BRUCE ECKEL

THINKING IN

JAVA

4TH EDITION

The definitive introduction to object-oriented programming
in the language of the world wide web

FOR
JAVA
SE5/6

CODE & SUPPLEMENTS AT WWW.MINDVIEW.NET

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Thinking
in
Java

Fourth Edition

Bruce Eckel
President, MindView, Inc.

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.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include custom covers and/or content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the U.S., please contact:

International Sales
international@pearsoned.com

Visit us on the Web: www.informit.com

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

Dedication

To Dawn

Overview

What's Inside

Interfaces 311

Inner Classes 345

Holding Your Objects 389

Error Handling
with Exceptions 443

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.

---

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

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*.

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

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.

Permission to use, copy, modify, and distribute this
computer source code (Source Code) and its documentation
without fee and without a written agreement for the
purposes set forth below is hereby granted, provided that
the above copyright notice, this paragraph and the
following five numbered paragraphs appear in all copies.

1. Permission is granted to compile the Source Code and to
include the compiled code, in executable format only, in
personal and commercial software programs.

2. Permission is granted to use the Source Code without
modification in classroom situations, including in
presentation materials, provided that the book "Thinking in
Java" is cited as the origin.

3. Permission to incorporate the Source Code into printed
media may be obtained by contacting:

MindView, Inc. 5343 Valle Vista La Mesa, California 91941
Wayne@MindView.net

4. The Source Code and documentation are copyrighted by
MindView, Inc. The Source code is provided without express
or implied warranty of any kind, including any implied
warranty of merchantability, fitness for a particular
purpose or non-infringement. MindView, Inc. does not
warrant that the operation of any program that includes the
Source Code will be uninterrupted or error-free. MindView,
Inc. makes no representation about the suitability of the
Source Code or of any software that includes the Source
Code for any purpose. The entire risk as to the quality
and performance of any program that includes the Source
Code is with the user of the Source Code. The user
```

understands that the Source Code was developed for research
and instructional purposes and is advised not to rely
exclusively for any reason on the Source Code or any
program that includes the Source Code. Should the Source
Code or any resulting software prove defective, the user
assumes the cost of all necessary servicing, repair, or
correction.

5. IN NO EVENT SHALL MINDVIEW, INC., OR ITS PUBLISHER BE
LIABLE TO ANY PARTY UNDER ANY LEGAL THEORY FOR DIRECT,
INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES,
INCLUDING LOST PROFITS, BUSINESS INTERRUPTION, LOSS OF
BUSINESS INFORMATION, OR ANY OTHER PECUNIARY LOSS, OR FOR
PERSONAL INJURIES, ARISING OUT OF THE USE OF THIS SOURCE
CODE AND ITS DOCUMENTATION, OR ARISING OUT OF THE INABILITY
TO USE ANY RESULTING PROGRAM, EVEN IF MINDVIEW, INC., OR
ITS PUBLISHER HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH
DAMAGE. MINDVIEW, INC. SPECIFICALLY DISCLAIMS ANY
WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE. THE SOURCE CODE AND DOCUMENTATION PROVIDED
HEREUNDER IS ON AN "AS IS" BASIS, WITHOUT ANY ACCOMPANYING
SERVICES FROM MINDVIEW, INC., AND MINDVIEW, INC. HAS NO
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.

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:

```java
//: 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.

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,

---

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

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:

```java
//: 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:

```java
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

---

[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."

