** operators can be used to concatenate strings, as you've already seen. It seems a natural use of these operators even though it doesn't fit with the traditional way that they are used.

This capability seemed like a good idea in C++, so *operator overloading* was added to C++ to allow the C++ programmer to add meanings to almost any operator. Unfortunately, operator overloading combined with some of the other restrictions in C++ turns out to be a fairly complicated feature for programmers to design into their classes. Although operator overloading would have been much simpler to implement in Java than it was in C++ (as has been demonstrated in the C# language, which *does* have straightforward operator overloading), this feature was still considered too complex, so Java programmers cannot implement their own overloaded operators like C++ and C# programmers can.

The use of the **String** operators has some interesting behavior. If an expression begins with a **String**, then all operands that follow must be **String**s (remember that the compiler automatically turns a double-quoted sequence of characters into a **String**):

```
//: operators/StringOperators.java
import static net.mindview.util.Print.*;

public class StringOperators {
  public static void main(String[] args) {
    int x = 0, y = 1, z = 2;
    String s = "x, y, z ";
    print(s + x + y + z);
    print(x + " " + s); // Converts x to a String
    s += "(summed) = "; // Concatenation operator
    print(s + (x + y + z));
    print("" + x); // Shorthand for Integer.toString()
  }
} /* Output:
x, y, z 012
0 x, y, z
x, y, z (summed) = 3
0
*///:~
```

Note that the output from the first print statement is '**o12**' instead of just '**3**', which is what you'd get if it was summing the integers. This is because the Java compiler converts **x**, **y**, and **z** into their **String** representations and concatenates those strings, instead of adding them together first. The second print statement converts the leading variable into a **String**, so the string conversion does not depend on what comes first. Finally, you see the use of the **+=** operator to append a string to **s**, and the use of parentheses to control the order of evaluation of the expression so that the **int**s are actually summed before they are displayed.

Notice the last example in **main( )**: you will sometimes see an empty **String** followed by a **+** and a primitive as a way to perform the conversion without calling the more cumbersome explicit method (**Integer.toString( )**, in this case).

Common pitfalls when using operators

One of the pitfalls when using operators is attempting to leave out the parentheses when you are even the least bit uncertain about how an expression will evaluate. This is still true in Java.

An extremely common error in C and C++ looks like this:

```
while(x = y) {
  // ....
}
```

The programmer was clearly trying to test for equivalence (**==**) rather than do an assignment. In C and C++ the result of this assignment will always be **true** if **y** is nonzero, and you'll probably get an infinite loop. In Java, the result of this expression is not a **boolean**, but the compiler expects a **boolean** and won't convert from an **int**, so it will conveniently give you a compile-time error and catch the problem before you ever try to run the program. So the pitfall never happens in Java. (The only time you won't get a compile-time error is when **x** and **y** are **boolean**, in which case **x = y** is a legal expression, and in the preceding example, probably an error.)

A similar problem in C and C++ is using bitwise AND and OR instead of the logical versions. Bitwise AND and OR use one of the characters (**&** or **|**) while logical AND and OR use two (**&&** and **||**). Just as with **=** and **==**, it's easy to type just one character instead of two. In Java, the compiler again prevents this, because it won't let you cavalierly use one type where it doesn't belong.

Casting operators

The word *cast* is used in the sense of "casting into a mold." Java will automatically change one type of data into another when appropriate. For instance, if you assign an integral value to a floating point variable, the compiler will automatically convert the **int** to a **float**. Casting allows you to make this type conversion explicit, or to force it when it wouldn't normally happen.

To perform a cast, put the desired data type inside parentheses to the left of any value. You can see this in the following example:

```
//: operators/Casting.java

public class Casting {
  public static void main(String[] args) {
    int i = 200;
    long lng = (long)i;
    lng = i; // "Widening," so cast not really required
    long lng2 = (long)200;
    lng2 = 200;
    // A "narrowing conversion":
    i = (int)lng2; // Cast required
  }
} ///:~
```

As you can see, it's possible to perform a cast on a numeric value as well as on a variable. Notice that you can introduce superfluous casts; for example, the compiler will automatically promote an **int** value to a **long** when necessary. However, you are allowed to use superfluous casts to make a point or to clarify your code. In other situations, a cast may be essential just to get the code to compile.

In C and C++, casting can cause some headaches. In Java, casting is safe, with the exception that when you perform a so-called *narrowing conversion* (that is, when you go from a data type that can hold more information to one that doesn't hold as much), you run the risk of losing information. Here the compiler forces you to use a cast, in effect saying, "This can be a dangerous thing to do—if you want me to do it anyway you must make the cast explicit." With a *widening conversion* an explicit cast is not needed, because the new type will more than hold the information from the old type so that no information is ever lost.

Java allows you to cast any primitive type to any other primitive type, except for **boolean**, which doesn't allow any casting at all. Class types do not allow casting. To convert one to the other, there must be special methods. (You'll find out later in this book that objects can be cast within a *family* of types; an **Oak** can be cast to a **Tree** and vice versa, but not to a foreign type such as a **Rock**.)

Truncation and rounding

When you are performing narrowing conversions, you must pay attention to issues of truncation and rounding. For example, if you cast from a floating point value to an integral value, what does Java do? For example, if you have the value 29.7 and you cast it to an **int**, is the resulting value 30 or 29? The answer to this can be seen in this example:

```
//: operators/CastingNumbers.java
// What happens when you cast a float
// or double to an integral value?
import static net.mindview.util.Print.*;

public class CastingNumbers {
  public static void main(String[] args) {
    double above = 0.7, below = 0.4;
    float fabove = 0.7f, fbelow = 0.4f;
    print("(int)above: " + (int)above);
    print("(int)below: " + (int)below);
    print("(int)fabove: " + (int)fabove);
    print("(int)fbelow: " + (int)fbelow);
  }
} /* Output:
(int)above: 0
(int)below: 0
(int)fabove: 0
(int)fbelow: 0
*///:~
```

So the answer is that casting from a **float** or **double** to an integral value always truncates the number. If instead you want the result to be rounded, use the **round( )** methods in **java.lang.Math**:

```
//: operators/RoundingNumbers.java
// Rounding floats and doubles.
import static net.mindview.util.Print.*;

public class RoundingNumbers {
  public static void main(String[] args) {
    double above = 0.7, below = 0.4;
    float fabove = 0.7f, fbelow = 0.4f;
    print("Math.round(above): " + Math.round(above));
    print("Math.round(below): " + Math.round(below));
```

```
    print("Math.round(fabove): " + Math.round(fabove));
    print("Math.round(fbelow): " + Math.round(fbelow));
  }
} /* Output:
Math.round(above): 1
Math.round(below): 0
Math.round(fabove): 1
Math.round(fbelow): 0
*///:~
```

Since the **round( )** is part of **java.lang**, you don't need an extra import to use it.

Promotion

You'll discover that if you perform any mathematical or bitwise operations on primitive data types that are smaller than an **int** (that is, **char**, **byte**, or **short**), those values will be promoted to **int** before performing the operations, and the resulting value will be of type **int**. So if you want to assign back into the smaller type, you must use a cast. (And, since you're assigning back into a smaller type, you might be losing information.) In general, the largest data type in an expression is the one that determines the size of the result of that expression; if you multiply a **float** and a **double**, the result will be **double**; if you add an **int** and a **long**, the result will be **long**.

Java has no "sizeof"

In C and C++, the **sizeof( )** operator tells you the number of bytes allocated for data items. The most compelling reason for **sizeof( )** in C and C++ is for portability. Different data types might be different sizes on different machines, so the programmer must discover how big those types are when performing operations that are sensitive to size. For example, one computer might store integers in 32 bits, whereas another might store integers as 16 bits. Programs could store larger values in integers on the first machine. As you might imagine, portability is a huge headache for C and C++ programmers.

Java does not need a **sizeof( )** operator for this purpose, because all the data types are the same size on all machines. You do not need to think about portability on this level—it is designed into the language.

A compendium of operators

The following example shows which primitive data types can be used with particular operators. Basically, it is the same example repeated over and over, but using different primitive data types. The file will compile without error because the lines that fail are commented out with a **//!**.

```
//: operators/AllOps.java
// Tests all the operators on all the primitive data types
// to show which ones are accepted by the Java compiler.

public class AllOps {
  // To accept the results of a boolean test:
  void f(boolean b) {}
  void boolTest(boolean x, boolean y) {
    // Arithmetic operators:
    //! x = x * y;
    //! x = x / y;
    //! x = x % y;
    //! x = x + y;
```

```
    //! x = x - y;
    //! x++;
    //! x--;
    //! x = +y;
    //! x = -y;
    // Relational and logical:
    //! f(x > y);
    //! f(x >= y);
    //! f(x < y);
    //! f(x <= y);
    f(x == y);
    f(x != y);
    f(!y);
    x = x && y;
    x = x || y;
    // Bitwise operators:
    //! x = ~y;
    x = x & y;
    x = x | y;
    x = x ^ y;
    //! x = x << 1;
    //! x = x >> 1;
    //! x = x >>> 1;
    // Compound assignment:
    //! x += y;
    //! x -= y;
    //! x *= y;
    //! x /= y;
    //! x %= y;
    //! x <<= 1;
    //! x >>= 1;
    //! x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Casting:
    //! char c = (char)x;
    //! byte b = (byte)x;
    //! short s = (short)x;
    //! int i = (int)x;
    //! long l = (long)x;
    //! float f = (float)x;
    //! double d = (double)x;
  }
  void charTest(char x, char y) {
    // Arithmetic operators:
    x = (char)(x * y);
    x = (char)(x / y);
    x = (char)(x % y);
    x = (char)(x + y);
    x = (char)(x - y);
    x++;
    x--;
    x = (char)+y;
    x = (char)-y;
    // Relational and logical:
    f(x > y);
```

```
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //! f(!x);
    //! f(x && y);
    //! f(x || y);
    // Bitwise operators:
    x= (char)~y;
    x = (char)(x & y);
    x  = (char)(x | y);
    x = (char)(x ^ y);
    x = (char)(x << 1);
    x = (char)(x >> 1);
    x = (char)(x >>> 1);
    // Compound assignment:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    x <<= 1;
    x >>= 1;
    x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Casting:
    //! boolean bl = (boolean)x;
    byte b = (byte)x;
    short s = (short)x;
    int i = (int)x;
    long l = (long)x;
    float f = (float)x;
    double d = (double)x;
  }
  void byteTest(byte x, byte y) {
    // Arithmetic operators:
    x = (byte)(x* y);
    x = (byte)(x / y);
    x = (byte)(x % y);
    x = (byte)(x + y);
    x = (byte)(x - y);
    x++;
    x--;
    x = (byte)+ y;
    x = (byte)- y;
    // Relational and logical:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //! f(!x);
    //! f(x && y);
```

```
    //! f(x || y);
    // Bitwise operators:
    x = (byte)~y;
    x = (byte)(x & y);
    x = (byte)(x | y);
    x = (byte)(x ^ y);
    x = (byte)(x << 1);
    x = (byte)(x >> 1);
    x = (byte)(x >>> 1);
    // Compound assignment:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    x <<= 1;
    x >>= 1;
    x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Casting:
    //! boolean bl = (boolean)x;
    char c = (char)x;
    short s = (short)x;
    int i = (int)x;
    long l = (long)x;
    float f = (float)x;
    double d = (double)x;
  }
  void shortTest(short x, short y) {
    // Arithmetic operators:
    x = (short)(x * y);
    x = (short)(x / y);
    x = (short)(x % y);
    x = (short)(x + y);
    x = (short)(x - y);
    x++;
    x--;
    x = (short)+y;
    x = (short)-y;
    // Relational and logical:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //! f(!x);
    //! f(x && y);
    //! f(x || y);
    // Bitwise operators:
    x = (short)~y;
    x = (short)(x & y);
    x = (short)(x | y);
    x = (short)(x ^ y);
    x = (short)(x << 1);
```

```
    x = (short)(x >> 1);
    x = (short)(x >>> 1);
    // Compound assignment:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    x <<= 1;
    x >>= 1;
    x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Casting:
    //! boolean bl = (boolean)x;
    char c = (char)x;
    byte b = (byte)x;
    int i = (int)x;
    long l = (long)x;
    float f = (float)x;
    double d = (double)x;
  }
  void intTest(int x, int y) {
    // Arithmetic operators:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
    x = -y;
    // Relational and logical:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //! f(!x);
    //! f(x && y);
    //! f(x || y);
    // Bitwise operators:
    x = ~y;
    x = x & y;
    x = x | y;
    x = x ^ y;
    x = x << 1;
    x = x >> 1;
    x = x >>> 1;
    // Compound assignment:
    x += y;
    x -= y;
    x *= y;
    x /= y;
```

```
    x %= y;
    x <<= 1;
    x >>= 1;
    x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Casting:
    //! boolean bl = (boolean)x;
    char c = (char)x;
    byte b = (byte)x;
    short s = (short)x;
    long l = (long)x;
    float f = (float)x;
    double d = (double)x;
  }
  void longTest(long x, long y) {
    // Arithmetic operators:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
    x = -y;
    // Relational and logical:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //! f(!x);
    //! f(x && y);
    //! f(x || y);
    // Bitwise operators:
    x = ~y;
    x = x & y;
    x = x | y;
    x = x ^ y;
    x = x << 1;
    x = x >> 1;
    x = x >>> 1;
    // Compound assignment:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    x <<= 1;
    x >>= 1;
    x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
```

```
    // Casting:
    //! boolean bl = (boolean)x;
    char c = (char)x;
    byte b = (byte)x;
    short s = (short)x;
    int i = (int)x;
    float f = (float)x;
    double d = (double)x;
  }
  void floatTest(float x, float y) {
    // Arithmetic operators:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
    x = -y;
    // Relational and logical:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //! f(!x);
    //! f(x && y);
    //! f(x || y);
    // Bitwise operators:
    //! x = ~y;
    //! x = x & y;
    //! x = x | y;
    //! x = x ^ y;
    //! x = x << 1;
    //! x = x >> 1;
    //! x = x >>> 1;
    // Compound assignment:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    //! x <<= 1;
    //! x >>= 1;
    //! x >>>= 1;
    //! x &= y;
    //! x ^= y;
    //! x |= y;
    // Casting:
    //! boolean bl = (boolean)x;
    char c = (char)x;
    byte b = (byte)x;
    short s = (short)x;
    int i = (int)x;
    long l = (long)x;
```

```
    double d = (double)x;
  }
  void doubleTest(double x, double y) {
    // Arithmetic operators:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
    x = -y;
    // Relational and logical:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //! f(!x);
    //! f(x && y);
    //! f(x || y);
    // Bitwise operators:
    //! x = ~y;
    //! x = x & y;
    //! x = x | y;
    //! x = x ^ y;
    //! x = x << 1;
    //! x = x >> 1;
    //! x = x >>> 1;
    // Compound assignment:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    //! x <<= 1;
    //! x >>= 1;
    //! x >>>= 1;
    //! x &= y;
    //! x ^= y;
    //! x |= y;
    // Casting:
    //! boolean bl = (boolean)x;
    char c = (char)x;
    byte b = (byte)x;
    short s = (short)x;
    int i = (int)x;
    long l = (long)x;
    float f = (float)x;
  }
} ///:~
```

Note that **boolean** is quite limited. You can assign to it the values **true** and **false**, and you can test it for truth or falsehood, but you cannot add booleans or perform any other type of operation on them.

In **char**, **byte**, and **short**, you can see the effect of promotion with the arithmetic operators. Each arithmetic operation on any of those types produces an **int** result, which must be explicitly cast back to the original type (a narrowing conversion that might lose information) to assign back to that type. With **int** values, however, you do not need to cast, because everything is already an **int**. Don't be lulled into thinking everything is safe, though. If you multiply two **int**s that are big enough, you'll overflow the result. The following example demonstrates this:

```
//: operators/Overflow.java
// Surprise! Java lets you overflow.

public class Overflow {
  public static void main(String[] args) {
    int big = Integer.MAX_VALUE;
    System.out.println("big = " + big);
    int bigger = big * 4;
    System.out.println("bigger = " + bigger);
  }
} /* Output:
big = 2147483647
bigger = -4
*///:~
```

You get no errors or warnings from the compiler, and no exceptions at run time. Java is good, but it's not *that* good.

Compound assignments do *not* require casts for **char**, **byte**, or **short**, even though they are performing promotions that have the same results as the direct arithmetic operations. On the other hand, the lack of the cast certainly simplifies the code.

You can see that, with the exception of **boolean**, any primitive type can be cast to any other primitive type. Again, you must be aware of the effect of a narrowing conversion when casting to a smaller type; otherwise, you might unknowingly lose information during the cast.

Exercise 14: (3) Write a method that takes two **String** arguments and uses all the **boolean** comparisons to compare the two **String**s and print the results. For the **==** and **!=**, also perform the **equals( )** test. In **main( )**, call your method with some different **String** objects.

Summary

If you've had experience with any languages that use C-like syntax, you can see that the operators in Java are so similar that there is virtually no learning curve. If you found this chapter challenging, make sure you view the multimedia presentation *Thinking in C*, available at *www.MindView.net*.

Solutions to selected exercises can be found in the electronic document *The Thinking in Java Annotated Solution Guide*, available for sale from *www.MindView.net*.

Controlling Execution

Like a sentient creature, a program must manipulate its world and make choices during execution. In Java you make choices with execution control statements.

Java uses all of C's execution control statements, so if you've programmed with C or C++, then most of what you see will be familiar. Most procedural programming languages have some kind of control statements, and there is often overlap among languages. In Java, the keywords include **if-else**, **while**, **do-while**, **for**, **return**, **break**, and a selection statement called **switch**. Java does not, however, support the much-maligned **goto** (which can still be the most expedient way to solve certain types of problems). You can still do a **goto**-like jump, but it is much more constrained than a typical **goto**.

true and **false**

All conditional statements use the truth or falsehood of a conditional expression to determine the execution path. An example of a conditional expression is **a == b**. This uses the conditional operator **==** to see if the value of **a** is equivalent to the value of **b**. The expression returns **true** or **false**. Any of the relational operators you've seen in the previous chapter can be used to produce a conditional statement. Note that Java doesn't allow you to use a number as a **boolean**, even though it's allowed in C and C++ (where truth is nonzero and falsehood is zero). If you want to use a non-**boolean** in a **boolean** test, such as **if(a)**, you must first convert it to a **boolean** value by using a conditional expression, such as **if(a != 0)**.

if-else

The **if-else** statement is the most basic way to control program flow. The **else** is optional, so you can use **if** in two forms:

```
if(Boolean-expression)
  statement
```

or

```
if(Boolean-expression)
  statement
else
  statement
```

The *Boolean-expression* must produce a **boolean** result. The *statement* is either a simple statement terminated by a semicolon, or a compound statement, which is a group of simple statements enclosed in braces. Whenever the word "*statement*" is used, it always implies that the statement can be simple or compound.

As an example of **if-else**, here is a **test( )** method that will tell you whether a guess is above, below, or equivalent to a target number:

```
//: control/IfElse.java
import static net.mindview.util.Print.*;

public class IfElse {
```

```
  static int result = 0;
  static void test(int testval, int target) {
    if(testval > target)
      result = +1;
    else if(testval < target)
      result = -1;
    else
      result = 0; // Match
  }
  public static void main(String[] args) {
    test(10, 5);
    print(result);
    test(5, 10);
    print(result);
    test(5, 5);
    print(result);
  }
} /* Output:
1
-1
0
*///:~
```

In the middle of **test( )**, you'll also see an "**else if**," which is not a new keyword but just an **else** followed by a new **if** statement.

Although Java, like C and C++ before it, is a "free-form" language, it is conventional to indent the body of a control flow statement so the reader can easily determine where it begins and ends.

Iteration

Looping is controlled by **while**, **do-while** and **for**, which are sometimes classified as *iteration statements*. A statement repeats until the controlling *Boolean-expression* evaluates to **false**. The form for a **while** loop is:

```
while(Boolean-expression)
  statement
```

The *Boolean-expression* is evaluated once at the beginning of the loop and again before each further iteration of the statement.

Here's a simple example that generates random numbers until a particular condition is met:

```
//: control/WhileTest.java
// Demonstrates the while loop.

public class WhileTest {
  static boolean condition() {
    boolean result = Math.random() < 0.99;
    System.out.print(result + ", ");
    return result;
  }
  public static void main(String[] args) {
    while(condition())
      System.out.println("Inside 'while'");
    System.out.println("Exited 'while'");
  }
} /* (Execute to see output) *///:~
```

The **condition( )** method uses the **static** method **random( )** in the **Math** library, which generates a **double** value between 0 and 1. (It includes 0, but not 1.) The **result** value comes from the comparison operator <, which produces a **boolean** result. If you print a **boolean** value, you automatically get the appropriate string "true" or "false." The conditional expression for the **while** says: "repeat the statements in the body as long as **condition( )** returns **true**."

do-while

The form for **do-while** is

```
do
  statement
while(Boolean-expression);
```

The sole difference between **while** and **do-while** is that the statement of the **do-while** always executes at least once, even if the expression evaluates to **false** the first time. In a **while**, if the conditional is **false** the first time the statement never executes. In practice, **do-while** is less common than **while**.

for

A **for** loop is perhaps the most commonly used form of iteration. This loop performs initialization before the first iteration. Then it performs conditional testing and, at the end of each iteration, some form of "stepping." The form of the **for** loop is:

```
for(initialization; Boolean-expression; step)
  statement
```

Any of the expressions *initialization*, *Boolean-expression* or *step* can be empty. The expression is tested before each iteration, and as soon as it evaluates to **false**, execution will continue at the line following the **for** statement. At the end of each loop, the *step* executes.

for loops are usually used for "counting" tasks:

```
//: control/ListCharacters.java
// Demonstrates "for" loop by listing
// all the lowercase ASCII letters.

public class ListCharacters {
  public static void main(String[] args) {
    for(char c = 0; c < 128; c++)
      if(Character.isLowerCase(c))
        System.out.println("value: " + (int)c +
          " character: " + c);
  }
} /* Output:
value: 97 character: a
value: 98 character: b
value: 99 character: c
value: 100 character: d
value: 101 character: e
value: 102 character: f
value: 103 character: g
value: 104 character: h
value: 105 character: i
value: 106 character: j
...
```

```
*///:~
```

Note that the variable **c** is defined at the point where it is used, inside the control expression of the **for** loop, rather than at the beginning of **main( )**. The scope of **c** is the statement controlled by the **for**.