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

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

booleans, 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:

```java
//: 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
1111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
1111111111111111111111
*///:~
```

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:
   1111111111111111111111111111111
maxneg, int: -2147483648, binary:
   10000000000000000000000000000000
i, int: -1172028779, binary:
   10111010001001000100001010010101
~i, int: 1172028778, binary:
   1000101110110111011110101101010
-i, int: 1172028779, binary:
   1000101110110111011110101101011
j, int: 1717241110, binary:
   1100110010110110000010100010110
i & j, int: 570425364, binary:
   100010000000000000000000010100
i | j, int: -25213033, binary:
   11111110011111110100011110010111
i ^ j, int: -595638397, binary:
   11011100011111110100011110000011
i << 5, int: 1149784736, binary:
   1000100100010000101001010100000
i >> 5, int: -36625900, binary:
   11111101110100010010001000010100
```

```
(~i) >> 5, int: 36625899, binary:
    10001011101101110111101011
i >>> 5, int: 97591828, binary:
    101110100010010001000010100
(~i) >>> 5, int: 36625899, binary:
    10001011101101110111101011
...
*///:~
```

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 '**012**' 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*.

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.

!

! · 105
!= · 103

&

& · 111
&& · 105
&= · 111

.

.NET · 57
.new syntax · 350
.this syntax · 350

@

@ symbol, for annotations · 1059
@author · 86
@Deprecated, annotation · 1060
@deprecated, Javadoc tag · 87
@docRoot · 85
@inheritDoc · 85
@interface, and extends keyword · 1070
@link · 85
@Override · 1059
@param · 86
@Retention · 1061
@return · 86
@see · 85
@since · 86

@SuppressWarnings · 1060
@Target · 1061
@Test · 1060
@Test, for @Unit · 1084
@TestObjectCleanup, @Unit tag · 1092
@TestObjectCreate, for @Unit · 1089
@throws · 87
@Unit · 1084; using · 1084
@version · 85

[

[ ], indexing operator · 193

^

^ · 111
^= · 111

|

| · 111
|| · 105
|= · 111

+

+ · 101; String conversion with operator + · 95, 118, 504

<

< · 103
<< · 112
<<= · 112
<= · 103

=

== · 103

>

> · 103
>= · 103
>> · 112
>>= · 112

A

abstract: class · 311; inheriting from
    abstract classes · 312; keyword · 312; vs.
    interface · 328
Abstract Window Toolkit (AWT) · 1303
AbstractButton · 1333
abstraction · 24
AbstractSequentialList · 859
AbstractSet · 793
access: class · 229; control · 210, 234;
    control, violating with reflection · 607;
    inner classes & access rights · 348;
    package access and friendly · 221;
    specifiers · 31, 210, 221; within a
    directory, via the default package · 223
action command · 1358
ActionEvent · 1358, 1406
ActionListener · 1316
ActionScript, for Macromedia Flex · 1417
active objects, in concurrency · 1295
Adapter design pattern · 325, 334, 434,
    630, 733, 737, 795
Adapter Method idiom · 434
adapters, listener · 1328
add( ), ArrayList · 390
addActionListener( ) · 1403, 1410
addChangeListener · 1363
addition · 98

addListener · 1321
Adler32 · 975
agent-based programming · 1299
aggregate array initialization · 193
aggregation · 32
aliasing · 97; and String · 504; arrays · 194
Allison, Chuck · 4, 18, 1449, 1460
allocate( ) · 948
allocateDirect( ) · 948
alphabetic sorting · 418
alphabetic vs. lexicographic sorting · 783
AND: bitwise · 120; logical (&&) · 105
annotation · 1059; apt processing tool ·
    1074; default element values · 1062,
    1063, 1065; default value · 1069;
    elements · 1061; elements, allowed types
    for · 1065; marker annotation · 1061;
    processor · 1064; processor based on