This program also uses the **java.lang.Character** "wrapper" class, which not only wraps the primitive **char** type in an object, but also provides other utilities. Here, the **static isLowerCase( )** method is used to detect whether the character in question is a lowercase letter.

Traditional procedural languages like C require that all variables be defined at the beginning of a block so that when the compiler creates a block, it can allocate space for those variables. In Java and C++, you can spread your variable declarations throughout the block, defining them at the point that you need them. This allows a more natural coding style and makes code easier to understand.

Exercise 1: (1) Write a program that prints values from 1 to 100.

Exercise 2: (2) Write a program that generates 25 random **int** values. For each value, use an **if-else** statement to classify it as greater than, less than, or equal to a second randomly generated value.

Exercise 3: (1) Modify Exercise 2 so that your code is surrounded by an "infinite" **while** loop. It will then run until you interrupt it from the keyboard (typically by pressing Control-C).

Exercise 4: (3) Write a program that uses two nested **for** loops and the modulus operator (**%**) to detect and print prime numbers (integral numbers that are not evenly divisible by any other numbers except for themselves and 1).

Exercise 5: (4) Repeat Exercise 10 from the previous chapter, using the ternary operator and a bitwise test to display the ones and zeroes, instead of **Integer.toBinaryString( )**.

The comma operator

Earlier in this chapter I stated that the comma *operator* (not the comma *separator*, which is used to separate definitions and method arguments) has only one use in Java: in the control expression of a **for** loop. In both the initialization and step portions of the control expression, you can have a number of statements separated by commas, and those statements will be evaluated sequentially.

Using the comma operator, you can define multiple variables within a **for** statement, but they must be of the same type:

```
//: control/CommaOperator.java

public class CommaOperator {
  public static void main(String[] args) {
    for(int i = 1, j = i + 10; i < 5; i++, j = i * 2) {
      System.out.println("i = " + i + " j = " + j);
    }
  }
} /* Output:
i = 1 j = 11
i = 2 j = 4
i = 3 j = 6
i = 4 j = 8
```

```
*///:~
```

The **int** definition in the **for** statement covers both **i** and **j**. The initialization portion can have any number of definitions *of one type*. The ability to define variables in a control expression is limited to the **for** loop. You cannot use this approach with any of the other selection or iteration statements.

You can see that in both the initialization and step portions, the statements are evaluated in sequential order.

Foreach syntax

Java SE5 introduces a new and more succinct **for** syntax, for use with arrays and containers (you'll learn more about these in the *Arrays* and *Containers in Depth* chapter). This is often called the *foreach syntax*, and it means that you don't have to create an **int** to count through a sequence of items—the foreach produces each item for you, automatically.

For example, suppose you have an array of **float** and you'd like to select each element in that array:

```
//: control/ForEachFloat.java
import java.util.*;

public class ForEachFloat {
  public static void main(String[] args) {
    Random rand = new Random(47);
    float f[] = new float[10];
    for(int i = 0; i < 10; i++)
      f[i] = rand.nextFloat();
    for(float x : f)
      System.out.println(x);
  }
} /* Output:
0.72711575
0.39982635
0.5309454
0.0534122
0.16020656
0.57799757
0.18847865
0.4170137
0.51660204
0.73734957
*///:~
```

The array is populated using the old **for** loop, because it must be accessed with an index. You can see the foreach syntax in the line:

```
    for(float x : f) {
```

This defines a variable **x** of type **float** and sequentially assigns each element of **f** to **x**.

Any method that returns an array is a candidate for use with foreach. For example, the **String** class has a method **toCharArray( )** that returns an array of **char**, so you can easily iterate through the characters in a string:

```
//: control/ForEachString.java
```

```
public class ForEachString {
  public static void main(String[] args) {
    for(char c : "An African Swallow".toCharArray() )
      System.out.print(c + " ");
  }
} /* Output:
A n   A f r i c a n   S w a l l o w
*///:~
```

As you'll see in the *Holding Your Objects* chapter, foreach will also work with any object that is **Iterable**.

Many **for** statements involve stepping through a sequence of integral values, like this:

```
for(int i = 0; i < 100; i++)
```

For these, the foreach syntax won't work unless you want to create an array of **int** first. To simplify this task, I've created a method called **range( )** in **net.mindview.util.Range** that automatically generates the appropriate array. My intent is for **range( )** to be used as a **static** import:

```
//: control/ForEachInt.java
import static net.mindview.util.Range.*;
import static net.mindview.util.Print.*;

public class ForEachInt {
  public static void main(String[] args) {
    for(int i : range(10)) // 0..9
      printnb(i + " ");
    print();
    for(int i : range(5, 10)) // 5..9
      printnb(i + " ");
    print();
    for(int i : range(5, 20, 3)) // 5..20 step 3
      printnb(i + " ");
    print();
  }
} /* Output:
0 1 2 3 4 5 6 7 8 9
5 6 7 8 9
5 8 11 14 17
*///:~
```

The **range( )** method has been *overloaded*, which means the same method name can be used with different argument lists (you'll learn about overloading soon). The first overloaded form of **range( )** just starts at zero and produces values up to but not including the top end of the range. The second form starts at the first value and goes until one less than the second, and the third form has a step value so it increases by that value. **range( )** is a very simple version of what's called a *generator*, which you'll see later in the book.

Note that although **range( )** allows the use of the foreach syntax in more places, and thus arguably increases readability, it is a little less efficient, so if you are tuning for performance you may want to use a *profiler*, which is a tool that measures the performance of your code.

You'll note the use of **printnb( )** in addition to **print( )**. The **printnb( )** method does not emit a newline, so it allows you to output a line in pieces.

The foreach syntax not only saves time when typing in code. More importantly, it is far easier to read and says *what* you are trying to do (get each element of the array) rather than giving the details of *how* you are doing it ("I'm creating this index so I can use it to select each of the array elements."). The foreach syntax will be used whenever possible in this book.

return

Several keywords represent *unconditional branching*, which simply means that the branch happens without any test. These include **return**, **break**, **continue**, and a way to jump to a labeled statement which is similar to the **goto** in other languages.

The **return** keyword has two purposes: It specifies what value a method will return (if it doesn't have a **void** return value) and it causes the current method to exit, returning that value. The preceding **test( )** method can be rewritten to take advantage of this:

```
//: control/IfElse2.java
import static net.mindview.util.Print.*;

public class IfElse2 {
  static int test(int testval, int target) {
    if(testval > target)
      return +1;
    else if(testval < target)
      return -1;
    else
      return 0; // Match
  }
  public static void main(String[] args) {
    print(test(10, 5));
    print(test(5, 10));
    print(test(5, 5));
  }
} /* Output:
1
-1
0
*///:~
```

There's no need for **else**, because the method will not continue after executing a **return**.

If you do not have a **return** statement in a method that returns **void**, there's an implicit **return** at the end of that method, so it's not always necessary to include a **return** statement. However, if your method states it will return anything other than **void**, you must ensure every code path will return a value.

Exercise 6: (2) Modify the two **test( )** methods in the previous two programs so that they take two extra arguments, **begin** and **end**, and so that **testval** is tested to see if it is within the range between (and including) **begin** and **end**.

break and continue

You can also control the flow of the loop inside the body of any of the iteration statements by using **break** and **continue**. **break** quits the loop without executing the rest of the statements in the loop. **continue** stops the execution of the current iteration and goes back to the beginning of the loop to begin the next iteration.

This program shows examples of **break** and **continue** within **for** and **while** loops:

```
//: control/BreakAndContinue.java
// Demonstrates break and continue keywords.
import static net.mindview.util.Range.*;

public class BreakAndContinue {
  public static void main(String[] args) {
    for(int i = 0; i < 100; i++) {
      if(i == 74) break; // Out of for loop
      if(i % 9 != 0) continue; // Next iteration
      System.out.print(i + " ");
    }
    System.out.println();
    // Using foreach:
    for(int i : range(100)) {
      if(i == 74) break; // Out of for loop
      if(i % 9 != 0) continue; // Next iteration
      System.out.print(i + " ");
    }
    System.out.println();
    int i = 0;
    // An "infinite loop":
    while(true) {
      i++;
      int j = i * 27;
      if(j == 1269) break; // Out of loop
      if(i % 10 != 0) continue; // Top of loop
      System.out.print(i + " ");
    }
  }
} /* Output:
0 9 18 27 36 45 54 63 72
0 9 18 27 36 45 54 63 72
10 20 30 40
*///:~
```

In the **for** loop, the value of **i** never gets to 100 because the **break** statement breaks out of the loop when **i** is 74. Normally, you'd use a **break** like this only if you didn't know when the terminating condition was going to occur. The **continue** statement causes execution to go back to the top of the iteration loop (thus incrementing **i**) whenever **i** is not evenly divisible by 9. When it is, the value is printed.

The second **for** loop shows the use of foreach, and that it produces the same results.

Finally, you see an "infinite" **while** loop that would, in theory, continue forever. However, inside the loop there is a **break** statement that will break out of the loop. In addition, you'll see that the **continue** statement moves control back to the top of the loop without completing anything after that **continue** statement. (Thus printing happens in the second loop only when the value of **i** is divisible by 10.) In the output, the value 0 is printed, because 0 % 9 produces 0.

A second form of the infinite loop is **for(;;)**. The compiler treats both **while(true)** and **for(;;)** in the same way, so whichever one you use is a matter of programming taste.

Exercise 7: (1) Modify Exercise 1 so that the program exits by using the **break** keyword at value 99. Try using **return** instead.

The infamous “goto”

The **goto** keyword has been present in programming languages from the beginning. Indeed, **goto** was the genesis of program control in assembly language: “If condition A, then jump here; otherwise, jump there.” If you read the assembly code that is ultimately generated by virtually any compiler, you’ll see that program control contains many jumps (the Java compiler produces its own “assembly code,” but this code is run by the Java Virtual Machine rather than directly on a hardware CPU).

A **goto** is a jump at the source-code level, and that’s what brought it into disrepute. If a program will always jump from one point to another, isn’t there some way to reorganize the code so the flow of control is not so jumpy? **goto** fell into true disfavor with the publication of the famous “Goto considered harmful” paper by Edsger Dijkstra, and since then **goto**-bashing has been a popular sport, with advocates of the cast-out keyword scurrying for cover.

As is typical in situations like this, the middle ground is the most fruitful. The problem is not the use of **goto**, but the overuse of **goto**; in rare situations **goto** is actually the best way to structure control flow.

Although **goto** is a reserved word in Java, it is not used in the language; Java has no **goto**. However, it does have something that looks a bit like a jump tied in with the **break** and **continue** keywords. It’s not a jump but rather a way to break out of an iteration statement. The reason it’s often thrown in with discussions of **goto** is because it uses the same mechanism: a label.

A label is an identifier followed by a colon, like this:

```
label1:
```

The *only* place a label is useful in Java is right before an iteration statement. And that means *right* before—it does no good to put any other statement between the label and the iteration. And the sole reason to put a label before an iteration is if you’re going to nest another iteration or a **switch** (which you’ll learn about shortly) inside it. That’s because the **break** and **continue** keywords will normally interrupt only the current loop, but when used with a label, they’ll interrupt the loops up to where the label exists:

```
label1:
outer-iteration {
  inner-iteration {
    //...
    break; // (1)
    //...
    continue;  // (2)
    //...
    continue label1; // (3)
    //...
    break label1;  // (4)
  }
}
```

In **(1)**, the **break** breaks out of the inner iteration and you end up in the outer iteration. In **(2)**, the **continue** moves back to the beginning of the inner iteration. But in **(3)**, the **continue label1** breaks out of the inner iteration *and* the outer iteration, all the way back to **label1**. Then it does in fact continue the iteration, but starting at the outer iteration. In **(4)**, the **break label1**

also breaks all the way out to **label1**, but it does not reenter the iteration. It actually does break out of both iterations.

Here is an example using **for** loops:

```
//: control/LabeledFor.java
// For loops with "labeled break" and "labeled continue."
import static net.mindview.util.Print.*;

public class LabeledFor {
  public static void main(String[] args) {
    int i = 0;
    outer: // Can't have statements here
    for(; true ;) { // infinite loop
      inner: // Can't have statements here
      for(; i < 10; i++) {
        print("i = " + i);
        if(i == 2) {
          print("continue");
          continue;
        }
        if(i == 3) {
          print("break");
          i++; // Otherwise i never
               // gets incremented.
          break;
        }
        if(i == 7) {
          print("continue outer");
          i++; // Otherwise i never
               // gets incremented.
          continue outer;
        }
        if(i == 8) {
          print("break outer");
          break outer;
        }
        for(int k = 0; k < 5; k++) {
          if(k == 3) {
            print("continue inner");
            continue inner;
          }
        }
      }
    }
    // Can't break or continue to labels here
  }
} /* Output:
i = 0
continue inner
i = 1
continue inner
i = 2
continue
i = 3
break
i = 4
```

```
continue inner
i = 5
continue inner
i = 6
continue inner
i = 7
continue outer
i = 8
break outer
*///:~
```

Note that **break** breaks out of the **for** loop, and that the increment expression doesn't occur until the end of the pass through the **for** loop. Since **break** skips the increment expression, the increment is performed directly in the case of **i == 3**. The **continue outer** statement in the case of **i == 7** also goes to the top of the loop and also skips the increment, so it too is incremented directly.

If not for the **break outer** statement, there would be no way to get out of the outer loop from within an inner loop, since **break** by itself can break out of only the innermost loop. (The same is true for **continue**.)

Of course, in the cases where breaking out of a loop will also exit the method, you can simply use a **return**.

Here is a demonstration of labeled **break** and **continue** statements with **while** loops:

```
//: control/LabeledWhile.java
// While loops with "labeled break" and "labeled continue."
import static net.mindview.util.Print.*;

public class LabeledWhile {
  public static void main(String[] args) {
    int i = 0;
    outer:
    while(true) {
      print("Outer while loop");
      while(true) {
        i++;
        print("i = " + i);
        if(i == 1) {
          print("continue");
          continue;
        }
        if(i == 3) {
          print("continue outer");
          continue outer;
        }
        if(i == 5) {
          print("break");
          break;
        }
        if(i == 7) {
          print("break outer");
          break outer;
        }
      }
    }
```

```
  }
} /* Output:
Outer while loop
i = 1
continue
i = 2
i = 3
continue outer
Outer while loop
i = 4
i = 5
break
Outer while loop
i = 6
i = 7
break outer
*///:~
```

The same rules hold true for **while**:

1. A plain **continue** goes to the top of the innermost loop and continues.
2. A labeled **continue** goes to the label and reenters the loop right after that label.
3. A **break** "drops out of the bottom" of the loop.
4. A labeled **break** drops out of the bottom of the end of the loop denoted by the label.

It's important to remember that the *only* reason to use labels in Java is when you have nested loops and you want to **break** or **continue** through more than one nested level.

In Dijkstra's "Goto considered harmful" paper, what he specifically objected to was the labels, not the **goto**. He observed that the number of bugs seems to increase with the number of labels in a program, and that labels and **goto**s make programs difficult to analyze. Note that Java labels don't suffer from this problem, since they are constrained in their placement and can't be used to transfer control in an ad hoc manner. It's also interesting to note that this is a case where a language feature is made more useful by restricting the power of the statement.

switch

The **switch** is sometimes called a *selection statement*. The **switch** statement selects from among pieces of code based on the value of an integral expression. Its general form is:

```
switch(integral-selector) {
  case integral-value1 : statement; break;
  case integral-value2 : statement; break;
  case integral-value3 : statement; break;
  case integral-value4 : statement; break;
  case integral-value5 : statement; break;
  // ...
  default: statement;
}
```

Integral-selector is an expression that produces an integral value. The **switch** compares the result of *integral-selector* to each *integral-value*. If it finds a match, the corresponding *statement* (a single statement or multiple statements; braces are not required) executes. If no match occurs, the **default** *statement* executes.

You will notice in the preceding definition that each **case** ends with a **break**, which causes execution to jump to the end of the **switch** body. This is the conventional way to build a **switch** statement, but the **break** is optional. If it is missing, the code for the following **case** statements executes until a **break** is encountered. Although you don't usually want this kind of behavior, it can be useful to an experienced programmer. Note that the last statement, following the **default**, doesn't have a **break** because the execution just falls through to where the **break** would have taken it anyway. You could put a **break** at the end of the **default** statement with no harm if you considered it important for style's sake.

The **switch** statement is a clean way to implement multiway selection (i.e., selecting from among a number of different execution paths), but it requires a selector that evaluates to an integral value, such as **int** or **char**. If you want to use, for example, a string or a floating point number as a selector, it won't work in a **switch** statement. For non-integral types, you must use a series of **if** statements. At the end of the next chapter, you'll see that Java SE5's new **enum** feature helps ease this restriction, as **enum**s are designed to work nicely with **switch**.

Here's an example that creates letters randomly and determines whether they're vowels or consonants:

```
//: control/VowelsAndConsonants.java
// Demonstrates the switch statement.
import java.util.*;
import static net.mindview.util.Print.*;

public class VowelsAndConsonants {
  public static void main(String[] args) {
    Random rand = new Random(47);
    for(int i = 0; i < 100; i++) {
      int c = rand.nextInt(26) + 'a';
      printnb((char)c + ", " + c + ": ");
      switch(c) {
        case 'a':
        case 'e':
        case 'i':
        case 'o':
        case 'u': print("vowel");
                  break;
        case 'y':
        case 'w': print("Sometimes a vowel");
                  break;
        default:  print("consonant");
      }
    }
  }
} /* Output:
y, 121: Sometimes a vowel
n, 110: consonant
z, 122: consonant
b, 98: consonant
r, 114: consonant
n, 110: consonant
y, 121: Sometimes a vowel
g, 103: consonant
c, 99: consonant
f, 102: consonant
o, 111: vowel
```

```
w, 119: Sometimes a vowel
z, 122: consonant
...
*///:~
```

Since **Random.nextInt(26)** generates a value between 0 and 26, you need only add an offset of '**a**' to produce the lowercase letters. The single-quoted characters in the **case** statements also produce integral values that are used for comparison.

Notice how the **case**s can be "stacked" on top of each other to provide multiple matches for a particular piece of code. You should also be aware that it's essential to put the **break** statement at the end of a particular case; otherwise, control will simply drop through and continue processing on the next case.

In the statement:

```
int c = rand.nextInt(26) + 'a';
```

Random.nextInt( ) produces a random **int** value from 0 to 25, which is added to the value of **'a'**. This means that **'a'** is automatically converted to an **int** to perform the addition.

In order to print **c** as a character, it must be cast to **char**; otherwise, you'll produce integral output.

Exercise 8: (2) Create a **switch** statement that prints a message for each **case**, and put the **switch** inside a **for** loop that tries each **case**. Put a **break** after each **case** and test it, then remove the **break**s and see what happens.

Exercise 9: (4) A *Fibonacci sequence* is the sequence of numbers 1, 1, 2, 3, 5, 8, 13, 21, 34, and so on, where each number (from the third on) is the sum of the previous two. Create a method that takes an integer as an argument and displays that many Fibonacci numbers starting from the beginning, e.g., If you run **java Fibonacci 5** (where **Fibonacci** is the name of the class) the output will be: 1, 1, 2, 3, 5.

Exercise 10: (5) A *vampire number* has an even number of digits and is formed by multiplying a pair of numbers containing half the number of digits of the result. The digits are taken from the original number in any order. Pairs of trailing zeroes are not allowed. Examples include:
1260 = 21 * 60
1827 = 21 * 87
2187 = 27 * 81
Write a program that finds all the 4-digit vampire numbers. (Suggested by Dan Forhan.)

Summary

This chapter concludes the study of fundamental features that appear in most programming languages: calculation, operator precedence, type casting, and selection and iteration. Now you're ready to begin taking steps that move you closer to the world of object-oriented programming. The next chapter will cover the important issues of initialization and cleanup of objects, followed in the subsequent chapter by the essential concept of implementation hiding.

Solutions to selected exercises can be found in the electronic document *The Thinking in Java Annotated Solution Guide*, available for sale from *www.MindView.net*.

Initialization & Cleanup

As the computer revolution progresses, "unsafe" programming has become one of the major culprits that makes programming expensive.