    reflection · 1071
anonymous inner class · 356, 904, 1314;
    and table-driven code · 859; generic ·
    645
application: builder · 1394; framework ·
    375
applying a method to a sequence · 728
apt, annotation processing tool · 1074
argument: constructor · 156; covariant
    argument types · 706; final · 266, 904;
    generic type argument inference · 632;
    variable argument lists (unknown
    quantity and type of arguments) · 198
Arnold, Ken · 1306
array: array of generic objects · 850;
    associative array · 394; bounds checking
    · 194; comparing arrays · 777;
    comparison with container · 748;
    copying an array · 775; covariance · 677;
    dynamic aggregate initialization syntax ·
    752; element comparisons · 778; first-
    class objects · 749; initialization · 193;
    length · 194, 749; multidimensional ·
    754; not Iterable · 433; of objects · 749;
    of primitives · 749; ragged · 755;
    returning an array · 753
ArrayBlockingQueue · 1215
ArrayList · 401, 817; add( ) · 390; get( ) ·
    390; size( ) · 390
Arrays: asList( ) · 396, 436, 816;
    binarySearch( ) · 784; class, container
    utility · 775
asCharBuffer( ) · 950
aspect-oriented programming (AOP) · 714

BufferedWriter · 924, 930
busy wait, concurrency · 1198
button: creating your own · 1329; radio button · 1344; Swing · 1311, 1333
ButtonGroup · 1334, 1344
ByteArrayInputStream · 916
ByteArrayOutputStream · 917
ByteBuffer · 946
bytecode engineering · 1101; Javassist · 1104

C

C#: programming language · 57
C++ · 103; exception handling · 492; Standard Template Library (STL) · 900; templates · 618, 652
CachedThreadPool · 1121
Callable, concurrency · 1124
callback · 903, 1312; and inner classes · 372
camel-casing · 88
capacity, of a HashMap or HashSet · 878
capitalization of package names · 75
Cascading Style Sheets (CSS), and Macromedia Flex · 1423
case statement · 151
CASE_INSENSITIVE_ORDER String Comparator · 884, 902
cast · 42; and generic types · 697; and primitive types · 133; asSubclass( ) · 569; operators · 120; via a generic class · 699
cast( ) · 568
catch: catching an exception · 447; catching any exception · 458; keyword · 448
Chain of Responsibility design pattern · 1036
chained exceptions · 464, 498
change, vector of · 377
channel, nio · 946
CharArrayReader · 923
CharArrayWriter · 923
CharBuffer · 950
CharSequence · 530
Charset · 952
check box · 1342
checked exceptions · 457, 491; converting to unchecked exceptions · 497
checkedCollection( ) · 710

CheckedInputStream · 973
checkedList( ) · 710
checkedMap( ) · 710
CheckedOutputStream · 973
checkedSet( ) · 710
checkedSortedMap( ) · 710
checkedSortedSet( ) · 710
Checksum class · 975
Chiba, Shigeru, Dr. · 1104, 1106
class · 27; abstract class · 311; access · 229; anonymous inner class · 356, 904, 1314; base class · 226, 241, 281; browser · 229; class hierarchies and exception handling · 489; class literal · 562, 576; creators · 30; data · 76; derived class · 281; equivalence, and instanceof/isInstance( ) · 586; final classes · 270; inheritance diagrams · 261; inheriting from abstract classes · 312; inheriting from inner classes · 382; initialization · 563; initialization & class loading · 272; initialization of fields · 182; initializing the base class · 244; initializing the derived class · 244; inner class · 345; inner class, and access rights · 348; inner class, and overriding · 383; inner class, and super · 383; inner class, and Swing · 1322; inner class, and upcasting · 352; inner class, identifiers and .class files · 387; inner class, in methods and scopes · 354; inner class, nesting within any arbitrary scope · 355; instance of · 25; keyword · 33; linking · 563; loading · 273, 563; member initialization · 239; methods · 76; multiply nested · 368; nested class (static inner class) · 364; nesting inside an interface · 366; order of initialization · 185; private inner classes · 377; public class, and compilation units · 211; referring to the outer-class object in an inner class · 350; static inner classes · 364; style of creating classes · 228; subobject · 244
Class · 1335; Class object · 556, 998, 1156; forName( ) · 558, 1326; getCanonicalName( ) · 560; getClass( ) · 459; getConstructors( ) · 592; getInterfaces( ) · 560; getMethods( ) · 592; getSimpleName( ) · 560; getSuperclass( ) · 561; isAssignableFrom( ) · 580; isInstance( )

· 578; isInterface( ) · 560;
newInstance( ) · 561; object creation
process · 189; references, and bounds ·
566; references, and generics · 565;
references, and wildcards · 566; RTTI
using the Class object · 556
class files, analyzing · 1101
class loader · 556
class name, discovering from class file ·
1101
ClassCastException · 309, 570
ClassNotFoundException · 574
classpath · 214
cleanup: and garbage collector · 251;
performing · 175; verifying the
termination condition with finalize( ) ·
176; with finally · 473
clear( ), nio · 949
client programmer · 30; vs. library creator
· 209
close( ) · 928
closure, and inner classes · 372
code: coding standards · 21; coding style ·
88; organization · 221; reuse · 237;
source code · 18
collecting parameter · 713, 742
collection · 44, 394, 427, 884; classes ·
389; filling with a Generator · 636; list
of methods for · 809; utilities · 879
Collections: addAll( ) · 396;
enumeration( ) · 894; fill( ) · 793;
unmodifiableList( ) · 815
collision: during hashing · 848; name · 217
combo box · 1345
comma operator · 140
Command design pattern · 381, 603, 1031,
1121
comments, and embedded documentation
· 81
Commitment, Theory of Escalating · 1146
common interface · 311
Communicating Sequential Processes
(CSP) · 1299
Comparable · 779, 822, 828
Comparator · 780, 822
compareTo( ), in java.lang.Comparable ·
778, 824
comparing arrays · 777
compatibility: backwards · 655; migration ·
655
compilation unit · 211
compile-time constant · 262

compiling a Java program · 80
component, and JavaBeans · 1395
composition · 32, 237; and design · 304;
and dynamic behavior change · 306;
combining composition & inheritance ·
249; vs. inheritance · 256, 262, 830, 895
compression, library · 973
concurrency: active objects · 1295; and
containers · 887; and exceptions · 1158;
and Swing · 1382; ArrayBlockingQueue ·
1215; atomicity · 1151; BlockingQueue ·
1215, 1235; Brian's Rule of
Synchronization · 1156; Callable · 1124;
Condition class; constructors ·
1137; contention, lock · 1272;
CountDownLatch · 1230; CyclicBarrier ·
1232; daemon threads · 1130;
DelayQueue · 1235; Exchanger · 1250;
Executor · 1120; I/O between tasks
using pipes · 1221;
LinkedBlockingQueue · 1215; lock,
explicit · 1157; lock-free code · 1161; long
and double non-atomicity · 1161; missed
signals · 1203; performance tuning ·
1270; priority · 1127;
PriorityBlockingQueue · 1239;
producer-consumer · 1208; race
condition · 1152; ReadWriteLock · 1292;
ScheduledExecutor · 1242; semaphore ·
1246; sleep( ) · 1126; SynchronousQueue
· 1259; task interference · 1150;
terminating tasks · 1179; the Goetz Test
for avoiding synchronization · 1160;
thread local storage · 1177; thread vs.
task, terminology · 1142;
UncaughtExceptionHandler · 1148;
word tearing · 1161
ConcurrentHashMap · 834, 1282, 1287
ConcurrentLinkedQueue · 1282
ConcurrentModificationException · 888;
using CopyOnWriteArrayList to
eliminate · 1281, 1298
Condition class, concurrency · 1212
conditional compilation · 220
conditional operator · 116
conference, Software Development
Conference · 14
console: sending exceptions to · 497;