Two of these safety issues are *initialization* and *cleanup*. Many C bugs occur when the programmer forgets to initialize a variable. This is especially true with libraries when users don't know how to initialize a library component, or even that they must. Cleanup is a special problem because it's easy to forget about an element when you're done with it, since it no longer concerns you. Thus, the resources used by that element are retained and you can easily end up running out of resources (most notably, memory).

C++ introduced the concept of a *constructor*, a special method automatically called when an object is created. Java also adopted the constructor, and in addition has a garbage collector that automatically releases memory resources when they're no longer being used. This chapter examines the issues of initialization and cleanup, and their support in Java.

Guaranteed initialization with the constructor

You can imagine creating a method called **initialize( )** for every class you write. The name is a hint that it should be called before using the object. Unfortunately, this means the user must remember to call that method. In Java, the class designer can guarantee initialization of every object by providing a constructor. If a class has a constructor, Java automatically calls that constructor when an object is created, before users can even get their hands on it. So initialization is guaranteed.

The next challenge is what to name this method. There are two issues. The first is that any name you use could clash with a name you might like to use as a member in the class. The second is that because the compiler is responsible for calling the constructor, it must always know which method to call. The C++ solution seems the easiest and most logical, so it's also used in Java: The name of the constructor is the same as the name of the class. It makes sense that such a method will be called automatically during initialization.

Here's a simple class with a constructor:

```
//: initialization/SimpleConstructor.java
// Demonstration of a simple constructor.

class Rock {
  Rock() { // This is the constructor
    System.out.print("Rock ");
  }
}

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

```
    for(int i = 0; i < 10; i++)
      new Rock();
  }
} /* Output:
Rock Rock Rock Rock Rock Rock Rock Rock Rock Rock
*///:~
```

Now, when an object is created:

```
new Rock();
```

storage is allocated and the constructor is called. It is guaranteed that the object will be properly initialized before you can get your hands on it.

Note that the coding style of making the first letter of all methods lowercase does not apply to constructors, since the name of the constructor must match the name of the class *exactly*.

A constructor that takes no arguments is called the *default constructor*. The Java documents typically use the term *no-arg* constructor, but "default constructor" has been in use for many years before Java appeared, so I will tend to use that. But like any method, the constructor can also have arguments to allow you to specify *how* an object is created. The preceding example can easily be changed so the constructor takes an argument:

```
//: initialization/SimpleConstructor2.java
// Constructors can have arguments.

class Rock2 {
  Rock2(int i) {
    System.out.print("Rock " + i + " ");
  }
}

public class SimpleConstructor2 {
  public static void main(String[] args) {
    for(int i = 0; i < 8; i++)
      new Rock2(i);
  }
} /* Output:
Rock 0 Rock 1 Rock 2 Rock 3 Rock 4 Rock 5 Rock 6 Rock 7
*///:~
```

Constructor arguments provide you with a way to provide parameters for the initialization of an object. For example, if the class **Tree** has a constructor that takes a single integer argument denoting the height of the tree, you create a **Tree** object like this:

```
Tree t = new Tree(12);  // 12-foot tree
```

If **Tree(int)** is your only constructor, then the compiler won't let you create a **Tree** object any other way.

Constructors eliminate a large class of problems and make the code easier to read. In the preceding code fragment, for example, you don't see an explicit call to some **initialize( )** method that is conceptually separate from creation. In Java, creation and initialization are unified concepts—you can't have one without the other.

The constructor is an unusual type of method because it has no return value. This is distinctly different from a **void** return value, in which the method returns nothing but you still have the option to make it return something else. Constructors return nothing and you don't have an

option (the **new** expression does return a reference to the newly created object, but the constructor itself has no return value). If there were a return value, and if you could select your own, the compiler would somehow need to know what to do with that return value.

Exercise 1: (1) Create a class containing an uninitialized **String** reference. Demonstrate that this reference is initialized by Java to **null**.

Exercise 2: (2) Create a class with a **String** field that is initialized at the point of definition, and another one that is initialized by the constructor. What is the difference between the two approaches?

Method overloading

One of the important features in any programming language is the use of names. When you create an object, you give a name to a region of storage. A method is a name for an action. You refer to all objects and methods by using names. Well-chosen names create a system that is easier for people to understand and change. It's a lot like writing prose—the goal is to communicate with your readers.

A problem arises when mapping the concept of nuance in human language onto a programming language. Often, the same word expresses a number of different meanings—it's *overloaded*. This is useful, especially when it comes to trivial differences. You say, "Wash the shirt," "Wash the car," and "Wash the dog." It would be silly to be forced to say, "shirtWash the shirt," "carWash the car," and "dogWash the dog" just so the listener doesn't need to make any distinction about the action performed. Most human languages are redundant, so even if you miss a few words, you can still determine the meaning. You don't need unique identifiers—you can deduce meaning from context.

Most programming languages (C in particular) require you to have a unique identifier for each method (often called *functions* in those languages). So you could *not* have one function called **print( )** for printing integers and another called **print( )** for printing floats—each function requires a unique name.

In Java (and C++), another factor forces the overloading of method names: the constructor. Because the constructor's name is predetermined by the name of the class, there can be only one constructor name. But what if you want to create an object in more than one way? For example, suppose you build a class that can initialize itself in a standard way or by reading information from a file. You need two constructors, the default constructor and one that takes a **String** as an argument, which is the name of the file from which to initialize the object. Both are constructors, so they must have the same name—the name of the class. Thus, *method overloading* is essential to allow the same method name to be used with different argument types. And although method overloading is a must for constructors, it's a general convenience and can be used with any method.

Here's an example that shows both overloaded constructors and overloaded methods:

```
//: initialization/Overloading.java
// Demonstration of both constructor
// and ordinary method overloading.
import static net.mindview.util.Print.*;

class Tree {
  int height;
  Tree() {
    print("Planting a seedling");
```

```
    height = 0;
  }
  Tree(int initialHeight) {
    height = initialHeight;
    print("Creating new Tree that is " +
      height + " feet tall");
  }
  void info() {
    print("Tree is " + height + " feet tall");
  }
  void info(String s) {
    print(s + ": Tree is " + height + " feet tall");
  }
}

public class Overloading {
  public static void main(String[] args) {
    for(int i = 0; i < 5; i++) {
      Tree t = new Tree(i);
      t.info();
      t.info("overloaded method");
    }
    // Overloaded constructor:
    new Tree();
  }
} /* Output:
Creating new Tree that is 0 feet tall
Tree is 0 feet tall
overloaded method: Tree is 0 feet tall
Creating new Tree that is 1 feet tall
Tree is 1 feet tall
overloaded method: Tree is 1 feet tall
Creating new Tree that is 2 feet tall
Tree is 2 feet tall
overloaded method: Tree is 2 feet tall
Creating new Tree that is 3 feet tall
Tree is 3 feet tall
overloaded method: Tree is 3 feet tall
Creating new Tree that is 4 feet tall
Tree is 4 feet tall
overloaded method: Tree is 4 feet tall
Planting a seedling
*///:~
```

A **Tree** object can be created either as a seedling, with no argument, or as a plant grown in a nursery, with an existing height. To support this, there is a default constructor, and one that takes the existing height.

You might also want to call the **info( )** method in more than one way. For example, if you have an extra message you want printed, you can use **info(String)**, and **info( )** if you have nothing more to say. It would seem strange to give two separate names to what is obviously the same concept. Fortunately, method overloading allows you to use the same name for both.

Distinguishing overloaded methods

If the methods have the same name, how can Java know which method you mean? There's a simple rule: Each overloaded method must take a unique list of argument types.

If you think about this for a second, it makes sense. How else could a programmer tell the difference between two methods that have the same name, other than by the types of their arguments?

Even differences in the ordering of arguments are sufficient to distinguish two methods, although you don't normally want to take this approach because it produces difficult-to-maintain code:

```
//: initialization/OverloadingOrder.java
// Overloading based on the order of the arguments.
import static net.mindview.util.Print.*;

public class OverloadingOrder {
  static void f(String s, int i) {
    print("String: " + s + ", int: " + i);
  }
  static void f(int i, String s) {
    print("int: " + i + ", String: " + s);
  }
  public static void main(String[] args) {
    f("String first", 11);
    f(99, "Int first");
  }
} /* Output:
String: String first, int: 11
int: 99, String: Int first
*///:~
```

The two **f( )** methods have identical arguments, but the order is different, and that's what makes them distinct.

Overloading with primitives

A primitive can be automatically promoted from a smaller type to a larger one, and this can be slightly confusing in combination with overloading. The following example demonstrates what happens when a primitive is handed to an overloaded method:

```
//: initialization/PrimitiveOverloading.java
// Promotion of primitives and overloading.
import static net.mindview.util.Print.*;

public class PrimitiveOverloading {
  void f1(char x) { printnb("f1(char) "); }
  void f1(byte x) { printnb("f1(byte) "); }
  void f1(short x) { printnb("f1(short) "); }
  void f1(int x) { printnb("f1(int) "); }
  void f1(long x) { printnb("f1(long) "); }
  void f1(float x) { printnb("f1(float) "); }
  void f1(double x) { printnb("f1(double) "); }

  void f2(byte x) { printnb("f2(byte) "); }
  void f2(short x) { printnb("f2(short) "); }
  void f2(int x) { printnb("f2(int) "); }
  void f2(long x) { printnb("f2(long) "); }
  void f2(float x) { printnb("f2(float) "); }
  void f2(double x) { printnb("f2(double) "); }
```

```
  void f3(short x) { printnb("f3(short) "); }
  void f3(int x) { printnb("f3(int) "); }
  void f3(long x) { printnb("f3(long) "); }
  void f3(float x) { printnb("f3(float) "); }
  void f3(double x) { printnb("f3(double) "); }

  void f4(int x) { printnb("f4(int) "); }
  void f4(long x) { printnb("f4(long) "); }
  void f4(float x) { printnb("f4(float) "); }
  void f4(double x) { printnb("f4(double) "); }

  void f5(long x) { printnb("f5(long) "); }
  void f5(float x) { printnb("f5(float) "); }
  void f5(double x) { printnb("f5(double) "); }

  void f6(float x) { printnb("f6(float) "); }
  void f6(double x) { printnb("f6(double) "); }

  void f7(double x) { printnb("f7(double) "); }

  void testConstVal() {
    printnb("5: ");
    f1(5);f2(5);f3(5);f4(5);f5(5);f6(5);f7(5); print();
  }
  void testChar() {
    char x = 'x';
    printnb("char: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x); print();
  }
  void testByte() {
    byte x = 0;
    printnb("byte: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x); print();
  }
  void testShort() {
    short x = 0;
    printnb("short: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x); print();
  }
  void testInt() {
    int x = 0;
    printnb("int: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x); print();
  }
  void testLong() {
    long x = 0;
    printnb("long: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x); print();
  }
  void testFloat() {
    float x = 0;
    printnb("float: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x); print();
  }
  void testDouble() {
    double x = 0;
    printnb("double: ");
```

```
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x); print();
  }
  public static void main(String[] args) {
    PrimitiveOverloading p =
      new PrimitiveOverloading();
    p.testConstVal();
    p.testChar();
    p.testByte();
    p.testShort();
    p.testInt();
    p.testLong();
    p.testFloat();
    p.testDouble();
  }
} /* Output:
5: f1(int) f2(int) f3(int) f4(int) f5(long) f6(float) f7(double)
char: f1(char) f2(int) f3(int) f4(int) f5(long) f6(float) f7(double)
byte: f1(byte) f2(byte) f3(short) f4(int) f5(long) f6(float) f7(double)
short: f1(short) f2(short) f3(short) f4(int) f5(long) f6(float) f7(double)
int: f1(int) f2(int) f3(int) f4(int) f5(long) f6(float) f7(double)
long: f1(long) f2(long) f3(long) f4(long) f5(long) f6(float) f7(double)
float: f1(float) f2(float) f3(float) f4(float) f5(float) f6(float)
f7(double)
double: f1(double) f2(double) f3(double) f4(double) f5(double) f6(double)
f7(double)
*///:~
```

You can see that the constant value 5 is treated as an **int**, so if an overloaded method is available that takes an **int**, it is used. In all other cases, if you have a data type that is smaller than the argument in the method, that data type is promoted. **char** produces a slightly different effect, since if it doesn't find an exact **char** match, it is promoted to **int**.

What happens if your argument is *bigger* than the argument expected by the overloaded method? A modification of the preceding program gives the answer:

```
//: initialization/Demotion.java
// Demotion of primitives and overloading.
import static net.mindview.util.Print.*;

public class Demotion {
  void f1(char x) { print("f1(char)"); }
  void f1(byte x) { print("f1(byte)"); }
  void f1(short x) { print("f1(short)"); }
  void f1(int x) { print("f1(int)"); }
  void f1(long x) { print("f1(long)"); }
  void f1(float x) { print("f1(float)"); }
  void f1(double x) { print("f1(double)"); }

  void f2(char x) { print("f2(char)"); }
  void f2(byte x) { print("f2(byte)"); }
  void f2(short x) { print("f2(short)"); }
  void f2(int x) { print("f2(int)"); }
  void f2(long x) { print("f2(long)"); }
  void f2(float x) { print("f2(float)"); }

  void f3(char x) { print("f3(char)"); }
  void f3(byte x) { print("f3(byte)"); }
```

```
  void f3(short x) { print("f3(short)"); }
  void f3(int x) { print("f3(int)"); }
  void f3(long x) { print("f3(long)"); }

  void f4(char x) { print("f4(char)"); }
  void f4(byte x) { print("f4(byte)"); }
  void f4(short x) { print("f4(short)"); }
  void f4(int x) { print("f4(int)"); }

  void f5(char x) { print("f5(char)"); }
  void f5(byte x) { print("f5(byte)"); }
  void f5(short x) { print("f5(short)"); }

  void f6(char x) { print("f6(char)"); }
  void f6(byte x) { print("f6(byte)"); }

  void f7(char x) { print("f7(char)"); }

  void testDouble() {
    double x = 0;
    print("double argument:");
    f1(x);f2((float)x);f3((long)x);f4((int)x);
    f5((short)x);f6((byte)x);f7((char)x);
  }
  public static void main(String[] args) {
    Demotion p = new Demotion();
    p.testDouble();
  }
} /* Output:
double argument:
f1(double)
f2(float)
f3(long)
f4(int)
f5(short)
f6(byte)
f7(char)
*///:~
```

Here, the methods take narrower primitive values. If your argument is wider, then you must perform a narrowing conversion with a cast. If you don't do this, the compiler will issue an error message.

Overloading on return values

It is common to wonder, "Why only class names and method argument lists? Why not distinguish between methods based on their return values?" For example, these two methods, which have the same name and arguments, are easily distinguished from each other:

```
void f() {}
int f() { return 1; }
```

This might work fine as long as the compiler could unequivocally determine the meaning from the context, as in **int x = f( )**. However, you can also call a method and ignore the return value. This is often referred to as *calling a method for its side effect*, since you don't care about the return value, but instead want the other effects of the method call. So if you call the method this way:

```
f();
```

how can Java determine which **f( )** should be called? And how could someone reading the code see it? Because of this sort of problem, you cannot use return value types to distinguish overloaded methods.

Default constructors

As mentioned previously, a default constructor (a.k.a. a "no-arg" constructor) is one without arguments that is used to create a "default object." If you create a class that has no constructors, the compiler will automatically create a default constructor for you. For example:

```
//: initialization/DefaultConstructor.java

class Bird {}

public class DefaultConstructor {
  public static void main(String[] args) {
    Bird b = new Bird(); // Default!
  }
} ///:~
```

The expression

```
new Bird()
```

creates a new object and calls the default constructor, even though one was not explicitly defined. Without it, you would have no method to call to build the object. However, if you define any constructors (with or without arguments), the compiler will *not* synthesize one for you:

```
//: initialization/NoSynthesis.java

class Bird2 {
  Bird2(int i) {}
  Bird2(double d) {}
}

public class NoSynthesis {
  public static void main(String[] args) {
    //! Bird2 b = new Bird2(); // No default
    Bird2 b2 = new Bird2(1);
    Bird2 b3 = new Bird2(1.0);
  }
} ///:~
```

If you say:

```
new Bird2()
```

the compiler will complain that it cannot find a constructor that matches. When you don't put in any constructors, it's as if the compiler says, "You are bound to need *some* constructor, so let me make one for you." But if you write a constructor, the compiler says, "You've written a constructor so you know what you're doing; if you didn't put in a default it's because you meant to leave it out."

Exercise 3: (1) Create a class with a default constructor (one that takes no arguments) that prints a message. Create an object of this class.

Exercise 4: (1) Add an overloaded constructor to the previous exercise that takes a **String** argument and prints it along with your message.

Exercise 5: (2) Create a class called **Dog** with an overloaded **bark( )** method. This method should be overloaded based on various primitive data types, and print different types of barking, howling, etc., depending on which overloaded version is called. Write a **main( )** that calls all the different versions.

Exercise 6: (1) Modify the previous exercise so that two of the overloaded methods have two arguments (of two different types), but in reversed order relative to each other. Verify that this works.

Exercise 7: (1) Create a class without a constructor, and then create an object of that class in **main( )** to verify that the default constructor is automatically synthesized.

The **this** keyword

If you have two objects of the same type called **a** and **b**, you might wonder how it is that you can call a method **peel( )** for both those objects:

```
//: initialization/BananaPeel.java

class Banana { void peel(int i) { /* ... */ } }

public class BananaPeel {
  public static void main(String[] args) {
    Banana a = new Banana(),
           b = new Banana();
    a.peel(1);
    b.peel(2);
  }
} ///:~
```

If there's only one method called **peel( )**, how can that method know whether it's being called for the object **a** or **b**?

To allow you to write the code in a convenient object-oriented syntax in which you "send a message to an object," the compiler does some undercover work for you. There's a secret first argument passed to the method **peel( )**, and that argument is the reference to the object that's being manipulated. So the two method calls become something like:

```
Banana.peel(a, 1);
Banana.peel(b, 2);
```

This is internal and you can't write these expressions and get the compiler to accept them, but it gives you an idea of what's happening.

Suppose you're inside a method and you'd like to get the reference to the current object. Since that reference is passed *secretly* by the compiler, there's no identifier for it. However, for this purpose there's a keyword: **this**. The **this** keyword—which can be used only inside a non-**static** method—produces the reference to the object that the method has been called for. You can treat the reference just like any other object reference. Keep in mind that if you're calling a method of your class from within another method of your class, you don't need to use **this**. You simply call the method. The current **this** reference is automatically used for the other method. Thus you can say:

```
//: initialization/Apricot.java
```

```
public class Apricot {
  void pick() { /* ... */ }
  void pit() { pick(); /* ... */ }
} ///:~
```

Inside **pit( )**, you *could* say **this.pick( )** but there's no need to.[1] The compiler does it for you automatically. The **this** keyword is used only for those special cases in which you need to explicitly use the reference to the current object. For example, it's often used in **return** statements when you want to return the reference to the current object:

```
//: initialization/Leaf.java
// Simple use of the "this" keyword.

public class Leaf {
  int i = 0;
  Leaf increment() {
    i++;
    return this;
  }
  void print() {
    System.out.println("i = " + i);
  }
  public static void main(String[] args) {
    Leaf x = new Leaf();
    x.increment().increment().increment().print();
  }
} /* Output:
i = 3
*///:~
```

Because **increment( )** returns the reference to the current object via the **this** keyword, multiple operations can easily be performed on the same object.

The **this** keyword is also useful for passing the current object to another method:

```
//: initialization/PassingThis.java

class Person {
  public void eat(Apple apple) {
    Apple peeled = apple.getPeeled();
    System.out.println("Yummy");
  }
}

class Peeler {
  static Apple peel(Apple apple) {
    // ... remove peel
    return apple; // Peeled
  }
}
```

[1] Some people will obsessively put **this** in front of every method call and field reference, arguing that it makes it "clearer and more explicit." Don't do it. There's a reason that we use high-level languages: They do things for us. If you put **this** in when it's not necessary, you will confuse and annoy everyone who reads your code, since all the rest of the code they've read *won't* use **this** everywhere. People expect **this** to be used only when it is necessary. Following a consistent and straightforward coding style saves time and money.