Swing display framework in
net.mindview.util.SwingConsole · 1310
constant: compile-time constant · 262;
constant folding · 262; groups of

Command · 381, 603, 1031, 1121; Data Transfer Object (Messenger idiom) · 621, 797, 860; Decorator · 717; Façade · 577; Factory Method · 339, 582, 627, 928; Factory Method, and anonymous classes · 361; Flyweight · 800, 1301; Iterator · 349, 406; Null Iterator · 598; Null Object · 598; Proxy · 593; Singleton · 232; State · 306; Strategy · 322, 332, 737, 764, 778, 780, 903, 910, 1036, 1238; Template Method · 375, 573, 666, 859, 969, 1173, 1279, 1284; Visitor · 1079

destructor · 173, 175, 473; Java doesn't have one · 251

diagram: class inheritance diagrams · 261; inheritance · 42

dialog: box · 1364; file · 1368; tabbed · 1349

dictionary · 394

Dijkstra, Edsger · 1224

dining philosophers, example of deadlock in concurrency · 1224

directory: and packages · 220; creating directories and paths · 912; lister · 902

dispatching: double dispatching · 1048; multiple, and enum · 1047

display framework, for Swing · 1310

dispose( ) · 1365

division · 98

documentation · 17; comments & embedded documentation · 81

double: and threading · 1161; literal value marker (d or D) · 109

double dispatching · 1048; with EnumMap · 1055

double-ended queue (deque) · 410

do-while · 138

downcast · 261, 308; type-safe downcast · 569

drawing lines in Swing · 1360

drop-down list · 1345

duck typing · 721, 733

dynamic: aggregate initialization syntax for arrays · 752; behavior change with composition · 306; binding · 277, 282; proxy · 594; type checking in Java · 814; type safety and containers · 710

E

early binding · 40, 281

East, BorderLayout · 1317

editor, creating one using the Swing JTextPane · 1341

efficiency: and arrays · 747; and final · 271

else keyword · 135

encapsulation · 228; using reflection to break · 607

encode( ), character set · 953

end sentinel · 626

endian: big endian · 958; little endian · 958

entrySet( ), in Map · 845

enum: adding methods · 1014; and Chain of Responsibility design pattern · 1036; and inheritance · 1020; and interface · 1023; and multiple dispatching · 1047; and random selection · 1021; and state machines · 1041; and static imports · 1013; and switch · 1016; constant-specific methods · 1032, 1053; groups of constant values in C & C++ · 335; keyword · 204, 1011; values( ) · 1011, 1017

enumerated types · 204

Enumeration · 894

EnumMap · 1030

EnumSet · 642, 899; instead of flags · 1028

equals( ) · 104; and hashCode( ) · 822, 853; and hashed data structures · 843; conditions for defining properly · 842; overriding for HashMap · 842

equivalence: == · 103; object equivalence · 103

erasure · 696; in generics · 650

Erlang language · 1113

error: handling with exceptions · 443; recovery · 443; reporting · 492; reporting errors in book · 21; standard error stream · 450

Escalating Commitment, Theory of · 1146

event: event-driven programming · 1312; event-driven system · 375; events and listeners · 1322; JavaBeans · 1394; listener · 1321; model, Swing · 1321; multicast, and JavaBeans · 1407; responding to a Swing event · 1312

EventSetDescriptors · 1401

H

handler, exception · 448
Harold, Elliotte Rusty · 1415, 1456; XOM
    XML library · 1003
has-a · 32; relationship, composition · 258
hash function · 847
hashCode( ) · 833, 839, 847; and hashed
    data structures · 843; equals( ) · 822;
    issues when writing · 851; recipe for
    generating decent · 853
hashing · 844, 847; and hash codes · 839;
    external chaining · 848; perfect hashing
    function · 848
HashMap · 834, 877, 1287, 1331
HashSet · 415, 821, 872
Hashtable · 877, 895
hasNext( ), Iterator · 407
Hexadecimal · 109