```
class Apple {
  Apple getPeeled() { return Peeler.peel(this); }
}

public class PassingThis {
  public static void main(String[] args) {
    new Person().eat(new Apple());
  }
} /* Output:
Yummy
*///:~
```

Apple needs to call **Peeler.peel( )**, which is a foreign utility method that performs an operation that, for some reason, needs to be external to **Apple** (perhaps the external method can be applied across many different classes, and you don't want to repeat the code). To pass itself to the foreign method, it must use **this**.

Exercise 8: (1) Create a class with two methods. Within the first method, call the second method twice: the first time without using **this**, and the second time using **this**—just to see it working; you should not use this form in practice.

Calling constructors from constructors

When you write several constructors for a class, there are times when you'd like to call one constructor from another to avoid duplicating code. You can make such a call by using the **this** keyword.

Normally, when you say **this**, it is in the sense of "this object" or "the current object," and by itself it produces the reference to the current object. In a constructor, the **this** keyword takes on a different meaning when you give it an argument list. It makes an explicit call to the constructor that matches that argument list. Thus you have a straightforward way to call other constructors:

```
//: initialization/Flower.java
// Calling constructors with "this"
import static net.mindview.util.Print.*;

public class Flower {
  int petalCount = 0;
  String s = "initial value";
  Flower(int petals) {
    petalCount = petals;
    print("Constructor w/ int arg only, petalCount= "
      + petalCount);
  }
  Flower(String ss) {
    print("Constructor w/ String arg only, s = " + ss);
    s = ss;
  }
  Flower(String s, int petals) {
    this(petals);
//!    this(s); // Can't call two!
    this.s = s; // Another use of "this"
    print("String & int args");
  }
  Flower() {
    this("hi", 47);
    print("default constructor (no args)");
```

```
  }
  void printPetalCount() {
//! this(11); // Not inside non-constructor!
    print("petalCount = " + petalCount + " s = "+ s);
  }
  public static void main(String[] args) {
    Flower x = new Flower();
    x.printPetalCount();
  }
} /* Output:
Constructor w/ int arg only, petalCount= 47
String & int args
default constructor (no args)
petalCount = 47 s = hi
*///:~
```

The constructor **Flower(String s, int petals)** shows that, while you can call one constructor using **this**, you cannot call two. In addition, the constructor call must be the first thing you do, or you'll get a compiler error message.

This example also shows another way you'll see **this** used. Since the name of the argument **s** and the name of the member data **s** are the same, there's an ambiguity. You can resolve it using **this.s**, to say that you're referring to the member data. You'll often see this form used in Java code, and it's used in numerous places in this book.

In **printPetalCount( )** you can see that the compiler won't let you call a constructor from inside any method other than a constructor.

Exercise 9: (1) Create a class with two (overloaded) constructors. Using **this**, call the second constructor inside the first one.

The meaning of **static**

With the **this** keyword in mind, you can more fully understand what it means to make a method **static**. It means that there is no **this** for that particular method. You cannot call non-**static** methods from inside **static** methods[2] (although the reverse is possible), and you can call a **static** method for the class itself, without any object. In fact, that's primarily what a **static** method is for. It's as if you're creating the equivalent of a global method. However, global methods are not permitted in Java, and putting the **static** method inside a class allows it access to other **static** methods and to **static** fields.

Some people argue that **static** methods are not object-oriented, since they do have the semantics of a global method; with a **static** method, you don't send a message to an object, since there's no **this**. This is probably a fair argument, and if you find yourself using a *lot* of **static** methods, you should probably rethink your strategy. However, **static**s are pragmatic, and there are times when you genuinely need them, so whether or not they are "proper OOP" should be left to the theoreticians.

[2] The one case in which this is possible occurs if you pass a reference to an object into the **static** method (the **static** method could also create its own object). Then, via the reference (which is now effectively **this**), you can call non-**static** methods and access non-**static** fields. But typically, if you want to do something like this, you'll just make an ordinary, non-**static** method.

Cleanup: finalization and garbage collection

Programmers know about the importance of initialization, but often forget the importance of cleanup. After all, who needs to clean up an **int**? But with libraries, simply "letting go" of an object once you're done with it is not always safe. Of course, Java has the garbage collector to reclaim the memory of objects that are no longer used. Now consider an unusual case: Suppose your object allocates "special" memory without using **new**. The garbage collector only knows how to release memory allocated *with* **new**, so it won't know how to release the object's "special" memory. To handle this case, Java provides a method called **finalize( )** that you can define for your class. Here's how it's *supposed* to work. When the garbage collector is ready to release the storage used for your object, it will first call **finalize( )**, and only on the next garbage-collection pass will it reclaim the object's memory. So if you choose to use **finalize( )**, it gives you the ability to perform some important cleanup *at the time of garbage collection*.

This is a potential programming pitfall because some programmers, especially C++ programmers, might initially mistake **finalize( )** for the *destructor* in C++, which is a function that is *always* called when an object is destroyed. It is important to distinguish between C++ and Java here, because in C++, *objects always get destroyed* (in a bug-free program), whereas in Java, objects do not always get garbage collected. Or, put another way:

1. *Your objects might not get garbage collected.*
2. *Garbage collection is not destruction.*

If you remember this, you will stay out of trouble. What it means is that if there is some activity that must be performed before you no longer need an object, you must perform that activity yourself. Java has no destructor or similar concept, so you must create an ordinary method to perform this cleanup. For example, suppose that in the process of creating your object, it draws itself on the screen. If you don't explicitly erase its image from the screen, it might never get cleaned up. If you put some kind of erasing functionality inside **finalize( )**, then if an object is garbage collected and **finalize( )** is called (and there's no guarantee this will happen), then the image will first be removed from the screen, but if it isn't, the image will remain.

You might find that the storage for an object never gets released because your program never nears the point of running out of storage. If your program completes and the garbage collector never gets around to releasing the storage for any of your objects, that storage will be returned to the operating system *en masse* as the program exits. This is a good thing, because garbage collection has some overhead, and if you never do it, you never incur that expense.

What is **finalize()** for?

So, if you should not use **finalize( )** as a general-purpose cleanup method, what good is it?

A third point to remember is:

3. *Garbage collection is only about memory.*

That is, the sole reason for the existence of the garbage collector is to recover memory that your program is no longer using. So any activity that is associated with garbage collection, most notably your **finalize( )** method, must also be only about memory and its deallocation.

Does this mean that if your object contains other objects, **finalize( )** should explicitly release those objects? Well, no—the garbage collector takes care of the release of all object memory regardless of how the object is created. It turns out that the need for **finalize( )** is limited to special cases in which your object can allocate storage in some way other than creating an object. But, you might observe, everything in Java is an object, so how can this be?

It would seem that **finalize( )** is in place because of the possibility that you'll do something C-like by allocating memory using a mechanism other than the normal one in Java. This can happen primarily through *native methods*, which are a way to call non-Java code from Java. (Native methods are covered in Appendix B in the electronic 2nd edition of this book, available at *www.MindView.net*.) C and C++ are the only languages currently supported by native methods, but since they can call subprograms in other languages, you can effectively call anything. Inside the non-Java code, C's **malloc( )** family of functions might be called to allocate storage, and unless you call **free( )**, that storage will not be released, causing a memory leak. Of course, **free( )** is a C and C++ function, so you'd need to call it in a native method inside your **finalize( )**.

After reading this, you probably get the idea that you won't use **finalize( )** much.[3] You're correct; it is not the appropriate place for normal cleanup to occur. So where should normal cleanup be performed?

You must perform cleanup

To clean up an object, the user of that object must call a cleanup method at the point the cleanup is desired. This sounds pretty straightforward, but it collides a bit with the C++ concept of the destructor. In C++, all objects are destroyed. Or rather, all objects *should be* destroyed. If the C++ object is created as a local (i.e., on the stack—not possible in Java), then the destruction happens at the closing curly brace of the scope in which the object was created. If the object was created using **new** (like in Java), the destructor is called when the programmer calls the C++ operator **delete** (which doesn't exist in Java). If the C++ programmer forgets to call **delete**, the destructor is never called, and you have a memory leak, plus the other parts of the object never get cleaned up. This kind of bug can be very difficult to track down, and is one of the compelling reasons to move from C++ to Java.

In contrast, Java doesn't allow you to create local objects—you must always use **new**. But in Java, there's no "delete" for releasing the object, because the garbage collector releases the storage for you. So from a simplistic standpoint, you could say that because of garbage collection, Java has no destructor. You'll see as this book progresses, however, that the presence of a garbage collector does not remove the need for or the utility of destructors. (And you should never call **finalize( )** directly, so that's not a solution.) If you want some kind of cleanup performed other than storage release, you must *still* explicitly call an appropriate method in Java, which is the equivalent of a C++ destructor without the convenience.

Remember that neither garbage collection nor finalization is guaranteed. If the JVM isn't close to running out of memory, then it might not waste time recovering memory through garbage collection.

---

[3] Joshua Bloch goes further in his section titled "avoid finalizers": "Finalizers are unpredictable, often dangerous, and generally unnecessary." *Effective Java™ Programming Language Guide,* p. 20 (Addison-Wesley, 2001).

The termination condition

In general, you can't rely on **finalize( )** being called, and you must create separate "cleanup" methods and call them explicitly. So it appears that **finalize( )** is only useful for obscure memory cleanup that most programmers will never use. However, there is an interesting use of **finalize( )** that does not rely on it being called every time. This is the verification of the *termination condition*[4] of an object.

At the point that you're no longer interested in an object—when it's ready to be cleaned up—that object should be in a state whereby its memory can be safely released. For example, if the object represents an open file, that file should be closed by the programmer before the object is garbage collected. If any portions of the object are not properly cleaned up, then you have a bug in your program that can be very difficult to find. **finalize( )** can be used to eventually discover this condition, even if it isn't always called. If one of the finalizations happens to reveal the bug, then you discover the problem, which is all you really care about.

Here's a simple example of how you might use it:

```
//: initialization/TerminationCondition.java
// Using finalize() to detect an object that
// hasn't been properly cleaned up.

class Book {
  boolean checkedOut = false;
  Book(boolean checkOut) {
    checkedOut = checkOut;
  }
  void checkIn() {
    checkedOut = false;
  }
  protected void finalize() {
    if(checkedOut)
      System.out.println("Error: checked out");
    // Normally, you'll also do this:
    // super.finalize(); // Call the base-class version
  }
}

public class TerminationCondition {
  public static void main(String[] args) {
    Book novel = new Book(true);
    // Proper cleanup:
    novel.checkIn();
    // Drop the reference, forget to clean up:
    new Book(true);
    // Force garbage collection & finalization:
    System.gc();
  }
} /* Output:
Error: checked out
*///:~
```

[4] A term coined by Bill Venners (*www.Artima.com*) during a seminar that he and I were giving together.

The termination condition is that all **Book** objects are supposed to be checked in before they are garbage collected, but in **main( )**, a programmer error doesn't check in one of the books. Without **finalize( )** to verify the termination condition, this can be a difficult bug to find.

Note that **System.gc( )** is used to force finalization. But even if it isn't, it's highly probable that the errant **Book** will eventually be discovered through repeated executions of the program (assuming the program allocates enough storage to cause the garbage collector to execute).

You should generally assume that the base-class version of **finalize( )** will also be doing something important, and call it using **super**, as you can see in **Book.finalize( )**. In this case, it is commented out because it requires exception handling, which we haven't covered yet.

Exercise 10: (2) Create a class with a **finalize( )** method that prints a message. In **main( )**, create an object of your class. Explain the behavior of your program.

Exercise 11: (4) Modify the previous exercise so that your **finalize( )** will always be called.

Exercise 12: (4) Create a class called **Tank** that can be filled and emptied, and has a *termination condition* that it must be empty when the object is cleaned up. Write a **finalize( )** that verifies this termination condition. In **main( )**, test the possible scenarios that can occur when your **Tank** is used.

How a garbage collector works

If you come from a programming language where allocating objects on the heap is expensive, you may naturally assume that Java's scheme of allocating everything (except primitives) on the heap is also expensive. However, it turns out that the garbage collector can have a significant impact on *increasing* the speed of object creation. This might sound a bit odd at first—that storage release affects storage allocation—but it's the way some JVMs work, and it means that allocating storage for heap objects in Java can be nearly as fast as creating storage *on the stack* in other languages.

For example, you can think of the C++ heap as a yard where each object stakes out its own piece of turf. This real estate can become abandoned sometime later and must be reused. In some JVMs, the Java heap is quite different; it's more like a conveyor belt that moves forward every time you allocate a new object. This means that object storage allocation is remarkably rapid. The "heap pointer" is simply moved forward into virgin territory, so it's effectively the same as C++'s stack allocation. (Of course, there's a little extra overhead for bookkeeping, but it's nothing like searching for storage.)

You might observe that the heap isn't in fact a conveyor belt, and if you treat it that way, you'll start paging memory—moving it on and off disk, so that you can appear to have more memory than you actually do. Paging significantly impacts performance. Eventually, after you create enough objects, you'll run out of memory. The trick is that the garbage collector steps in, and while it collects the garbage it compacts all the objects in the heap so that you've effectively moved the "heap pointer" closer to the beginning of the conveyor belt and farther away from a page fault. The garbage collector rearranges things and makes it possible for the high-speed, infinite-free-heap model to be used while allocating storage.

To understand garbage collection in Java, it's helpful learn how garbage-collection schemes work in other systems. A simple but slow garbage-collection technique is called *reference counting*. This means that each object contains a reference counter, and every time a reference is attached to that object, the reference count is increased. Every time a reference goes out of scope or is set to **null**, the reference count is decreased. Thus, managing reference counts is a small but

constant overhead that happens throughout the lifetime of your program. The garbage collector moves through the entire list of objects, and when it finds one with a reference count of zero it releases that storage (however, reference counting schemes often release an object as soon as the count goes to zero). The one drawback is that if objects circularly refer to each other they can have nonzero reference counts while still being garbage. Locating such self-referential groups requires significant extra work for the garbage collector. Reference counting is commonly used to explain one kind of garbage collection, but it doesn't seem to be used in any JVM implementations.

In faster schemes, garbage collection is not based on reference counting. Instead, it is based on the idea that any non-dead object must ultimately be traceable back to a reference that lives either on the stack or in static storage. The chain might go through several layers of objects. Thus, if you start in the stack and in the static storage area and walk through all the references, you'll find all the live objects. For each reference that you find, you must trace into the object that it points to and then follow all the references in *that* object, tracing into the objects they point to, etc., until you've moved through the entire Web that originated with the reference on the stack or in static storage. Each object that you move through must still be alive. Note that there is no problem with detached self-referential groups—these are simply not found, and are therefore automatically garbage.

In the approach described here, the JVM uses an *adaptive* garbage-collection scheme, and what it does with the live objects that it locates depends on the variant currently being used. One of these variants is *stop-and-copy*. This means that—for reasons that will become apparent—the program is first stopped (this is not a background collection scheme). Then, each live object is copied from one heap to another, leaving behind all the garbage. In addition, as the objects are copied into the new heap, they are packed end-to-end, thus compacting the new heap (and allowing new storage to simply be reeled off the end as previously described).

Of course, when an object is moved from one place to another, all references that point at the object must be changed. The reference that goes from the heap or the static storage area to the object can be changed right away, but there can be other references pointing to this object that will be encountered later during the "walk." These are fixed up as they are found (you could imagine a table that maps old addresses to new ones).

There are two issues that make these so-called "copy collectors" inefficient. The first is the idea that you have two heaps and you slosh all the memory back and forth between these two separate heaps, maintaining twice as much memory as you actually need. Some JVMs deal with this by allocating the heap in chunks as needed and simply copying from one chunk to another.

The second issue is the copying process itself. Once your program becomes stable, it might be generating little or no garbage. Despite that, a copy collector will still copy all the memory from one place to another, which is wasteful. To prevent this, some JVMs detect that no new garbage is being generated and switch to a different scheme (this is the "adaptive" part). This other scheme is called *mark-and-sweep*, and it's what earlier versions of Sun's JVM used all the time. For general use, mark-and-sweep is fairly slow, but when you know you're generating little or no garbage, it's fast.

Mark-and-sweep follows the same logic of starting from the stack and static storage, and tracing through all the references to find live objects. However, each time it finds a live object, that object is marked by setting a flag in it, but the object isn't collected yet. Only when the marking process is finished does the sweep occur. During the sweep, the dead objects are released. However, no copying happens, so if the collector chooses to compact a fragmented heap, it does so by shuffling objects around.

"Stop-and-copy" refers to the idea that this type of garbage collection is *not* done in the background; instead, the program is stopped while the garbage collection occurs. In the Sun literature you'll find many references to garbage collection as a low-priority background process, but it turns out that the garbage collection was not implemented that way in earlier versions of the Sun JVM. Instead, the Sun garbage collector stopped the program when memory got low. Mark-and-sweep also requires that the program be stopped.

As previously mentioned, in the JVM described here memory is allocated in big blocks. If you allocate a large object, it gets its own block. Strict stop-and-copy requires copying every live object from the source heap to a new heap before you can free the old one, which translates to lots of memory. With blocks, the garbage collection can typically copy objects to dead blocks as it collects. Each block has a *generation count* to keep track of whether it's alive. In the normal case, only the blocks created since the last garbage collection are compacted; all other blocks get their generation count bumped if they have been referenced from somewhere. This handles the normal case of lots of short-lived temporary objects. Periodically, a full sweep is made—large objects are still not copied (they just get their generation count bumped), and blocks containing small objects are copied and compacted. The JVM monitors the efficiency of garbage collection and if it becomes a waste of time because all objects are long-lived, then it switches to mark-and-sweep. Similarly, the JVM keeps track of how successful mark-and-sweep is, and if the heap starts to become fragmented, it switches back to stop-and-copy. This is where the "adaptive" part comes in, so you end up with a mouthful: "Adaptive generational stop-and-copy mark-and-sweep."

There are a number of additional speedups possible in a JVM. An especially important one involves the operation of the loader and what is called a *just-in-time* (JIT) compiler. A JIT compiler partially or fully converts a program into native machine code so that it doesn't need to be interpreted by the JVM and thus runs much faster. When a class must be loaded (typically, the first time you want to create an object of that class), the **.class** file is located, and the bytecodes for that class are brought into memory. At this point, one approach is to simply JIT compile all the code, but this has two drawbacks: It takes a little more time, which, compounded throughout the life of the program, can add up; and it increases the size of the executable (bytecodes are significantly more compact than expanded JIT code), and this might cause paging, which definitely slows down a program. An alternative approach is *lazy evaluation,* which means that the code is not JIT compiled until necessary. Thus, code that never gets executed might never be JIT compiled. The Java HotSpot technologies in recent JDKs take a similar approach by increasingly optimizing a piece of code each time it is executed, so the more the code is executed, the faster it gets.

Member initialization

Java goes out of its way to guarantee that variables are properly initialized before they are used. In the case of a method's local variables, this guarantee comes in the form of a compile-time error. So if you say:

```
void f() {
  int i;
  i++; // Error -- i not initialized
}
```

you'll get an error message that says that **i** might not have been initialized. Of course, the compiler could have given **i** a default value, but an uninitialized local variable is probably a programmer error, and a default value would have covered that up. Forcing the programmer to provide an initialization value is more likely to catch a bug.

If a primitive is a field in a class, however, things are a bit different. As you saw in the *Everything Is an Object* chapter, each primitive field of a class is guaranteed to get an initial value. Here's a program that verifies this, and shows the values:

```
//: initialization/InitialValues.java
// Shows default initial values.
import static net.mindview.util.Print.*;

public class InitialValues {
  boolean t;
  char c;
  byte b;
  short s;
  int i;
  long l;
  float f;
  double d;
  InitialValues reference;
  void printInitialValues() {
    print("Data type      Initial value");
    print("boolean        " + t);
    print("char           [" + c + "]");
    print("byte           " + b);
    print("short          " + s);
    print("int            " + i);
    print("long           " + l);
    print("float          " + f);
    print("double         " + d);
    print("reference      " + reference);
  }
  public static void main(String[] args) {
    InitialValues iv = new InitialValues();
    iv.printInitialValues();
    /* You could also say:
    new InitialValues().printInitialValues();
    */
  }
} /* Output:
Data type      Initial value
boolean        false
char           [ ]
byte           0
short          0
int            0
long           0
float          0.0
double         0.0
reference      null
*///:~
```

You can see that even though the values are not specified, they automatically get initialized (the **char** value is a zero, which prints as a space). So at least there's no threat of working with uninitialized variables.

When you define an object reference inside a class without initializing it to a new object, that reference is given a special value of **null**.

Specifying initialization

What happens if you want to give a variable an initial value? One direct way to do this is simply to assign the value at the point you define the variable in the class. (Notice you cannot do this in C++, although C++ novices always try.) Here the field definitions in class **InitialValues** are changed to provide initial values:

```
//: initialization/InitialValues2.java
// Providing explicit initial values.

public class InitialValues2 {
  boolean bool = true;
  char ch = 'x';
  byte b = 47;
  short s = 0xff;
  int i = 999;
  long lng = 1;
  float f = 3.14f;
  double d = 3.14159;
} ///:~
```

You can also initialize non-primitive objects in this same way. If **Depth** is a class, you can create a variable and initialize it like so:

```
//: initialization/Measurement.java
class Depth {}

public class Measurement {
  Depth d = new Depth();
  // ...
} ///:~
```

If you haven't given **d** an initial value and you try to use it anyway, you'll get a runtime error called an *exception* (covered in the *Error Handling with Exceptions* chapter).

You can even call a method to provide an initialization value:

```
//: initialization/MethodInit.java
public class MethodInit {
  int i = f();
  int f() { return 11; }
} ///:~
```

This method can have arguments, of course, but those arguments cannot be other class members that haven't been initialized yet. Thus, you can do this:

```
//: initialization/MethodInit2.java
public class MethodInit2 {
  int i = f();
  int j = g(i);
  int f() { return 11; }
  int g(int n) { return n * 10; }
} ///:~
```

But you cannot do this:

```
//: initialization/MethodInit3.java
public class MethodInit3 {
  //! int j = g(i); // Illegal forward reference
```