hiding, implementation · 228
Holub, Allen · 1295
HTML on Swing components · 1370

I

I/O: available( ) · 930; basic usage,
    examples · 927; between tasks using
    pipes · 1221; blocking, and available( ) ·
    930; BufferedInputStream · 920;
    BufferedOutputStream · 921;
    BufferedReader · 483, 924, 927;
    BufferedWriter · 924, 930;
    ByteArrayInputStream · 916;
    ByteArrayOutputStream · 917;
    characteristics of files · 912;
    CharArrayReader · 923;
    CharArrayWriter · 923;
    CheckedInputStream · 973;
    CheckedOutputStream · 973; close( ) ·
    928; compression library · 973;
    controlling the process of serialization ·
    986; DataInput · 926; DataInputStream
    · 920, 924, 929; DataOutput · 926;
    DataOutputStream · 921, 925;
    DeflaterOutputStream · 973; directory
    lister · 902; directory, creating
    directories and paths · 912;
    Externalizable · 986; File · 916, 925; File
    class · 901; File.list( ) · 901;
    FileDescriptor · 916; FileInputReader ·

927; FileInputStream · 916;
    FilenameFilter · 901; FileOutputStream
    · 917; FileReader · 483, 923; FileWriter ·
    923, 930; FilterInputStream · 916;
    FilterOutputStream · 917; FilterReader ·
    924; FilterWriter · 924; from standard
    input · 941; GZIPInputStream · 973;
    GZIPOutputStream · 973;
    InflaterInputStream · 973; input · 914;
    InputStream · 914; InputStreamReader ·
    922, 923; internationalization · 923;
    interruptible · 1189; library · 901;
    lightweight persistence · 980;
    LineNumberInputStream · 920;
    LineNumberReader · 924; mark( ) ·
    926; mkdirs( ) · 914; network I/O · 946;
    new nio · 946; ObjectOutputStream ·
    981; output · 914; OutputStream · 914,
    917; OutputStreamWriter · 922, 923;
    pipe · 915; piped streams · 936;
    PipedInputStream · 916;
    PipedOutputStream · 916, 917;
    PipedReader · 923; PipedWriter · 923;
    PrintStream · 921; PrintWriter · 924,
    930, 932; PushbackInputStream · 920;
    PushbackReader · 924;
    RandomAccessFile · 925, 926, 934;
    read( ) · 914; readDouble( ) · 934;
    Reader · 914, 922, 923; readExternal( ) ·
    986; readLine( ) · 485, 924, 931, 942;
    readObject( ) · 981; redirecting standard
    I/O · 942; renameTo( ) · 914; reset( ) ·
    926; seek( ) · 926, 934;
    SequenceInputStream · 916, 925;
    Serializable · 986; setErr(PrintStream) ·
    943; setIn(InputStream) · 943;
    setOut(PrintStream) · 943;
    StreamTokenizer · 924; StringBuffer ·
    916; StringBufferInputStream · 916;
    StringReader · 923, 928; StringWriter ·
    923; System.err · 941; System.in · 941;
    System.out · 941; transient · 991; typical
    I/O configurations · 927; Unicode · 923;
    write( ) · 914; writeBytes( ) · 933;
    writeChars( ) · 933; writeDouble( ) ·
    934; writeExternal( ) · 986;
    writeObject( ) · 981; Writer · 914, 922,
    923; ZipEntry · 977; ZipInputStream ·
    973; ZipOutputStream · 973
Icon · 1335
IdentityHashMap · 834, 877
if-else statement · 116, 135

isInstance( ) · 578; and generics · 663
isInterface( ) · 560
is-like-a · 307
Iterable · 629, 797; and array · 433; and
    foreach · 431
Iterator · 406, 409, 427; hasNext( ) · 407;
    next( ) · 407
Iterator design pattern · 349

J

Jacobsen, Ivar · 1457
JApplet · 1317; menus · 1352
JAR · 1412; file · 212; jar files and classpath
    · 216; utility · 978
Java: and set-top boxes · 111; AWT · 1303;
    bytecodes · 506; compiling and running
    a program · 80; Java Foundation
    Classes (JFC/Swing) · 1303; Java
    Virtual Machine (JVM) · 556; Java Web
    Start · 1376; public Java seminars · 15
Java standard library, and thread-safety ·
    1232
JavaBeans, see Beans · 1393
javac · 81
javadoc · 82