```
  int i = f();
  int f() { return 11; }
  int g(int n) { return n * 10; }
} ///:~
```

This is one place in which the compiler, appropriately, *does* complain about forward referencing, since this has to do with the order of initialization and not the way the program is compiled.

This approach to initialization is simple and straightforward. It has the limitation that *every* object of type **InitialValues** will get these same initialization values. Sometimes this is exactly what you need, but at other times you need more flexibility.

Constructor initialization

The constructor can be used to perform initialization, and this gives you greater flexibility in your programming because you can call methods and perform actions at run time to determine the initial values. There's one thing to keep in mind, however: You aren't precluding the automatic initialization, which happens before the constructor is entered. So, for example, if you say:

```
//: initialization/Counter.java
public class Counter {
  int i;
  Counter() { i = 7; }
  // ...
} ///:~
```

then **i** will first be initialized to 0, then to 7. This is true with all the primitive types and with object references, including those that are given explicit initialization at the point of definition. For this reason, the compiler doesn't try to force you to initialize elements in the constructor at any particular place, or before they are used—initialization is already guaranteed.

Order of initialization

Within a class, the order of initialization is determined by the order that the variables are defined within the class. The variable definitions may be scattered throughout and in between method definitions, but the variables are initialized before any methods can be called—even the constructor. For example:

```
//: initialization/OrderOfInitialization.java
// Demonstrates initialization order.
import static net.mindview.util.Print.*;

// When the constructor is called to create a
// Window object, you'll see a message:
class Window {
  Window(int marker) { print("Window(" + marker + ")"); }
}

class House {
  Window w1 = new Window(1); // Before constructor
  House() {
    // Show that we're in the constructor:
    print("House()");
    w3 = new Window(33); // Reinitialize w3
  }
  Window w2 = new Window(2); // After constructor
  void f() { print("f()"); }
```

```
  Window w3 = new Window(3); // At end
}

public class OrderOfInitialization {
  public static void main(String[] args) {
    House h = new House();
    h.f(); // Shows that construction is done
  }
} /* Output:
Window(1)
Window(2)
Window(3)
House()
Window(33)
f()
*///:~
```

In **House**, the definitions of the **Window** objects are intentionally scattered about to prove that they'll all get initialized before the constructor is entered or anything else can happen. In addition, **w3** is reinitialized inside the constructor.

From the output, you can see that the **w3** reference gets initialized twice: once before and once during the constructor call. (The first object is dropped, so it can be garbage collected later.) This might not seem efficient at first, but it guarantees proper initialization—what would happen if an overloaded constructor were defined that did *not* initialize **w3** and there wasn't a "default" initialization for **w3** in its definition?

static data initialization

There's only a single piece of storage for a **static**, regardless of how many objects are created. You can't apply the **static** keyword to local variables, so it only applies to fields. If a field is a **static** primitive and you don't initialize it, it gets the standard initial value for its type. If it's a reference to an object, the default initialization value is **null**.

If you want to place initialization at the point of definition, it looks the same as for non-**static**s.

To see *when* the **static** storage gets initialized, here's an example:

```
//: initialization/StaticInitialization.java
// Specifying initial values in a class definition.
import static net.mindview.util.Print.*;

class Bowl {
  Bowl(int marker) {
    print("Bowl(" + marker + ")");
  }
  void f1(int marker) {
    print("f1(" + marker + ")");
  }
}

class Table {
  static Bowl bowl1 = new Bowl(1);
  Table() {
    print("Table()");
    bowl2.f1(1);
  }
```

```
  void f2(int marker) {
    print("f2(" + marker + ")");
  }
  static Bowl bowl2 = new Bowl(2);
}

class Cupboard {
  Bowl bowl3 = new Bowl(3);
  static Bowl bowl4 = new Bowl(4);
  Cupboard() {
    print("Cupboard()");
    bowl4.f1(2);
  }
  void f3(int marker) {
    print("f3(" + marker + ")");
  }
  static Bowl bowl5 = new Bowl(5);
}

public class StaticInitialization {
  public static void main(String[] args) {
    print("Creating new Cupboard() in main");
    new Cupboard();
    print("Creating new Cupboard() in main");
    new Cupboard();
    table.f2(1);
    cupboard.f3(1);
  }
  static Table table = new Table();
  static Cupboard cupboard = new Cupboard();
} /* Output:
Bowl(1)
Bowl(2)
Table()
f1(1)
Bowl(4)
Bowl(5)
Bowl(3)
Cupboard()
f1(2)
Creating new Cupboard() in main
Bowl(3)
Cupboard()
f1(2)
Creating new Cupboard() in main
Bowl(3)
Cupboard()
f1(2)
f2(1)
f3(1)
*///:~
```

Bowl allows you to view the creation of a class, and **Table** and **Cupboard** have **static** members of **Bowl** scattered through their class definitions. Note that **Cupboard** creates a non-**static** **Bowl bowl3** prior to the **static** definitions.

From the output, you can see that the **static** initialization occurs only if it's necessary. If you don't create a **Table** object and you never refer to **Table.bowl1** or **Table.bowl2**, the **static Bowl bowl1** and **bowl2** will never be created. They are initialized only when the *first* **Table** object is created (or the first **static** access occurs). After that, the **static** objects are not reinitialized.

The order of initialization is **static**s first, if they haven't already been initialized by a previous object creation, and then the non-**static** objects. You can see the evidence of this in the output. To execute **main( )** (a **static** method), the **StaticInitialization** class must be loaded, and its **static** fields **table** and **cupboard** are then initialized, which causes *those* classes to be loaded, and since they both contain **static Bowl** objects, **Bowl** is then loaded. Thus, all the classes in this particular program get loaded before **main( )** starts. This is usually not the case, because in typical programs you won't have everything linked together by **static**s as you do in this example.

To summarize the process of creating an object, consider a class called **Dog**:

1. Even though it doesn't explicitly use the **static** keyword, the constructor is actually a **static** method. So the first time an object of type **Dog** is created, *or* the first time a **static** method or **static** field of class **Dog** is accessed, the Java interpreter must locate **Dog.class**, which it does by searching through the classpath.
2. As **Dog.class** is loaded (creating a **Class** object, which you'll learn about later), all of its **static** initializers are run. Thus, **static** initialization takes place only once, as the **Class** object is loaded for the first time.
3. When you create a **new Dog( )**, the construction process for a **Dog** object first allocates enough storage for a **Dog** object on the heap.
4. This storage is wiped to zero, automatically setting all the primitives in that **Dog** object to their default values (zero for numbers and the equivalent for **boolean** and **char**) and the references to **null**.
5. Any initializations that occur at the point of field definition are executed.
6. Constructors are executed. As you shall see in the *Reusing Classes* chapter, this might actually involve a fair amount of activity, especially when inheritance is involved.

Explicit **static** initialization

Java allows you to group other **static** initializations inside a special "**static** clause" (sometimes called a *static block*) in a class. It looks like this:

```
//: initialization/Spoon.java
public class Spoon {
  static int i;
  static {
    i = 47;
  }
} ///:~
```

It appears to be a method, but it's just the **static** keyword followed by a block of code. This code, like other **static** initializations, is executed only once: the first time you make an object of that class *or* the first time you access a **static** member of that class (even if you never make an object of that class). For example:

```
//: initialization/ExplicitStatic.java
```

```
// Explicit static initialization with the "static" clause.
import static net.mindview.util.Print.*;

class Cup {
  Cup(int marker) {
    print("Cup(" + marker + ")");
  }
  void f(int marker) {
    print("f(" + marker + ")");
  }
}

class Cups {
  static Cup cup1;
  static Cup cup2;
  static {
    cup1 = new Cup(1);
    cup2 = new Cup(2);
  }
  Cups() {
    print("Cups()");
  }
}

public class ExplicitStatic {
  public static void main(String[] args) {
    print("Inside main()");
    Cups.cup1.f(99);  // (1)
  }
  // static Cups cups1 = new Cups();  // (2)
  // static Cups cups2 = new Cups();  // (2)
} /* Output:
Inside main()
Cup(1)
Cup(2)
f(99)
*///:~
```

The **static** initializers for **Cups** run when either the access of the **static** object **cup1** occurs on the line marked **(1)**, or if line **(1)** is commented out and the lines marked **(2)** are uncommented. If both **(1)** and **(2)** are commented out, the **static** initialization for **Cups** never occurs, as you can see from the output. Also, it doesn't matter if one or both of the lines marked **(2)** are uncommented; the static initialization only occurs once.

Exercise 13: (1) Verify the statements in the previous paragraph.

Exercise 14: (1) Create a class with a **static String** field that is initialized at the point of definition, and another one that is initialized by the **static** block. Add a **static** method that prints both fields and demonstrates that they are both initialized before they are used.

Non-**static** instance initialization

Java provides a similar syntax, called *instance initialization*, for initializing non-**static** variables for each object. Here's an example:

```
//: initialization/Mugs.java
// Java "Instance Initialization."
```

```
import static net.mindview.util.Print.*;

class Mug {
  Mug(int marker) {
    print("Mug(" + marker + ")");
  }
  void f(int marker) {
    print("f(" + marker + ")");
  }
}

public class Mugs {
  Mug mug1;
  Mug mug2;
  {
    mug1 = new Mug(1);
    mug2 = new Mug(2);
    print("mug1 & mug2 initialized");
  }
  Mugs() {
    print("Mugs()");
  }
  Mugs(int i) {
    print("Mugs(int)");
  }
  public static void main(String[] args) {
    print("Inside main()");
    new Mugs();
    print("new Mugs() completed");
    new Mugs(1);
    print("new Mugs(1) completed");
  }
} /* Output:
Inside main()
Mug(1)
Mug(2)
mug1 & mug2 initialized
Mugs()
new Mugs() completed
Mug(1)
Mug(2)
mug1 & mug2 initialized
Mugs(int)
new Mugs(1) completed
*///:~
```

You can see that the instance initialization clause:

```
  {
    mug1 = new Mug(1);
    mug2 = new Mug(2);
    print("mug1 & mug2 initialized");
  }
```

looks exactly like the static initialization clause except for the missing **static** keyword. This syntax is necessary to support the initialization of *anonymous inner classes* (see the *Inner Classes* chapter), but it also allows you to guarantee that certain operations occur regardless of

which explicit constructor is called. From the output, you can see that the instance initialization clause is executed before either one of the constructors.

Exercise 15: (1) Create a class with a **String** that is initialized using instance initialization.

Array initialization

An array is simply a sequence of either objects or primitives that are all the same type and are packaged together under one identifier name. Arrays are defined and used with the square-brackets *indexing operator* **[ ]**. To define an array reference, you simply follow your type name with empty square brackets:

```
int[] a1;
```

You can also put the square brackets after the identifier to produce exactly the same meaning:

```
int a1[];
```

This conforms to expectations from C and C++ programmers. The former style, however, is probably a more sensible syntax, since it says that the type is "an **int** array." That style will be used in this book.

The compiler doesn't allow you to tell it how big the array is. This brings us back to that issue of "references." All that you have at this point is a reference to an array (you've allocated enough storage for that reference), and there's been no space allocated for the array object itself. To create storage for the array, you must write an initialization expression. For arrays, initialization can appear anywhere in your code, but you can also use a special kind of initialization expression that must occur at the point where the array is created. This special initialization is a set of values surrounded by curly braces. The storage allocation (the equivalent of using **new**) is taken care of by the compiler in this case. For example:

```
int[] a1 = { 1, 2, 3, 4, 5 };
```

So why would you ever define an array reference without an array?

```
int[] a2;
```

Well, it's possible to assign one array to another in Java, so you can say:

```
a2 = a1;
```

What you're really doing is copying a reference, as demonstrated here:

```
//: initialization/ArraysOfPrimitives.java
import static net.mindview.util.Print.*;

public class ArraysOfPrimitives {
  public static void main(String[] args) {
    int[] a1 = { 1, 2, 3, 4, 5 };
    int[] a2;
    a2 = a1;
    for(int i = 0; i < a2.length; i++)
      a2[i] = a2[i] + 1;
    for(int i = 0; i < a1.length; i++)
      print("a1[" + i + "] = " + a1[i]);
  }
} /* Output:
a1[0] = 2
```

```
a1[1] = 3
a1[2] = 4
a1[3] = 5
a1[4] = 6
*///:~
```

You can see that **a1** is given an initialization value but **a2** is not; **a2** is assigned later—in this case, to another array. Since **a2** and **a1** are then aliased to the same array, the changes made via **a2** are seen in **a1**.

All arrays have an intrinsic member (whether they're arrays of objects or arrays of primitives) that you can query—but not change—to tell you how many elements there are in the array. This member is **length**. Since arrays in Java, like C and C++, start counting from element zero, the largest element you can index is **length - 1**. If you go out of bounds, C and C++ quietly accept this and allow you to stomp all over your memory, which is the source of many infamous bugs. However, Java protects you against such problems by causing a runtime error (an *exception*) if you step out of bounds.[5]

What if you don't know how many elements you're going to need in your array while you're writing the program? You simply use **new** to create the elements in the array. Here, **new** works even though it's creating an array of primitives (**new** won't create a non-array primitive):

```
//: initialization/ArrayNew.java
// Creating arrays with new.
import java.util.*;
import static net.mindview.util.Print.*;

public class ArrayNew {
  public static void main(String[] args) {
    int[] a;
    Random rand = new Random(47);
    a = new int[rand.nextInt(20)];
    print("length of a = " + a.length);
    print(Arrays.toString(a));
  }
} /* Output:
length of a = 18
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
*///:~
```

The size of the array is chosen at random by using the **Random.nextInt( )** method, which produces a value between zero and that of its argument. Because of the randomness, it's clear that array creation is actually happening at run time. In addition, the output of this program shows that array elements of primitive types are automatically initialized to "empty" values. (For numerics and **char**, this is zero, and for **boolean**, it's **false**.)

The **Arrays.toString( )** method, which is part of the standard **java.util** library, produces a printable version of a one-dimensional array.

---

[5] Of course, checking every array access costs time and code and there's no way to turn it off, which means that array accesses might be a source of inefficiency in your program if they occur at a critical juncture. For Internet security and programmer productivity, the Java designers saw that this was a worthwhile trade-off. Although you may be tempted to write code that you think might make array accesses more efficient, this is a waste of time because automatic compile-time and runtime optimizations will speed array accesses.

Of course, in this case the array could also have been defined and initialized in the same statement:

```
int[] a = new int[rand.nextInt(20)];
```

This is the preferred way to do it, if you can.

If you create a non-primitive array, you create an array of references. Consider the wrapper type **Integer**, which is a class and not a primitive:

```
//: initialization/ArrayClassObj.java
// Creating an array of nonprimitive objects.
import java.util.*;
import static net.mindview.util.Print.*;

public class ArrayClassObj {
  public static void main(String[] args) {
    Random rand = new Random(47);
    Integer[] a = new Integer[rand.nextInt(20)];
    print("length of a = " + a.length);
    for(int i = 0; i < a.length; i++)
      a[i] = rand.nextInt(500); // Autoboxing
    print(Arrays.toString(a));
  }
} /* Output: (Sample)
length of a = 18
[55, 193, 361, 461, 429, 368, 200, 22, 207, 288, 128, 51, 89, 309, 278,
498, 361, 20]
*///:~
```

Here, even after **new** is called to create the array:

```
Integer[] a = new Integer[rand.nextInt(20)];
```

it's only an array of references, and the initialization is not complete until the reference itself is initialized by creating a new **Integer** object (via autoboxing, in this case):

```
a[i] = rand.nextInt(500);
```

If you forget to create the object, however, you'll get an exception at run time when you try to use the empty array location.

It's also possible to initialize arrays of objects by using the curly brace-enclosed list. There are two forms:

```
//: initialization/ArrayInit.java
// Array initialization.
import java.util.*;

public class ArrayInit {
  public static void main(String[] args) {
    Integer[] a = {
      new Integer(1),
      new Integer(2),
      3, // Autoboxing
    };
    Integer[] b = new Integer[]{
      new Integer(1),
      new Integer(2),
```

```
      3, // Autoboxing
    };
    System.out.println(Arrays.toString(a));
    System.out.println(Arrays.toString(b));
  }
} /* Output:
[1, 2, 3]
[1, 2, 3]
*///:~
```

In both cases, the final comma in the list of initializers is optional. (This feature makes for easier maintenance of long lists.)

Although the first form is useful, it's more limited because it can only be used at the point where the array is defined. You can use the second and third forms anywhere, even inside a method call. For example, you could create an array of **String** objects to pass to the **main( )** of another method, to provide alternate command-line arguments to that **main( )**:

```
//: initialization/DynamicArray.java
// Array initialization.

public class DynamicArray {
  public static void main(String[] args) {
    Other.main(new String[]{ "fiddle", "de", "dum" });
  }
}

class Other {
  public static void main(String[] args) {
    for(String s : args)
      System.out.print(s + " ");
  }
} /* Output:
fiddle de dum
*///:~
```

The array created for the argument of **Other.main( )** is created at the point of the method call, so you can even provide alternate arguments at the time of the call.

Exercise 16: (1) Create an array of **String** objects and assign a **String** to each element. Print the array by using a **for** loop.

Exercise 17: (2) Create a class with a constructor that takes a **String** argument. During construction, print the argument. Create an array of object references to this class, but don't actually create objects to assign into the array. When you run the program, notice whether the initialization messages from the constructor calls are printed.

Exercise 18: (1) Complete the previous exercise by creating objects to attach to the array of references.

Variable argument lists

The second form provides a convenient syntax to create and call methods that can produce an effect similar to C's *variable argument lists* (known as "varargs" in C). These can include unknown quantities of arguments as well as unknown types. Since all classes are ultimately inherited from the common root class **Object** (a subject you will learn more about as this book progresses), you can create a method that takes an array of **Object** and call it like this:

```
//: initialization/VarArgs.java
// Using array syntax to create variable argument lists.

class A {}

public class VarArgs {
  static void printArray(Object[] args) {
    for(Object obj : args)
      System.out.print(obj + " ");
    System.out.println();
  }
  public static void main(String[] args) {
    printArray(new Object[]{
      new Integer(47), new Float(3.14), new Double(11.11)
    });
    printArray(new Object[]{"one", "two", "three" });
    printArray(new Object[]{new A(), new A(), new A()});
  }
} /* Output: (Sample)
47 3.14 11.11
one two three
A@1a46e30 A@3e25a5 A@19821f
*///:~
```

You can see that **print( )** takes an array of **Object**, then steps through the array using the foreach syntax and prints each one. The standard Java library classes produce sensible output, but the objects of the classes created here print the class name, followed by an '@' sign and hexadecimal digits. Thus, the default behavior (if you don't define a **toString( )** method for your class, which will be described later in the book) is to print the class name and the address of the object.

You may see pre-Java SE5 code written like the above in order to produce variable argument lists. In Java SE5, however, this long-requested feature was finally added, so you can now use ellipses to define a variable argument list, as you can see in **printArray( )**:

```
//: initialization/NewVarArgs.java
// Using array syntax to create variable argument lists.

public class NewVarArgs {
  static void printArray(Object... args) {
    for(Object obj : args)
      System.out.print(obj + " ");
    System.out.println();
  }
  public static void main(String[] args) {
    // Can take individual elements:
    printArray(new Integer(47), new Float(3.14),
      new Double(11.11));
    printArray(47, 3.14F, 11.11);
    printArray("one", "two", "three");
    printArray(new A(), new A(), new A());
    // Or an array:
    printArray((Object[])new Integer[]{ 1, 2, 3, 4 });
    printArray(); // Empty list is OK
  }
} /* Output: (75% match)
47 3.14 11.11
```

```
47 3.14 11.11
one two three
A@1bab50a A@c3c749 A@150bd4d
1 2 3 4
*///:~
```

With varargs, you no longer have to explicitly write out the array syntax—the compiler will actually fill it in for you when you specify varargs. You're still getting an array, which is why **print( )** is able to use foreach to iterate through the array. However, it's more than just an automatic conversion from a list of elements to an array. Notice the second-to-last line in the program, where an array of **Integer** (created using autoboxing) is cast to an **Object** array (to remove a compiler warning) and passed to **printArray( )**. Clearly, the compiler sees that this is already an array and performs no conversion on it. So if you have a group of items you can pass them in as a list, and if you already have an array it will accept that as the variable argument list.

The last line of the program shows that it's possible to pass zero arguments to a vararg list. This is helpful when you have optional trailing arguments:

```
//: initialization/OptionalTrailingArguments.java

public class OptionalTrailingArguments {
  static void f(int required, String... trailing) {
    System.out.print("required: " + required + " ");
    for(String s : trailing)
      System.out.print(s + " ");
    System.out.println();
  }
  public static void main(String[] args) {
    f(1, "one");
    f(2, "two", "three");
    f(0);
  }
} /* Output:
required: 1 one
required: 2 two three
required: 0
*///:~
```

This also shows how you can use varargs with a specified type other than **Object**. Here, all the varargs must be **String** objects. It's possible to use any type of argument in varargs, including a primitive type. The following example also shows that the vararg list becomes an array, and if there's nothing in the list it's an array of size zero:

```
//: initialization/VarargType.java

public class VarargType {
  static void f(Character... args) {
    System.out.print(args.getClass());
    System.out.println(" length " + args.length);
  }
  static void g(int... args) {
    System.out.print(args.getClass());
    System.out.println(" length " + args.length);
  }
  public static void main(String[] args) {
    f('a');
    f();
```