javap decompiler · 505, 610, 660
Javassist · 1104
JButton · 1335; Swing · 1311
JCheckBox · 1335, 1342
JCheckBoxMenuItem · 1353, 1357
JComboBox · 1345
JComponent · 1337, 1360
JDialog · 1364; menus · 1352
JDK 1.1 I/O streams · 922
JDK, downloading and installing · 80
JFC, Java Foundation Classes (Swing) ·
    1303
JFileChooser · 1368
JFrame · 1317; menus · 1352
JIT, just-in-time compilers · 181
JLabel · 1340
JList · 1347
JMenu · 1352, 1357
JMenuBar · 1352, 1358
JMenuItem · 1336, 1352, 1357, 1358, 1360
JNLP, Java Network Launch Protocol ·
    1376
join( ), threading · 1143
JOptionPane · 1350
Joy, Bill · 103

JPanel · 1334, 1360, 1392
JPopupMenu · 1359
JProgressBar · 1373
JRadioButton · 1335, 1344
JScrollPane · 1316, 1349
JSlider · 1373
JTabbedPane · 1349
JTextArea · 1315
JTextField · 1312, 1338
JTextPane · 1341
JToggleButton · 1334
JUnit, problems with · 1083
JVM (Java Virtual Machine) · 556

K

keyboard: navigation, and Swing · 1305;
    shortcuts · 1358
keySet( ) · 877

L

label · 146
labeled: break · 147; continue · 147
late binding · 40, 277, 281
latent typing · 721, 733
layout, controlling layout with layout
    managers · 1317
lazy initialization · 239
least-recently-used (LRU) · 838
left-shift operator (<<) · 112
length: array member · 194; for arrays ·
    749
less than (<) · 103
less than or equal to (<=) · 103
lexicographic: sorting · 418; vs. alphabetic
    sorting · 783
library: creator, vs. client programmer ·
    209; design · 210; use · 210
LIFO (last-in, first-out) · 412
lightweight: object · 406; persistence · 980
LineNumberInputStream · 920
LineNumberReader · 924
LinkedBlockingQueue · 1215
LinkedHashMap · 834, 838, 877
LinkedHashSet · 416, 821, 872, 874
LinkedList · 401, 410, 423, 817
linking, class · 563
list: boxes · 1347; drop-down list · 1345

160; generics · 699; lack of name hiding during inheritance · 255; method overloading · 158; on return values · 165; operator + and += overloading for String · 242, 504; operator overloading · 118; vs. overriding · 255

overriding: and inner classes · 383; function · 36; private methods · 290; vs. overloading · 255

P

package · 210; access, and friendly · 221; and directory structure · 220; creating unique package names · 214; default · 211, 223; names, capitalization · 75; package access, and protected · 258
paintComponent( ) · 1360, 1368
painting on a JPanel in Swing · 1360
parameter, collecting · 713, 742
parameterized types · 617
parseInt( ) · 1368
pattern, regular expression · 527
perfect hashing function · 848
performance: and final · 271; nio · 967; test, containers · 859; tuning, for concurrency · 1270
persistence · 996; lightweight persistence · 980
PhantomReference · 889
philosophers, dining, example of deadlock in concurrency · 1224
pipe · 915
piped streams · 936
PipedInputStream · 916
PipedOutputStream · 916, 917
PipedReader · 923, 1221
PipedWriter · 923, 1221
pipes, and I/O · 1221
Plauger, P.J. · 1458
pluggable look & feel · 1373
pointer, Java exclusion of pointers · 372
polymorphism · 38, 277, 310, 554, 613; and constructors · 293; and multiple dispatching · 1048; behavior of polymorphic methods inside constructors · 301
pool, object · 1246
portability in C, C++ and Java · 123
position, absolute, when laying out Swing components · 1320

possessive quantifiers · 529
post-decrement · 102
postfix · 102
post-increment · 102
pre-decrement · 102
preferences API · 1006
prefix · 102
pre-increment · 102
prerequisites, for this book · 23
primitive: comparison · 104; data types, and use with operators · 123; final · 263; final static primitives · 264; initialization of class fields · 182; types · 65
primordial class loader · 556
printf( ) · 514
printStackTrace( ) · 458, 461
PrintStream · 921