```
    g(1);
    g();
    System.out.println("int[]: " + new int[0].getClass());
  }
} /* Output:
class [Ljava.lang.Character; length 1
class [Ljava.lang.Character; length 0
class [I length 1
class [I length 0
int[]: class [I
*///:~
```

The **getClass( )** method is part of **Object**, and will be explored fully in the *Type Information* chapter. It produces the class of an object, and when you print this class, you see an encoded string representing the class type. The leading '**[**' indicates that this is an array of the type that follows. The '**I**' is for a primitive **int**; to double-check, I created an array of **int** in the last line and printed its type. This verifies that using varargs does not depend on autoboxing, but that it actually uses the primitive types.

Varargs do work in harmony with autoboxing, however. For example:

```
//: initialization/AutoboxingVarargs.java

public class AutoboxingVarargs {
  public static void f(Integer... args) {
    for(Integer i : args)
      System.out.print(i + " ");
    System.out.println();
  }
  public static void main(String[] args) {
    f(new Integer(1), new Integer(2));
    f(4, 5, 6, 7, 8, 9);
    f(10, new Integer(11), 12);
  }
} /* Output:
1 2
4 5 6 7 8 9
10 11 12
*///:~
```

Notice that you can mix the types together in a single argument list, and autoboxing selectively promotes the **int** arguments to **Integer**.

Varargs complicate the process of overloading, although it seems safe enough at first:

```
//: initialization/OverloadingVarargs.java

public class OverloadingVarargs {
  static void f(Character... args) {
    System.out.print("first");
    for(Character c : args)
      System.out.print(" " + c);
    System.out.println();
  }
  static void f(Integer... args) {
    System.out.print("second");
    for(Integer i : args)
      System.out.print(" " + i);
```

```
    System.out.println();
  }
  static void f(Long... args) {
    System.out.println("third");
  }
  public static void main(String[] args) {
    f('a', 'b', 'c');
    f(1);
    f(2, 1);
    f(0);
    f(0L);
    //! f(); // Won't compile -- ambiguous
  }
} /* Output:
first a b c
second 1
second 2 1
second 0
third
*///:~
```

In each case, the compiler is using autoboxing to match the overloaded method, and it calls the most specifically matching method.

But when you call **f( )** without arguments, it has no way of knowing which one to call. Although this error is understandable, it will probably surprise the client programmer.

You might try solving the problem by adding a non-vararg argument to one of the methods:

```
//: initialization/OverloadingVarargs2.java
// {CompileTimeError} (Won't compile)

public class OverloadingVarargs2 {
  static void f(float i, Character... args) {
    System.out.println("first");
  }
  static void f(Character... args) {
    System.out.print("second");
  }
  public static void main(String[] args) {
    f(1, 'a');
    f('a', 'b');
  }
} ///:~
```

The **{CompileTimeError}** comment tag excludes the file from this book's Ant build. If you compile it by hand you'll see the error message:

reference to f is ambiguous, both method f(float,java.lang.Character...) in OverloadingVarargs2 and method f(java.lang.Character...) in OverloadingVarargs2 match

If you give *both* methods a non-vararg argument, it works:

```
//: initialization/OverloadingVarargs3.java

public class OverloadingVarargs3 {
  static void f(float i, Character... args) {
    System.out.println("first");
```

```
  }
  static void f(char c, Character... args) {
    System.out.println("second");
  }
  public static void main(String[] args) {
    f(1, 'a');
    f('a', 'b');
  }
} /* Output:
first
second
*///:~
```

You should generally only use a variable argument list on one version of an overloaded method. Or consider not doing it at all.

Exercise 19: (2) Write a method that takes a vararg **String** array. Verify that you can pass either a comma-separated list of **String**s or a **String[]** into this method.

Exercise 20: (1) Create a **main( )** that uses varargs instead of the ordinary **main( )** syntax. Print all the elements in the resulting **args** array. Test it with various numbers of command-line arguments.

Enumerated types

An apparently small addition in Java SE5 is the **enum** keyword, which makes your life much easier when you need to group together and use a set of *enumerated types*. In the past you would have created a set of constant integral values, but these do not naturally restrict themselves to your set and thus are riskier and more difficult to use. Enumerated types are a common enough need that C, C++, and a number of other languages have always had them. Before Java SE5, Java programmers were forced to know a lot and be quite careful when they wanted to properly produce the **enum** effect. Now Java has **enum**, too, and it's much more full-featured than what you find in C/C++. Here's a simple example:

```
//: initialization/Spiciness.java
public enum Spiciness {
  NOT, MILD, MEDIUM, HOT, FLAMING
} ///:~
```

This creates an enumerated type called **Spiciness** with five named values. Because the instances of enumerated types are constants, they are in all capital letters by convention (if there are multiple words in a name, they are separated by underscores).

To use an **enum**, you create a reference of that type and assign it to an instance:

```
//: initialization/SimpleEnumUse.java
public class SimpleEnumUse {
  public static void main(String[] args) {
    Spiciness howHot = Spiciness.MEDIUM;
    System.out.println(howHot);
  }
} /* Output:
MEDIUM
*///:~
```

The compiler automatically adds useful features when you create an **enum**. For example, it creates a **toString( )** so that you can easily display the name of an **enum** instance, which is how

the print statement above produced its output. The compiler also creates an **ordinal( )** method to indicate the declaration order of a particular **enum** constant, and a **static values( )** method that produces an array of values of the **enum** constants in the order that they were declared:

```
//: initialization/EnumOrder.java
public class EnumOrder {
  public static void main(String[] args) {
    for(Spiciness s : Spiciness.values())
      System.out.println(s + ", ordinal " + s.ordinal());
  }
} /* Output:
NOT, ordinal 0
MILD, ordinal 1
MEDIUM, ordinal 2
HOT, ordinal 3
FLAMING, ordinal 4
*///:~
```

Although **enum**s appear to be a new data type, the keyword only produces some compiler behavior while generating a class for the **enum**, so in many ways you can treat an **enum** as if it were any other class. In fact, **enum**s *are* classes and have their own methods.

An especially nice feature is the way that **enum**s can be used inside **switch** statements:

```
//: initialization/Burrito.java

public class Burrito {
  Spiciness degree;
  public Burrito(Spiciness degree) { this.degree = degree;}
  public void describe() {
    System.out.print("This burrito is ");
    switch(degree) {
      case NOT:    System.out.println("not spicy at all.");
                   break;
      case MILD:
      case MEDIUM: System.out.println("a little hot.");
                   break;
      case HOT:
      case FLAMING:
      default:     System.out.println("maybe too hot.");
    }
  }
  public static void main(String[] args) {
    Burrito
      plain = new Burrito(Spiciness.NOT),
      greenChile = new Burrito(Spiciness.MEDIUM),
      jalapeno = new Burrito(Spiciness.HOT);
    plain.describe();
    greenChile.describe();
    jalapeno.describe();
  }
} /* Output:
This burrito is not spicy at all.
This burrito is a little hot.
This burrito is maybe too hot.
*///:~
```

Since a **switch** is intended to select from a limited set of possibilities, it's an ideal match for an **enum**. Notice how the **enum** names can produce a much clearer indication of what the program means to do.

In general you can use an **enum** as if it were another way to create a data type, and then just put the results to work. That's the point, so you don't have to think too hard about them. Before the introduction of **enum** in Java SE5, you had to go to a lot of effort to make an equivalent enumerated type that was safe to use.

This is enough for you to understand and use basic **enum**s, but we'll look more deeply at them later in the book—they have their own chapter: *Enumerated Types*.

Exercise 21: (1) Create an **enum** of the least-valuable six types of paper currency. Loop through the **values( )** and print each value and its **ordinal( )**.

Exercise 22: (2) Write a **switch** statement for the **enum** in the previous example. For each **case**, output a description of that particular currency.

Summary

This seemingly elaborate mechanism for initialization, the constructor, should give you a strong hint about the critical importance placed on initialization in the language. As Bjarne Stroustrup, the inventor of C++, was designing that language, one of the first observations he made about productivity in C was that improper initialization of variables causes a significant portion of programming problems. These kinds of bugs are hard to find, and similar issues apply to improper cleanup. Because constructors allow you to *guarantee* proper initialization and cleanup (the compiler will not allow an object to be created without the proper constructor calls), you get complete control and safety.

In C++, destruction is quite important because objects created with **new** must be explicitly destroyed. In Java, the garbage collector automatically releases the memory for all objects, so the equivalent cleanup method in Java isn't necessary much of the time (but when it is, you must do it yourself). In cases where you don't need destructor-like behavior, Java's garbage collector greatly simplifies programming and adds much-needed safety in managing memory. Some garbage collectors can even clean up other resources like graphics and file handles. However, the garbage collector does add a runtime cost, the expense of which is difficult to put into perspective because of the historical slowness of Java interpreters. Although Java has had significant performance increases over time, the speed problem has taken its toll on the adoption of the language for certain types of programming problems.

Because of the guarantee that all objects will be constructed, there's actually more to the constructor than what is shown here. In particular, when you create new classes using either *composition* or *inheritance,* the guarantee of construction also holds, and some additional syntax is necessary to support this. You'll learn about composition, inheritance, and how they affect constructors in future chapters.

Solutions to selected exercises can be found in the electronic document *The Thinking in Java Annotated Solution Guide*, available for sale from *www.MindView.net*.

Access Control

Access control (or *implementation hiding*) is about "not getting it right the first time."

All good writers—including those who write software—know that a piece of work isn't good until it's been rewritten, often many times. If you leave a piece of code in a drawer for a while and come back to it, you may see a much better way to do it. This is one of the prime motivations for *refactoring*, which rewrites working code in order to make it more readable, understandable, and thus maintainable.[1]

There is a tension, however, in this desire to change and improve your code. There are often consumers (*client programmers*) who rely on some aspect of your code staying the same. So you want to change it; they want it to stay the same. Thus a primary consideration in object-oriented design is to "separate the things that change from the things that stay the same."

This is particularly important for libraries. Consumers of that library must rely on the part they use, and know that they won't need to rewrite code if a new version of the library comes out. On the flip side, the library creator must have the freedom to make modifications and improvements with the certainty that the client code won't be affected by those changes.

This can be achieved through convention. For example, the library programmer must agree not to remove existing methods when modifying a class in the library, since that would break the client programmer's code. The reverse situation is thornier, however. In the case of a field, how can the library creator know which fields have been accessed by client programmers? This is also true with methods that are only part of the implementation of a class, and not meant to be used directly by the client programmer. What if the library creator wants to rip out an old implementation and put in a new one? Changing any of those members might break a client programmer's code. Thus the library creator is in a strait jacket and can't change anything.

To solve this problem, Java provides *access specifiers* to allow the library creator to say what is available to the client programmer and what is not. The levels of access control from "most access" to "least access" are **public**, **protected**, package access (which has no keyword), and **private**. From the previous paragraph you might think that, as a library designer, you'll want to keep everything as "private" as possible, and expose only the methods that you want the client programmer to use. This is exactly right, even though it's often counterintuitive for people who program in other languages (especially C) and who are used to accessing everything without restriction. By the end of this chapter you should be convinced of the value of access control in Java.

The concept of a library of components and the control over who can access the components of that library is not complete, however. There's still the question of how the components are bundled together into a cohesive library unit. This is controlled with the **package** keyword in Java, and the access specifiers are affected by whether a class is in the same package or in a

---

[1] See *Refactoring: Improving the Design of Existing Code*, by Martin Fowler, et al. (Addison-Wesley, 1999). Occasionally someone will argue against refactoring, suggesting that code which works is perfectly good and it's a waste of time to refactor it. The problem with this way of thinking is that the lion's share of a project's time and money is not in the initial writing of the code, but in maintaining it. Making code easier to understand translates into very significant dollars.

separate package. So to begin this chapter, you'll learn how library components are placed into packages. Then you'll be able to understand the complete meaning of the access specifiers.

package: the library unit

A package contains a group of classes, organized together under a single *namespace*.

For example, there's a utility library that's part of the standard Java distribution, organized under the namespace **java.util**. One of the classes in **java.util** is called **ArrayList**. One way to use an **ArrayList** is to specify the full name **java.util.ArrayList**.

```
//: access/FullQualification.java

public class FullQualification {
  public static void main(String[] args) {
    java.util.ArrayList list = new java.util.ArrayList();
  }
} ///:~
```

This rapidly becomes tedious, so you'll probably want to use the **import** keyword instead. If you want to import a single class, you can name that class in the **import** statement:

```
//: access/SingleImport.java
import java.util.ArrayList;

public class SingleImport {
  public static void main(String[] args) {
    ArrayList list = new java.util.ArrayList();
  }
} ///:~
```

Now you can use **ArrayList** with no qualification. However, none of the other classes in **java.util** are available. To import everything, you simply use the '*****' as you've been seeing in the rest of the examples in this book:

```
import java.util.*;
```

The reason for all this importing is to provide a mechanism to manage namespaces. The names of all your class members are insulated from each other. A method **f( )** inside a class **A** will not clash with an **f( )** that has the same signature in class **B**. But what about the class names? Suppose you create a **Stack** class that is installed on a machine that already has a **Stack** class that's written by someone else? This potential clashing of names is why it's important to have complete control over the namespaces in Java, and to create a unique identifier combination for each class.

Most of the examples thus far in this book have existed in a single file and have been designed for local use, so they haven't bothered with package names. These examples have actually been in packages: the "unnamed" or *default package*. This is certainly an option, and for simplicity's sake this approach will be used whenever possible throughout the rest of this book. However, if you're planning to create libraries or programs that are friendly to other Java programs on the same machine, you must think about preventing class name clashes.

When you create a source-code file for Java, it's commonly called a *compilation unit* (sometimes a *translation unit*). Each compilation unit must have a name ending in **.java**, and inside the compilation unit there can be a **public** class that must have the same name as the file (including capitalization, but excluding the **.java** file name extension). There can be only *one* **public** class

in each compilation unit; otherwise, the compiler will complain. If there are additional classes in that compilation unit, they are hidden from the world outside that package because they're *not* **public**, and they comprise "support" classes for the main **public** class.

Code organization

When you compile a **.java** file, you get an output file *for each class in the* **.java** file. Each output file has the name of a class in the **.java** file, but with an extension of **.class**. Thus you can end up with quite a few **.class** files from a small number of **.java** files. If you've programmed with a compiled language, you might be used to the compiler spitting out an intermediate form (usually an "obj" file) that is then packaged together with others of its kind using a linker (to create an executable file) or a librarian (to create a library). That's not how Java works. A working program is a bunch of **.class** files, which can be packaged and compressed into a Java ARchive (JAR) file (using Java's **jar** archiver). The Java interpreter is responsible for finding, loading, and interpreting[2] these files.

A library is a group of these class files. Each source file usually has a **public** class and any number of non-**public** classes, so there's one **public** component for each source file. If you want to say that all these components (each in its own separate **.java** and **.class** files) belong together, that's where the **package** keyword comes in.

If you use a **package** statement, it *must* appear as the first non-comment in the file. When you say:

```
package access;
```

you're stating that this compilation unit is part of a library named **access**. Put another way, you're saying that the **public** class name within this compilation unit is under the umbrella of the name **access**, and anyone who wants to use that name must either fully specify the name or use the **import** keyword in combination with **access**, using the choices given previously. (Note that the convention for Java package names is to use all lowercase letters, even for intermediate words.)

For example, suppose the name of the file is **MyClass.java**. This means there can be one and only one **public** class in that file, and the name of that class must be **MyClass** (including the capitalization):

```
//: access/mypackage/MyClass.java
package access.mypackage;

public class MyClass {
  // ...
} ///:~
```

Now, if someone wants to use **MyClass** or, for that matter, any of the other **public** classes in **access**, they must use the **import** keyword to make the name or names in **access** available. The alternative is to give the fully qualified name:

```
//: access/QualifiedMyClass.java

public class QualifiedMyClass {
```

[2] There's nothing in Java that forces the use of an interpreter. There exist native-code Java compilers that generate a single executable file.

```
  public static void main(String[] args) {
    access.mypackage.MyClass m =
      new access.mypackage.MyClass();
  }
} ///:~
```

The **import** keyword can make this much cleaner:

```
//: access/ImportedMyClass.java
import access.mypackage.*;

public class ImportedMyClass {
  public static void main(String[] args) {
    MyClass m = new MyClass();
  }
} ///:~
```

It's worth keeping in mind that what the **package** and **import** keywords allow you to do, as a library designer, is to divide up the single global namespace so you won't have clashing names, no matter how many people get on the Internet and start writing classes in Java.

Creating unique package names

You might observe that, since a package never really gets "packaged" into a single file, a package can be made up of many **.class** files, and things could get a bit cluttered. To prevent this, a logical thing to do is to place all the **.class** files for a particular package into a single directory; that is, use the hierarchical file structure of the operating system to your advantage. This is one way that Java references the problem of clutter; you'll see the other way later when the **jar** utility is introduced.

Collecting the package files into a single subdirectory solves two other problems: creating unique package names, and finding those classes that might be buried in a directory structure someplace. This is accomplished by encoding the path of the location of the **.class** file into the name of the **package**. By convention, the first part of the **package** name is the reversed Internet domain name of the creator of the class. Since Internet domain names are guaranteed to be unique, *if* you follow this convention, your **package** name will be unique and you'll never have a name clash. (That is, until you lose the domain name to someone else who starts writing Java code with the same path names as you did.) Of course, if you don't have your own domain name, then you must fabricate an unlikely combination (such as your first and last name) to create unique package names. If you've decided to start publishing Java code, it's worth the relatively small effort to get a domain name.

The second part of this trick is resolving the **package** name into a directory on your machine, so that when the Java program runs and it needs to load the **.class** file, it can locate the directory where the **.class** file resides.

The Java interpreter proceeds as follows. First, it finds the environment variable CLASSPATH[3] (set via the operating system, and sometimes by the installation program that installs Java or a Java-based tool on your machine). CLASSPATH contains one or more directories that are used as roots in a search for **.class** files. Starting at that root, the interpreter will take the package name and replace each dot with a slash to generate a path name off of the CLASSPATH root (so **package foo.bar.baz** becomes **foo\bar\baz** or **foo/bar/baz** or possibly something else,

[3] When referring to the environment variable, capital letters will be used (CLASSPATH).

depending on your operating system). This is then concatenated to the various entries in the CLASSPATH. That's where it looks for the **.class** file with the name corresponding to the class you're trying to create. (It also searches some standard directories relative to where the Java interpreter resides.)

To understand this, consider my domain name, which is **MindView.net**. By reversing this and making it all lowercase, **net.mindview** establishes my unique global name for my classes. (The com, edu, org, etc., extensions were formerly capitalized in Java packages, but this was changed in Java 2 so the entire package name is lowercase.) I can further subdivide this by deciding that I want to create a library named **simple**, so I'll end up with a package name:

```
package net.mindview.simple;
```

Now this package name can be used as an umbrella namespace for the following two files:

```
//: net/mindview/simple/Vector.java
// Creating a package.
package net.mindview.simple;

public class Vector {
  public Vector() {
    System.out.println("net.mindview.simple.Vector");
  }
} ///:~
```

As mentioned before, the **package** statement must be the first non-comment code in the file. The second file looks much the same:

```
//: net/mindview/simple/List.java
// Creating a package.
package net.mindview.simple;

public class List {
  public List() {
    System.out.println("net.mindview.simple.List");
  }
} ///:~
```

Both of these files are placed in the subdirectory on my system:

```
C:\DOC\JavaT\net\mindview\simple
```

(Notice that the first comment line in every file in this book establishes the directory location of that file in the source-code tree—this is used by the automatic code-extraction tool for this book.)

If you walk back through this path, you can see the package name **net.mindview.simple**, but what about the first portion of the path? That's taken care of by the CLASSPATH environment variable, which is, on my machine:

```
CLASSPATH=.;D:\JAVA\LIB;C:\DOC\JavaT
```

You can see that the CLASSPATH can contain a number of alternative search paths.

There's a variation when using JAR files, however. You must put the actual name of the JAR file in the classpath, not just the path where it's located. So for a JAR named **grape.jar** your classpath would include:

```
CLASSPATH=.;D:\JAVA\LIB;C:\flavors\grape.jar
```

Once the classpath is set up properly, the following file can be placed in any directory:

```
//: access/LibTest.java
// Uses the library.
import net.mindview.simple.*;

public class LibTest {
  public static void main(String[] args) {
    Vector v = new Vector();
    List l = new List();
  }
} /* Output:
net.mindview.simple.Vector
net.mindview.simple.List
*///:~
```

When the compiler encounters the **import** statement for the **simple** library, it begins searching at the directories specified by CLASSPATH, looking for subdirectory **net/mindview/simple**, then seeking the compiled files of the appropriate names (**Vector.class** for **Vector**, and **List.class** for **List**). Note that both the classes and the desired methods in **Vector** and **List** must be **public**.

Setting the CLASSPATH has been such a trial for beginning Java users (it was for me, when I started) that Sun made the JDK in later versions of Java a bit smarter. You'll find that when you install it, even if you don't set the CLASSPATH, you'll be able to compile and run basic Java programs. To compile and run the source-code package for this book (available at *www.MindView.net*), however, you will need to add the base directory of the book's code tree to your CLASSPATH.

Exercise 1: (1) Create a class in a package. Create an instance of your class outside of that package.

Collisions

What happens if two libraries are imported via '*****' and they include the same names? For example, suppose a program does this:

```
import net.mindview.simple.*;
import java.util.*;
```

Since **java.util.*** also contains a **Vector** class, this causes a potential collision. However, as long as you don't write the code that actually causes the collision, everything is OK—this is good, because otherwise you might end up doing a lot of typing to prevent collisions that would never happen.

The collision *does* occur if you now try to make a **Vector**:

```
Vector v = new Vector();
```

Which **Vector** class does this refer to? The compiler can't know, and the reader can't know either. So the compiler complains and forces you to be explicit. If I want the standard Java **Vector**, for example, I must say:

```
java.util.Vector v = new java.util.Vector();
```

Since this (along with the CLASSPATH) completely specifies the location of that **Vector**, there's no need for the **import java.util.*** statement unless I'm using something else from **java.util**.

Alternatively, you can use the single-class import form to prevent clashes—as long as you don't use both colliding names in the same program (in which case you must fall back to fully specifying the names).

Exercise 2: (1) Take the code fragments in this section and turn them into a program, and verify that collisions do in fact occur.

A custom tool library

With this knowledge, you can now create your own libraries of tools to reduce or eliminate duplicate code. Consider, for example, the alias we've been using for **System.out.println( )**, to reduce typing. This can be part of a class called **Print** so that you end up with a readable **static import**:

```
//: net/mindview/util/Print.java
// Print methods that can be used without
// qualifiers, using Java SE5 static imports:
package net.mindview.util;
import java.io.*;

public class Print {
  // Print with a newline:
  public static void print(Object obj) {
    System.out.println(obj);
  }
  // Print a newline by itself:
  public static void print() {
    System.out.println();
  }
  // Print with no line break:
  public static void printnb(Object obj) {
    System.out.print(obj);
  }
  // The new Java SE5 printf() (from C):
  public static PrintStream
  printf(String format, Object... args) {
    return System.out.printf(format, args);
  }
} ///:~
```

You can use the printing shorthand to print anything, either with a newline (**print( )**) or without a newline (**printnb( )**).

You can guess that the location of this file must be in a directory that starts at one of the CLASSPATH locations, then continues into **net/mindview**. After compiling, the **static print( )** and **printnb( )** methods can be used anywhere on your system with an **import static** statement:

```
//: access/PrintTest.java
// Uses the static printing methods in Print.java.
import static net.mindview.util.Print.*;

public class PrintTest {
  public static void main(String[] args) {
    print("Available from now on!");
    print(100);
    print(100L);
```

```
    print(3.14159);
  }
} /* Output:
Available from now on!
100
100
3.14159
*///:~
```

A second component of this library can be the **range( )** methods, introduced in the *Controlling Execution* chapter, that allow the use of the foreach syntax for simple integer sequences:

```
//: net/mindview/util/Range.java
// Array creation methods that can be used without
// qualifiers, using Java SE5 static imports:
package net.mindview.util;

public class Range {
  // Produce a sequence [0..n)
  public static int[] range(int n) {
    int[] result = new int[n];
    for(int i = 0; i < n; i++)
      result[i] = i;
    return result;
  }
  // Produce a sequence [start..end)
  public static int[] range(int start, int end) {
    int sz = end - start;
    int[] result = new int[sz];
    for(int i = 0; i < sz; i++)
      result[i] = start + i;
    return result;
  }
  // Produce a sequence [start..end) incrementing by step
  public static int[] range(int start, int end, int step) {
    int sz = (end - start)/step;
    int[] result = new int[sz];
    for(int i = 0; i < sz; i++)
      result[i] = start + (i * step);
    return result;
  }
} ///:~
```

From now on, whenever you come up with a useful new utility, you can add it to your own library. You'll see more components added to the **net.mindview.util** library throughout the book.

Using imports to change behavior

A feature that is missing from Java is C's *conditional compilation*, which allows you to change a switch and get different behavior without changing any other code. The reason such a feature was left out of Java is probably because it is most often used in C to solve cross-platform issues: Different portions of the code are compiled depending on the target platform. Since Java is intended to be automatically cross-platform, such a feature should not be necessary.

However, there are other valuable uses for conditional compilation. A very common use is for debugging code. The debugging features are enabled during development and disabled in the

shipping product. You can accomplish this by changing the **package** that's imported in order to change the code used in your program from the debug version to the production version. This technique can be used for any kind of conditional code.

Exercise 3: (2) Create two packages: **debug** and **debugoff**, containing an identical class with a **debug( )** method. The first version displays its **String** argument to the console, the second does nothing. Use a **static import** line to import the class into a test program, and demonstrate the conditional compilation effect.

Package caveat

It's worth remembering that anytime you create a package, you implicitly specify a directory structure when you give the package a name. The package *must* live in the directory indicated by its name, which must be a directory that is searchable starting from the CLASSPATH. Experimenting with the **package** keyword can be a bit frustrating at first, because unless you adhere to the package-name to directory-path rule, you'll get a lot of mysterious runtime messages about not being able to find a particular class, even if that class is sitting there in the same directory. If you get a message like this, try commenting out the **package** statement, and if it runs, you'll know where the problem lies.

Note that compiled code is often placed in a different directory than source code, but the path to the compiled code must still be found by the JVM using the CLASSPATH.

Java access specifiers

The Java access specifiers **public**, **protected**, and **private** are placed in front of each definition for each member in your class, whether it's a field or a method. Each access specifier only controls the access for that particular definition.

If you don't provide an access specifier, it means "package access." So one way or another, everything has some kind of access control. In the following sections, you'll learn about the various types of access.

Package access

All the examples before this chapter used no access specifiers. The default access has no keyword, but it is commonly referred to as *package access* (and sometimes "friendly"). It means that all the other classes in the current package have access to that member, but to all the classes outside of this package, the member appears to be **private**. Since a compilation unit—a file—can belong only to a single package, all the classes within a single compilation unit are automatically available to each other via package access.

Package access allows you to group related classes together in a package so that they can easily interact with each other. When you put classes together in a package, thus granting mutual access to their package-access members, you "own" the code in that package. It makes sense that only code that you own should have package access to other code that you own. You could say that package access gives a meaning or a reason for grouping classes together in a package. In many languages the way you organize your definitions in files can be arbitrary, but in Java you're compelled to organize them in a sensible fashion. In addition, you'll probably want to exclude classes that shouldn't have access to the classes being defined in the current package.

The class controls the code that has access to its members. Code from another package can't just come around and say, "Hi, I'm a friend of **Bob**'s!" and expect to be shown the **protected**, package-access, and **private** members of **Bob**. The only way to grant access to a member is to:

1. Make the member **public**. Then everybody, everywhere, can access it.
2. Give the member package access by leaving off any access specifier, and put the other classes in the same package. Then the other classes in that package can access the member.
3. As you'll see in the *Reusing Classes* chapter, when inheritance is introduced, an inherited class can access a **protected** member as well as a **public** member (but not **private** members). It can access package-access members only if the two classes are in the same package. But don't worry about inheritance and **protected** right now.
4. Provide "accessor/mutator" methods (also known as "get/set" methods) that read and change the value. This is the most civilized approach in terms of OOP, and it is fundamental to JavaBeans, as you'll see in the *Graphical User Interfaces* chapter.

public: interface access

When you use the **public** keyword, it means that the member declaration that immediately follows **public** is available to everyone, in particular to the client programmer who uses the library. Suppose you define a package **dessert** containing the following compilation unit:

```
//: access/dessert/Cookie.java
// Creates a library.
package access.dessert;

public class Cookie {
  public Cookie() {
   System.out.println("Cookie constructor");
  }
  void bite() { System.out.println("bite"); }
} ///:~
```

Remember, the class file produced by **Cookie.java** must reside in a subdirectory called **dessert**, in a directory under **access** (indicating the *Access Control* chapter of this book) that must be under one of the CLASSPATH directories. Don't make the mistake of thinking that Java will always look at the current directory as one of the starting points for searching. If you don't have a '**.**' as one of the paths in your CLASSPATH, Java won't look there.

Now if you create a program that uses **Cookie**:

```
//: access/Dinner.java
// Uses the library.
import access.dessert.*;

public class Dinner {
  public static void main(String[] args) {
    Cookie x = new Cookie();
    //! x.bite(); // Can't access
  }
} /* Output:
Cookie constructor
*///:~
```

you can create a **Cookie** object, since its constructor is **public** and the class is **public**. (We'll look more at the concept of a **public** class later.) However, the **bite( )** member is inaccessible inside **Dinner.java** since **bite( )** provides access only within package **dessert**, so the compiler prevents you from using it.

The default package

You might be surprised to discover that the following code compiles, even though it would appear that it breaks the rules:

```
//: access/Cake.java
// Accesses a class in a separate compilation unit.

class Cake {
  public static void main(String[] args) {
    Pie x = new Pie();
    x.f();
  }
} /* Output:
Pie.f()
*///:~
```

In a second file in the same directory:

```
//: access/Pie.java
// The other class.

class Pie {
  void f() { System.out.println("Pie.f()"); }
} ///:~
```

You might initially view these as completely foreign files, and yet **Cake** is able to create a **Pie** object and call its **f( )** method. (Note that you must have '.' in your CLASSPATH in order for the files to compile.) You'd typically think that **Pie** and **f( )** have package access and are therefore not available to **Cake**. They *do* have package access—that part is correct. The reason that they are available in **Cake.java** is because they are in the same directory and have no explicit package name. Java treats files like this as implicitly part of the "default package" for that directory, and thus they provide package access to all the other files in that directory.

private: you can't touch that!

The **private** keyword means that no one can access that member except the class that contains that member, inside methods of that class. Other classes in the same package cannot access **private** members, so it's as if you're even insulating the class against yourself. On the other hand, it's not unlikely that a package might be created by several people collaborating together, so **private** allows you to freely change that member without concern that it will affect another class in the same package.

The default package access often provides an adequate amount of hiding; remember, a package-access member is inaccessible to the client programmer using the class. This is nice, since the default access is the one that you normally use (and the one that you'll get if you forget to add any access control). Thus, you'll typically think about access for the members that you explicitly want to make **public** for the client programmer, and as a result, you might initially think that you won't use the **private** keyword very often, since it's tolerable to get away without it. However, it turns out that the consistent use of **private** is very important, especially where multithreading is concerned. (As you'll see in the *Concurrency* chapter.)

Here's an example of the use of **private**:

```
//: access/IceCream.java
// Demonstrates "private" keyword.
```

```
class Sundae {
  private Sundae() {}
  static Sundae makeASundae() {
    return new Sundae();
  }
}

public class IceCream {
  public static void main(String[] args) {
    //! Sundae x = new Sundae();
    Sundae x = Sundae.makeASundae();
  }
} ///:~
```

This shows an example in which **private** comes in handy: You might want to control how an object is created and prevent someone from directly accessing a particular constructor (or all of them). In the preceding example, you cannot create a **Sundae** object via its constructor; instead, you must call the **makeASundae( )** method to do it for you.[4]

Any method that you're certain is only a "helper" method for that class can be made **private**, to ensure that you don't accidentally use it elsewhere in the package and thus prohibit yourself from changing or removing the method. Making a method **private** guarantees that you retain this option.

The same is true for a **private** field inside a class. Unless you must expose the underlying implementation (which is less likely than you might think), you should make all fields **private**. However, just because a reference to an object is **private** inside a class doesn't mean that some other object can't have a **public** reference to the same object. (See the online supplements for this book to learn about aliasing issues.)

protected: inheritance access

Understanding the **protected** access specifier requires a jump ahead. First, you should be aware that you don't need to understand this section to continue through this book up through inheritance (the *Reusing Classes* chapter). But for completeness, here is a brief description and example using **protected**.

The **protected** keyword deals with a concept called *inheritance*, which takes an existing class—which we refer to as the *base class*—and adds new members to that class without touching the existing class. You can also change the behavior of existing members of the class. To inherit from a class, you say that your new class **extends** an existing class, like this:

```
class Foo extends Bar {
```

The rest of the class definition looks the same.

If you create a new package and inherit from a class in another package, the only members you have access to are the **public** members of the original package. (Of course, if you perform the inheritance in the *same* package, you can manipulate all the members that have package access.) Sometimes the creator of the base class would like to take a particular member and grant access

---

[4] There's another effect in this case: Since the default constructor is the only one defined, and it's **private**, it will prevent inheritance of this class. (A subject that will be introduced later.)

to derived classes but not the world in general. That's what **protected** does. **protected** also gives package access—that is, other classes in the same package may access **protected** elements.

If you refer back to the file **Cookie.java**, the following class *cannot* call the package-access member **bite( )**:

```
//: access/ChocolateChip.java
// Can't use package-access member from another package.
import access.dessert.*;

public class ChocolateChip extends Cookie {
  public ChocolateChip() {
   System.out.println("ChocolateChip constructor");
  }
  public void chomp() {
    //! bite(); // Can't access bite
  }
  public static void main(String[] args) {
    ChocolateChip x = new ChocolateChip();
    x.chomp();
  }
} /* Output:
Cookie constructor
ChocolateChip constructor
*///:~
```

One of the interesting things about inheritance is that if a method **bite( )** exists in class **Cookie**, then it also exists in any class inherited from **Cookie**. But since **bite( )** has package access and is in a foreign package, it's unavailable to us in this one. Of course, you could make it **public**, but then everyone would have access, and maybe that's not what you want. If you change the class **Cookie** as follows:

```
//: access/cookie2/Cookie.java
package access.cookie2;

public class Cookie {
  public Cookie() {
    System.out.println("Cookie constructor");
  }
  protected void bite() {
    System.out.println("bite");
  }
} ///:~
```

now **bite( )** becomes accessible to anyone inheriting from **Cookie**:

```
//: access/ChocolateChip2.java
import access.cookie2.*;

public class ChocolateChip2 extends Cookie {
  public ChocolateChip2() {
   System.out.println("ChocolateChip2 constructor");
  }
  public void chomp() { bite(); } // Protected method
  public static void main(String[] args) {
    ChocolateChip2 x = new ChocolateChip2();
    x.chomp();
  }
```

```
} /* Output:
Cookie constructor
ChocolateChip2 constructor
bite
*///:~
```

Note that, although **bite( )** also has package access, it is *not* **public**.

Exercise 4: (2) Show that **protected** methods have package access but are not **public**.

Exercise 5: (2) Create a class with **public**, **private**, **protected**, and package-access fields and method members. Create an object of this class and see what kind of compiler messages you get when you try to access all the class members. Be aware that classes in the same directory are part of the "default" package.

Exercise 6: (1) Create a class with **protected** data. Create a second class in the same file with a method that manipulates the **protected** data in the first class.

Interface and implementation

Access control is often referred to as *implementation hiding*. Wrapping data and methods within classes in combination with implementation hiding is often called *encapsulation*.[5] The result is a data type with characteristics and behaviors.

Access control puts boundaries within a data type for two important reasons. The first is to establish what the client programmers can and can't use. You can build your internal mechanisms into the structure without worrying that the client programmers will accidentally treat the internals as part of the interface that they should be using.

This feeds directly into the second reason, which is to separate the interface from the implementation. If the structure is used in a set of programs, but client programmers can't do anything but send messages to the **public** interface, then you are free to change anything that's *not* **public** (e.g., package access, **protected**, or **private**) without breaking client code.

For clarity, you might prefer a style of creating classes that puts the **public** members at the beginning, followed by the **protected**, package-access, and **private** members. The advantage is that the user of the class can then read down from the top and see first what's important to them (the **public** members, because they can be accessed outside the file), and stop reading when they encounter the non-**public** members, which are part of the internal implementation:

```
//: access/OrganizedByAccess.java

public class OrganizedByAccess {
  public void pub1() { /* ... */ }
  public void pub2() { /* ... */ }
  public void pub3() { /* ... */ }
  private void priv1() { /* ... */ }
  private void priv2() { /* ... */ }
  private void priv3() { /* ... */ }
  private int i;
  // ...
} ///:~
```

[5] However, people often refer to implementation hiding alone as encapsulation.

This will make it only partially easier to read, because the interface and implementation are still mixed together. That is, you still see the source code—the implementation—because it's right there in the class. In addition, the comment documentation supported by Javadoc lessens the importance of code readability by the client programmer. Displaying the interface to the consumer of a class is really the job of the *class browser*, a tool whose job is to look at all the available classes and show you what you can do with them (i.e., what members are available) in a useful fashion. In Java, viewing the JDK documentation with a Web browser gives you the same effect as a class browser.

Class access

In Java, the access specifiers can also be used to determine which classes *within* a library will be available to the users of that library. If you want a class to be available to a client programmer, you use the **public** keyword on the entire class definition. This controls whether the client programmer can even create an object of the class.

To control the access of a class, the specifier must appear before the keyword **class**. Thus you can say:

```
public class Widget {
```

Now if the name of your library is **access**, any client programmer can access **Widget** by saying

```
import access.Widget;
```

or

```
import access.*;
```

However, there's an extra set of constraints:

1. There can be only one **public** class per compilation unit (file). The idea is that each compilation unit has a single public interface represented by that **public** class. It can have as many supporting package-access classes as you want. If you have more than one **public** class inside a compilation unit, the compiler will give you an error message.

2. The name of the **public** class must exactly match the name of the file containing the compilation unit, including capitalization. So for **Widget**, the name of the file must be **Widget.java**, not **widget.java** or **WIDGET.java**. Again, you'll get a compile-time error if they don't agree.

3. It is possible, though not typical, to have a compilation unit with no **public** class at all. In this case, you can name the file whatever you like (although naming it arbitrarily will be confusing to people reading and maintaining the code).

What if you've got a class inside **access** that you're only using to accomplish the tasks performed by **Widget** or some other **public** class in **access**? You don't want to go to the bother of creating documentation for the client programmer, and you think that sometime later you might want to completely change things and rip out your class altogether, substituting a different one. To give you this flexibility, you need to ensure that no client programmers become dependent on your particular implementation details hidden inside **access**. To accomplish this, you just leave the **public** keyword off the class, in which case it has package access. (That class can be used only within that package.)

Exercise 7: (1) Create the library according to the code fragments describing **access** and **Widget**. Create a **Widget** in a class that is not part of the **access** package.

When you create a package-access class, it still makes sense to make the fields of the class **private**—you should always make fields as **private** as possible—but it's generally reasonable to give the methods the same access as the class (package access). Since a package-access class is usually used only within the package, you only need to make the methods of such a class **public** if you're forced to, and in those cases, the compiler will tell you.

Note that a class cannot be **private** (that would make it inaccessible to anyone but the class) or **protected**.[6] So you have only two choices for class access: package access or **public**. If you don't want anyone else to have access to that class, you can make all the constructors **private**, thereby preventing anyone but you, inside a **static** member of the class, from creating an object of that class. Here's an example:

```
//: access/Lunch.java
// Demonstrates class access specifiers. Make a class
// effectively private with private constructors:

class Soup1 {
  private Soup1() {}
  // (1) Allow creation via static method:
  public static Soup1 makeSoup() {
    return new Soup1();
  }
}

class Soup2 {
  private Soup2() {}
  // (2) Create a static object and return a reference
  // upon request.(The "Singleton" pattern):
  private static Soup2 ps1 = new Soup2();
  public static Soup2 access() {
    return ps1;
  }
  public void f() {}
}

// Only one public class allowed per file:
public class Lunch {
  void testPrivate() {
    // Can't do this! Private constructor:
    //! Soup1 soup = new Soup1();
  }
  void testStatic() {
    Soup1 soup = Soup1.makeSoup();
  }
  void testSingleton() {
    Soup2.access().f();
  }
} ///:~
```

[6] Actually, an *inner class* can be private or protected, but that's a special case. These will be introduced in the *Inner Classes* chapter.

Up to now, most of the methods have been returning either **void** or a primitive type, so the definition:

```
public static Soup1 makeSoup() {
  return new Soup1();
}
```

might look a little confusing at first. The word **Soup1** before the method name (**makeSoup**) tells what the method returns. So far in this book, this has usually been **void**, which means it returns nothing. But you can also return a reference to an object, which is what happens here. This method returns a reference to an object of class **Soup1**.

The classes **Soup1** and **Soup2** show how to prevent direct creation of a class by making all the constructors **private**. Remember that if you don't explicitly create at least one constructor, the default constructor (a constructor with no arguments) will be created for you. By writing the default constructor, it won't be created automatically. By making it **private**, no one can create an object of that class. But now how does anyone use this class? The preceding example shows two options. In **Soup1**, a **static** method is created that creates a new **Soup1** and returns a reference to it. This can be useful if you want to do some extra operations on the **Soup1** before returning it, or if you want to keep count of how many **Soup1** objects to create (perhaps to restrict their population).

Soup2 uses what's called a *design pattern*, which is covered in *Thinking in Patterns (with Java)* at *www.MindView.net*. This particular pattern is called a *Singleton*, because it allows only a single object to ever be created. The object of class **Soup2** is created as a **static private** member of **Soup2**, so there's one and only one, and you can't get at it except through the **public** method **access( )**.

As previously mentioned, if you don't put an access specifier for class access, it defaults to package access. This means that an object of that class can be created by any other class in the package, but not outside the package. (Remember, all the files within the same directory that don't have explicit **package** declarations are implicitly part of the default package for that directory.) However, if a **static** member of that class is **public**, the client programmer can still access that **static** member even though they cannot create an object of that class.

Exercise 8: (4) Following the form of the example **Lunch.java**, create a class called **ConnectionManager** that manages a fixed array of **Connection** objects. The client programmer must not be able to explicitly create **Connection** objects, but can only get them via a **static** method in **ConnectionManager**. When the **ConnectionManager** runs out of objects, it returns a **null** reference. Test the classes in **main( )**.

Exercise 9: (2) Create the following file in the **access/local** directory (presumably in your CLASSPATH):