PrintWriter · 924, 930, 932; convenience constructor in Java SE5 · 937
priority, concurrency · 1127
PriorityBlockingQueue, for concurrency · 1239
PriorityQueue · 425, 827
private · 31, 210, 221, 224, 258, 1155; illusion of overriding private methods · 268; inner classes · 377; interfaces, when nested · 339; method overriding · 290; methods · 303
problem space · 24
process control · 944
process, concurrent · 1112
ProcessBuilder · 944
ProcessFiles · 1100
producer-consumer, concurrency · 1208
programmer, client · 30
programming: basic concepts of object-oriented programming (OOP) · 23; event-driven programming · 1312; Extreme Programming (XP) · 1457; multiparadigm · 25; object-oriented · 553
progress bar · 1371
promotion, to int · 122, 132
property · 1394; bound properties · 1414; constrained properties · 1414; custom property editor · 1414; custom property sheet · 1414; indexed property · 1414
PropertyChangeEvent · 1414
PropertyDescriptors · 1400
ProptertyVetoException · 1414

protected · 31, 210, 221, 225, 258; and package access · 258; is also package access · 227
protocol · 316
proxy: and java.lang.ref.Reference · 890; for unmodifiable methods in the Collections class · 817
Proxy design pattern · 593
public · 31, 210, 221, 222; and interface · 316; class, and compilation units · 211
pure substitution · 37, 307
PushbackInputStream · 920
PushbackReader · 924
pushdown stack · 412; generic · 625
Python · 1, 5, 9, 53, 60, 722, 787, 1113, 1460

Q

qualified name · 560
quantifier: greedy · 529; possessive · 529; regular expression · 529; reluctant · 529
queue · 389, 410, 423, 827; performance · 863; synchronized, concurrency · 1215
queuing discipline · 425

R

race condition, in concurrency · 1152
RAD (Rapid Application Development) · 588
radio button · 1344
ragged array · 755
random selection, and enum · 1021
random( ) · 419
RandomAccess, tagging interface for containers · 441
RandomAccessFile · 925, 926, 934, 948
raw type · 651
reachable objects and garbage collection · 889
read( ) · 914; nio · 948
readDouble( ) · 934
Reader · 914, 922, 923
readExternal( ) · 986
reading from standard input · 941
readLine( ) · 485, 924, 931, 942
readObject( ) · 981; with Serializable · 992
ReadWriteLock · 1292

recursion, unintended via toString( ) · 509
redirecting standard I/O · 942
ReentrantLock · 1160, 1192
refactoring · 209
reference: assigning objects by copying references · 96; final · 263; finding exact type of a base reference · 555; null · 67; reference equivalence vs. object equivalence · 104
reference counting, garbage collection · 178
Reference, from java.lang.ref · 889
referencing, forward · 184
reflection · 588, 1324, 1398; and Beans · 1394; and weak typing · 496; annotation processor · 1064, 1071; breaking encapsulation with · 607; difference between RTTI and reflection · 589; example · 1334; latent typing and generics · 726
regex · 527
Registered Factories, variation of Factory Method design pattern · 582
regular expressions · 523
rehashing · 878
reification, and generics · 655
relational operators · 103
reluctant quantifiers · 529
removeActionListener( ) · 1403, 1410
removeXXXListener( ) · 1322
renameTo( ) · 914
reporting errors in book · 21
request, in OOP · 27
reset( ) · 926
responsive user interfaces · 1145
resume( ), and deadlocks · 1184
resumption, termination vs. resumption, exception handling · 449
re-throwing an exception · 461
return: an array · 753; and finally · 476; constructor return value · 157; covariant return types · 303, 706; overloading on return value · 165; returning multiple objects · 621
reusability · 32
reuse: code reuse · 237; reusable code · 1393
rewind( ) · 953
right-shift operator (>>) · 112
rollover · 1337
RoShamBo · 1048
Rumbaugh, James · 1457

T

W

X

Y

Z