```
// access/local/PackagedClass.java
package access.local;

class PackagedClass {
  public PackagedClass() {
    System.out.println("Creating a packaged class");
  }
}
```

Then create the following file in a directory other than **access/local**:

```
// access/foreign/Foreign.java
```

```
package access.foreign;
import access.local.*;

public class Foreign {
   public static void main(String[] args) {
      PackagedClass pc = new PackagedClass();
   }
}
```

Explain why the compiler generates an error. Would making the **Foreign** class part of the **access.local** package change anything?

Summary

In any relationship it's important to have boundaries that are respected by all parties involved. When you create a library, you establish a relationship with the user of that library—the client programmer—who is another programmer, but one using your library to build an application or a bigger library.

Without rules, client programmers can do anything they want with all the members of a class, even if you might prefer they don't directly manipulate some of the members. Everything's naked to the world.

This chapter looked at how classes are built to form libraries: first, the way a group of classes is packaged within a library, and second, the way the class controls access to its members.

It is estimated that a C programming project begins to break down somewhere between 50K and 100K lines of code because C has a single namespace, and names begin to collide, causing extra management overhead. In Java, the **package** keyword, the package naming scheme, and the **import** keyword give you complete control over names, so the issue of name collision is easily avoided.

There are two reasons for controlling access to members. The first is to keep users' hands off portions that they shouldn't touch. These pieces are necessary for the internal operations of the class, but not part of the interface that the client programmer needs. So making methods and fields **private** is a service to client programmers, because they can easily see what's important to them and what they can ignore. It simplifies their understanding of the class.

The second and most important reason for access control is to allow the library designer to change the internal workings of the class without worrying about how it will affect the client programmer. You might, for example, build a class one way at first, and then discover that restructuring your code will provide much greater speed. If the interface and implementation are clearly separated and protected, you can accomplish this without forcing client programmers to rewrite their code. Access control ensures that no client programmer becomes dependent on any part of the underlying implementation of a class.

When you have the ability to change the underlying implementation, you not only have the freedom to improve your design, you also have the freedom to make mistakes. No matter how carefully you plan and design, you'll make mistakes. Knowing that it's relatively safe to make these mistakes means you'll be more experimental, you'll learn more quickly, and you'll finish your project sooner.

The public interface to a class is what the user *does* see, so that is the most important part of the class to get "right" during analysis and design. Even that allows you some leeway for change. If

you don't get the interface right the first time, you can *add* more methods, as long as you don't remove any that client programmers have already used in their code.

Notice that access control focuses on a relationship—and a kind of communication—between a library creator and the external clients of that library. There are many situations where this is not the case. For example, you are writing all the code yourself, or you are working in close quarters with a small team and everything goes into the same package. These situations have a different kind of communication, and rigid adherence to access rules may not be optimal. Default (package) access may be just fine.

Solutions to selected exercises can be found in the electronic document *The Thinking in Java Annotated Solution Guide*, available for sale from *www.MindView.net*.

A: Supplements

There are a number of supplements to this book, including the items, seminars, and services available through the MindView Web site.

This appendix describes these supplements so that you can decide if they will be helpful to you.

Note that although the seminars are often held as public events, they may be given as private, in-house seminars at your location.

Downloadable supplements

The code for this book is available for download from *www.MindView.net*. This includes the Ant build files and other support files necessary to do a successful build and execution of all the examples in the book.

In addition, a few portions of the book were moved to electronic form. The subjects include:

- Cloning Objects
- Passing & Returning Objects
- Analysis and Design
- Portions of other chapters from *Thinking in Java, 3rd Edition* that were not relevant enough to put in the print version of the 4th edition of this book.

Thinking in C: Foundations for Java

At *www.MindView.net*, you will find the *Thinking in C* seminar as a free download. This presentation, created by Chuck Allison and developed by MindView, is a multimedia Flash course which gives you an introduction to the C syntax, operators and functions that Java syntax is based upon.

Note that you must have the Flash Player from *www.Macromedia.com* installed on your system in order to play *Thinking in C*.

Thinking in Java seminar

My company, MindView, Inc., provides five-day, hands-on, public and in-house training seminars based on the material in this book. Formerly called the *Hands-On Java* seminar, this is our main introductory seminar that provides the foundation for our more advanced seminars. Selected material from each chapter represents a lesson, which is followed by a monitored exercise period so that each student receives personal attention. You can find schedule and location information, testimonials, and details at *www.MindView.net*.

Hands-On Java seminar-on-CD

The *Hands-On Java CD* contains an extended version of the material from the *Thinking in Java* seminar and is based on this book. It provides at least some of the experience of the live seminar without the travel and expense. There is an audio lecture and slides corresponding to every

chapter in the book. I created the seminar and I narrate the material on the CD. The material is in Flash format, so it should run on any platform that supports the Flash Player. The *Hands-On Java CD* is for sale at *www.MindView.net*, where you can find trial demos of the product.

Thinking in Objects seminar

This seminar introduces the ideas of object-oriented programming from the standpoint of the designer. It explores the process of developing and building a system, primarily focusing on so-called "Agile Methods" or "Lightweight Methodologies," especially Extreme Programming (XP). I introduce methodologies in general, small tools like the "index-card" planning techniques described in *Planning Extreme Programming* by Beck and Fowler (Addison-Wesley, 2001), CRC cards for object design, pair programming, iteration planning, unit testing, automated building, source-code control, and similar topics. The course includes an XP project that will be developed throughout the week.

If you are starting a project and would like to begin using object-oriented design techniques, we can use your project as the example and produce a first-cut design by the end of the week.

Visit *www.MindView.net* for schedule and location information, testimonials, and details.

Thinking in Enterprise Java

This book has been spawned from some of the more advanced chapters in earlier editions of *Thinking in Java*. This book isn't a second volume of *Thinking in Java*, but rather focused coverage of the more advanced topic of enterprise programming. It is currently available (in some form, likely still in development) as a free download from *www.MindView.net*. Because it is a separate book, it can expand to fit the necessary topics. The goal, like *Thinking in Java*, is to produce a very understandable introduction to the basics of the enterprise programming technologies so that the reader is prepared for more advanced coverage of those topics.

The list of topics will include, but is not limited to:

- Introduction to Enterprise Programming
- Network Programming with Sockets and Channels
- Remote Method Invocation (RMI)
- Connecting to Databases
- Naming and Directory Services
- Servlets
- Java Server Pages
- Tags, JSP Fragments and Expression Language
- Automating the Creation of User Interfaces
- Enterprise JavaBeans
- XML
- Web Services
- Automated Testing

You can find the current state of *Thinking in Enterprise Java* at *www.MindView.net*.

Thinking in Patterns (with Java)

One of the most important steps forward in object-oriented design is the "design patterns" movement, chronicled in *Design Patterns*, by Gamma, Helm, Johnson & Vlissides (Addison-Wesley, 1995). That book shows 23 general classes of problems and their solutions, primarily

written in C++. The *Design Patterns* book is a source of what has now become an essential, almost mandatory, vocabulary for OOP programmers. *Thinking in Patterns* introduces the basic concepts of design patterns along with examples in Java. The book is not intended to be a simple translation of *Design Patterns*, but rather a new perspective with a Java mindset. It is not limited to the traditional 23 patterns, but also includes other ideas and problem-solving techniques as appropriate.

This book began as the last chapter in *Thinking in Java*, *1st Edition*, and as ideas continued to develop, it became clear that it needed to be its own book. At the time of this writing, it is still in development, but the material has been worked and reworked through numerous presentations of the *Objects & Patterns* seminar (which has now been split into the *Designing Objects & Systems* and *Thinking in Patterns* seminars).

You can find out more about this book at *www.MindView.net*.

Thinking in Patterns seminar

This seminar has evolved from the *Objects & Patterns* seminar that Bill Venners and I gave for the past several years. That seminar grew too full, so we've split it into two seminars: this one, and the *Designing Objects & Systems* seminar, described earlier in this appendix.

The seminar strongly follows the material and presentation in the *Thinking in Patterns* book, so the best way to find out what's in the seminar is to learn about the book from *www.MindView.net*.

Much of the presentation emphasizes the design evolution process, starting with an initial solution and moving through the logic and process of evolving the solution to more appropriate designs. The last project shown (a trash recycling simulation) has evolved over time, and you can look at that evolution as a prototype for the way your own design can start as an adequate solution to a particular problem and evolve into a flexible approach to a class of problems.

This seminar will help you:

- Dramatically increase the flexibility of your designs.
- Build in extensibility and reusability.
- Create denser communications about designs using the language of patterns.

Following each lecture there will be a set of patterns exercises for you to solve, where you are guided to write code to apply particular patterns to the solution of programming problems.

Visit *www.MindView.net* for schedule and location information, testimonials, and details.

Design consulting and reviews

My company also provides consulting, mentoring, design reviews and implementation reviews to help guide your project through its development cycle, including your company's first Java project. Visit *www.MindView.net* for availability and details.

B: Resources

Software

The JDK from *http://java.sun.com*. Even if you choose to use a third-party development environment, it's always a good idea to have the JDK on hand in case you come up against what might be a compiler error. The JDK is the touchstone, and if there is a bug in it, chances are it will be well known.

The JDK documentation from *http://java.sun.com*, in HTML. I have never found a reference book on the standard Java libraries that wasn't out of date or missing information. Although the JDK documentation from Sun is shot through with small bugs and is sometimes unusably terse, all the classes and methods are at least *there*. Sometimes people are initially uncomfortable using an online resource rather than a printed book, but it's worth your while to get over this and open the HTML docs so you can at least get the big picture. If you can't figure it out at that point, then reach for the printed books.

Editors & IDEs

There is a healthy competition in this arena. Many offerings are free (and the non-free ones usually have free trials), so your best bet is to simply try them out yourself and see which one fits your needs. Here are a few:

JEdit, Slava Pestov's free editor, is written in Java, so you get the bonus of seeing a desktop Java application in action. This editor is based heavily on plug-ins, many of which have been written by the active community. Download from *www.jedit.org*.

NetBeans, a free IDE from Sun, at *www.netbeans.org*. Designed for drag-and-drop GUI building, code editing, debugging, and more.

Eclipse, an open-source project backed by IBM, among others. The Eclipse platform is also designed to be an extensible foundation, so you can build your own standalone applications on top of Eclipse. This project created the SWT described in the *Graphical User Interfaces* chapter. Download from *www.Eclipse.org*.

IntelliJ IDEA, the payware favorite of a large faction of Java programmers, many of whom claim that IDEA is always a step or two ahead of Eclipse, possibly because IntelliJ is not creating both an IDE and a development platform, but just sticking to creating an IDE. You can download a free trial from *www.jetbrains.com*.

Books

Core Java™ 2, 7th Edition, Volumes I & II, by Horstmann & Cornell (Prentice Hall, 2005). Huge, comprehensive, and the first place I go when I'm hunting for answers. The book I recommend when you've completed *Thinking in Java* and need to cast a bigger net.

The Java™ Class Libraries: An Annotated Reference, by Patrick Chan and Rosanna Lee (Addison-Wesley, 1997). Although sadly out of date, this is what the JDK reference *should* have been: enough description to make it usable. One of the technical reviewers for *Thinking in Java* said, "If I had only one Java book, this would be it (well, in addition to yours, of course)." I'm not

as thrilled with it as he is. It's big, it's expensive, and the quality of the examples doesn't satisfy me. *But* it's a place to look when you're stuck, and it seems to have more depth (and sheer size) than most alternatives. However, *Core Java 2* has more recent coverage of many of the library components.

Java Network Programming, 2nd Edition, by Elliotte Rusty Harold (O'Reilly, 2000). I didn't begin to understand Java networking (or networking in general, for that matter) until I found this book. I also find his Web site, Café au Lait, to be a stimulating, opinionated, and up-to-date perspective on Java developments, unencumbered by allegiances to any vendors. His regular updates keep up with fast-changing news about Java. See *www.cafeaulait.org*.

Design Patterns, by Gamma, Helm, Johnson and Vlissides (Addison-Wesley, 1995). The seminal book that started the patterns movement in programming, mentioned numerous places in this book.

Refactoring to Patterns, by Joshua Kerievsky (Addison-Wesley, 2005). Marries refactoring and design patterns. The most valuable thing about this book is that it shows you how to evolve a design by folding in patterns as they are needed.

The Art of UNIX Programming, by Eric Raymond (Addison-Wesley, 2004). Although Java is a cross-platform language, the prevalence of Java on the server has made knowledge of Unix/Linux important. Eric's book is an excellent introduction to the history and philosophy of this operating system, and is a fascinating read if you just want to understand some of the roots of computing.

Analysis & design

Extreme Programming Explained, 2nd Edition, by Kent Beck with Cynthia Andres. (Addison-Wesley, 2005). I've always felt that there might be a much different, much better program development process, and I think XP comes pretty darn close. The only book that has had a similar impact on me was *Peopleware* (described later), which talks primarily about the environment and dealing with corporate culture. *Extreme Programming Explained* talks about programming and turns most things, even recent "findings," on their ear. They even go so far as to say that pictures are OK as long as you don't spend too much time on them and are willing to throw them away. (You'll notice that the book does *not* have the "UML stamp of approval" on its cover.) I could see deciding to work for a company based solely on whether they used XP. Small book, small chapters, effortless to read, exciting to think about. You start imagining yourself working in such an atmosphere, and it brings visions of a whole new world.

UML Distilled, 2nd Edition, by Martin Fowler (Addison-Wesley, 2000). When you first encounter UML, it is daunting because there are so many diagrams and details. According to Fowler, most of this stuff is unnecessary, so he cuts through to the essentials. For most projects, you only need to know a few diagramming tools, and Fowler's goal is to come up with a good design rather than worry about all the artifacts of getting there. In fact, the first half of the book is all that most people will need. A nice, thin, readable book; the first one you should get if you need to understand UML.

Domain-Driven Design, by Eric Evans (Addison-Wesley, 2004). This book focuses on the *domain model* as the primary artifact of the design process. I have found this to be an important shift in emphasis that helps keep designers at the right level of abstraction.

The Unified Software Development Process, by Ivar Jacobsen, Grady Booch, and James Rumbaugh (Addison-Wesley, 1999). I went in fully prepared to dislike this book. It seemed to have all the makings of a boring college text. I was pleasantly surprised—although there are a few

parts that have explanations that seem as if those concepts aren't clear to the authors. The bulk of the book is not only clear, but enjoyable. And best of all, the process makes a lot of practical sense. It's not Extreme Programming (and does not have their clarity about testing), but it's also part of the UML juggernaut; even if you can't get XP through the door, most people have climbed aboard the "UML is good" bandwagon (regardless of their *actual* level of experience with it), so you can probably get it adopted. I think this book should be the flagship of UML, and the one you can read after Fowler's *UML Distilled* when you want more detail.

Before you choose any method, it's helpful to gain perspective from those who are not trying to sell you one. It's easy to adopt a method without really understanding what you want out of it or what it will do for you. Others are using it, which seems a compelling reason. However, humans have a strange little psychological quirk: If they want to believe something will solve their problems, they'll try it. (This is experimentation, which is good.) But if it doesn't solve their problems, they may redouble their efforts and begin to announce loudly what a great thing they've discovered. (This is denial, which is not good.) The assumption here may be that if you can get other people in the same boat, you won't be lonely, even if it's going nowhere (or sinking).

This is not to suggest that all methodologies go nowhere, but that you should be armed to the teeth with mental tools that help you stay in experimentation mode ("It's not working; let's try something else") and out of denial mode ("No, that's not really a problem. Everything's wonderful, we don't need to change"). I think the following books, read *before* you choose a method, will provide you with these tools.

Software Creativity, by Robert L. Glass (Prentice Hall, 1995). This is the best book I've seen that discusses *perspective* on the whole methodology issue. It's a collection of short essays and papers that Glass has written and sometimes acquired (P.J. Plauger is one contributor), reflecting his many years of thinking and study on the subject. They're entertaining and only long enough to say what's necessary; he doesn't ramble and bore you. He's not just blowing smoke, either; there are hundreds of references to other papers and studies. All programmers and managers should read this book before wading into the methodology mire.

Software Runaways: Monumental Software Disasters, by Robert L. Glass (Prentice Hall, 1998). The great thing about this book is that it brings to the forefront what we don't talk about: the number of projects that not only fail, but fail spectacularly. I find that most of us still think, "That can't happen to me" (or "That can't happen *again*"), and I think this puts us at a disadvantage. By keeping in mind that things can always go wrong, you're in a much better position to make them go right.

Peopleware, 2nd Edition, by Tom DeMarco and Timothy Lister (Dorset House, 1999). You *must* read this book. It's not only fun, it rocks your world and destroys your assumptions. Although DeMarco and Lister have backgrounds in software development, this book is about projects and teams in general. But the focus is on the *people* and their needs, rather than the technology and its needs. They talk about creating an environment where people will be happy and productive, rather than deciding what rules those people should follow to be adequate components of a machine. This latter attitude, I think, is the biggest contributor to programmers smiling and nodding when XYZ method is adopted, and then quietly doing whatever they've always done.

Secrets of Consulting: A Guide to Giving & Getting Advice Successfully, by Gerald M. Weinberg (Dorset House, 1985). A superb book, one of my all-time favorites. It's perfect if you are trying to be a consultant *or* if you're using consultants and trying to do a better job. Short chapters, filled with stories and anecdotes that teach you how to get to the core of the issue with

minimal struggle. Also see *More Secrets of Consulting*, published in 2002, or most any other Weinberg book.

Complexity, by M. Mitchell Waldrop (Simon & Schuster, 1992). This chronicles the coming together in Santa Fe, New Mexico, of a group of scientists from different disciplines to discuss real problems that their individual disciplines couldn't solve (the stock market in economics, the initial formation of life in biology, why people do what they do in sociology, etc.). By crossing physics, economics, chemistry, math, computer science, sociology, and others, a multidisciplinary approach to these problems is developing. But more important, a different way of *thinking* about these ultra-complex problems is emerging: away from mathematical determinism and the illusion that you can write an equation that predicts all behavior, and toward first *observing* and looking for a pattern and trying to emulate that pattern by any means possible. (The book chronicles, for example, the emergence of genetic algorithms.) This kind of thinking, I believe, is useful as we observe ways to manage more and more complex software projects.

Python

Learning Python, 2nd Edition, by Mark Lutz and David Ascher (O'Reilly, 2003). A nice programmer's introduction to my favorite language, an excellent companion to Java. The book includes an introduction to Jython, which allows you to combine Java and Python in a single program (the Jython interpreter is compiled to pure Java bytecodes, so there is nothing special you need to add to accomplish this). This language union promises great possibilities.

My own list of books

Not all of these are currently available, but some can be found through used-book outlets.

Computer Interfacing with Pascal & C (self-published under the Eisys imprint, 1988. Available for sale only from *www.MindView.net*). An introduction to electronics from back when CP/M was still king and DOS was an upstart. I used high-level languages and often the parallel port of the computer to drive various electronic projects. Adapted from my columns in the first and best magazine I wrote for, *Micro Cornucopia*. Alas, Micro C was lost long before the Internet appeared. Creating this book was an extremely satisfying publishing experience.

Using C++ (Osborne/McGraw-Hill, 1989). One of the first books out on C++. This is out of print and replaced by its 2nd edition, the renamed *C++ Inside & Out*.

C++ Inside & Out (Osborne/McGraw-Hill, 1993). As noted, actually the 2nd edition of *Using C++*. The C++ in this book is reasonably accurate, but it's circa 1992 and *Thinking in C++* is intended to replace it. You can find out more about this book and download the source code at *www.MindView.net*.

Thinking in C++, 1st Edition (Prentice Hall, 1995). This won the *Software Development Magazine* Jolt Award for best book of the year.

Thinking in C++, 2nd Edition, Volume 1 (Prentice Hall, 2000). Downloadable from *www.MindView.net*. Updated to follow the finalized language standard.

Thinking in C++, 2nd Edition, Volume 2, coauthored with Chuck Allison (Prentice Hall, 2003). Downloadable from *www.MindView.net*.

Black Belt C++: The Master's Collection, Bruce Eckel, editor (M&T Books, 1994). Out of print. A collection of chapters by various C++ luminaries based on their presentations in the C++

track at the Software Development Conference, which I chaired. The cover on this book stimulated me to gain control over all future cover designs.

Thinking in Java, 1st Edition (Prentice Hall, 1998). The 1st edition of this book won the *Software Development Magazine* Productivity Award, the *Java Developer's Journal* Editor's Choice Award, and the *JavaWorld* Reader's Choice Award for best book. Downloadable from *www.MindView.net*.

Thinking in Java, 2nd Edition (Prentice Hall, 2000). This edition won the *JavaWorld* Editor's Choice Award for best book. Downloadable from *www.MindView.net*.

Thinking in Java, 3rd Edition, (Prentice Hall, 2003). This edition won the *Software Development Magazine* Jolt Award for best book of the year, along with other awards listed on the back cover. Downloadable from *www.MindView.net*.

Index

Please note that some names will be duplicated in capitalized form. Following Java style, the capitalized names refer to Java classes, while lowercase names refer to a general concept.

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z