Comments About "Thinking in C++":

Best Book! Winner of the Software Development Magazine Jolt Award!

"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

Bruce Eckel
President, MindView Inc.

Revision 8, July 7 1997

Full text, source code and updates to this book available at http://www.EckelObjects.com/Eckel

contact the author if you would like to include an electronic version of this book on your product CD

Bruce Eckel's Hands-On Java Seminar Multimedia CD

It's like coming to the seminar!

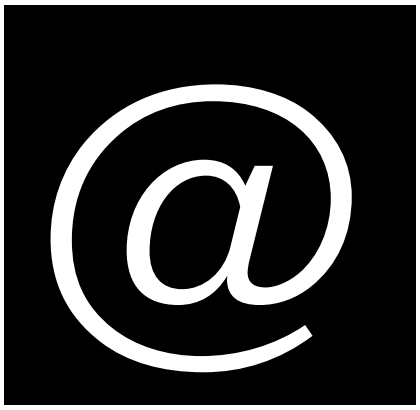
Available at http://www.EckelObjects.com/Eckel

(August or September 1997)

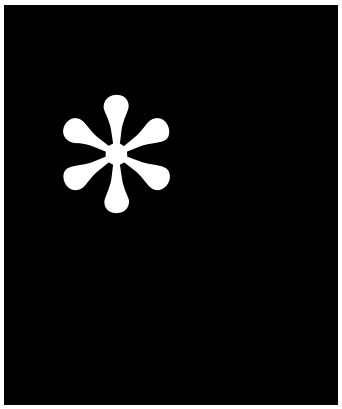
Contains:

- Overhead slides and Audio for all the lectures: just play it to see and hear the lectures as they happened!
- Multimedia introduction to each lecture
- Entire set of lectures is keyword indexed so you can rapidly locate the discussion of the subject you're interested in
- Electronic version of "Thinking in Java" book on CD with automatic Internet update system
- Source code from the book

Overview



what's inside...



preface

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 just look at Java as a collection of features; some of the features make no sense in isolation. You can only use the sum of the parts if you are thinking about *design*, not simply coding. And to understand Java in this way, you must understand the problems with Java and with programming in general. This book discusses programming problems, why they are problems, and the approach Java has taken to solve such problems. Thus, the set of features I explain in each chapter will be 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 be able to feed it to your model and deduce the answer.

prerequisites

This book assumes you have some programming familiarity, so you understand that a program is a collection of statements, the idea of a subroutine/function/macro, control statements like "if" and looping constructs like "while," etc. However, you may have learned this in many places, such as programming with a macro language or a tool like Perl. Just so 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 the C programmers and more so for the C++ programmers, but don't count yourself out if you're not experienced with those languages (but come willing to work hard). I'll be introducing the concepts of object-oriented programming and Java's basic control mechanisms, so you'll be exposed to those, and the first exercises will involve the basic control-flow statements.

Although references will often 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 which, after all, Java is descended from. I will attempt to make these references simple and to explain anything 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 the language. Teaching programming languages has become my profession; I've seen nodding heads, blank faces, and puzzled expressions in audiences all over the world since 1989. 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 chairing the C++ track at the Software Development Conference for the last few years (and now also the Java track), that I and other speakers tended to give the typical audience too many topics, too fast. 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 Java program design). Eventually I developed a course using everything I had learned from my teaching experience, one I would be happy giving for a long time. It tackles the learning problem in discrete, easy-to-digest steps and for a hands-on seminar (the ideal learning situation), there are exercises following each of the short lessons. I now give this course in public Java seminars which you can find out about at http://www.EckelObjects.com/Eckel.

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

goals

Like my previous book *Thinking in C++*, this book has come to be structured around the process of teaching the language. In particular, my motivation is to create something that provides me a way to teach the language in my own seminars. Thus, when I think of a chapter in the book, I think in terms of what makes a good lesson during a seminar. My goal is to get bite-sized pieces that can be taught in a reasonable amount of time, followed by exercises that are feasible to accomplish in a classroom situation.

My goals in this book are to:

- Present the material a simple step at a time, so the reader can easily digest each concept before moving on.
- 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.
- Carefully sequence the presentation of features so that you aren't seeing something you haven't been exposed to. Of course, this isn't always possible; in those situations, a brief introductory description will be given.
- Give you what I think is important for you to understand about the language, rather than everything I know. I believe there is an "information importance hierarchy," and there are some facts that 95% of programmers will never need to know, but would just confuse people and add to 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 have to think about it, it will confuse the reader/maintainer of that code. So forget about precedence, and use parentheses when things aren't clear.

- Keep each section focused enough so the lecture time — and the time between exercise periods — is small. Not only does this keep the audience' minds more active and involved during a hands-on seminar, but it gives the reader a greater sense of accomplishment.
- Provide the reader with a solid foundation so they can understand the issues well enough to move on to more difficult coursework and books.

chapters

This course was designed with one thing in mind: the way people learn the Java language. Audience feedback helped me understand which parts were difficult and needed extra 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 have to explain them all, and the student's confusion is easily compounded. As a result, I've taken a great deal of trouble to introduce the features as few at a time as possible.

The goal, then, is for each chapter to teach a single feature, or a small group of associated features, in such a way that no additional features are relied upon. That way you can digest each piece in the context of your current knowledge before moving on.

Here is a brief description of the chapters contained in the book, which correspond to lectures and exercise periods in my hands-on seminars.

Chapter 1: Introduction to Objects
This chapter is an overview of what object-oriented programming is all about, including the answer to the basic question "what's an object?", interface vs. implementation, abstraction and encapsulation, messages and functions, inheritance and composition, and the all-important polymorphism. Then you'll be introduced to issues of object creation like constructors, where the objects live, where to put them once you create them (answer: in containers), and the magical garbage collector that cleans everything up when it's no longer needed. Other issues will be introduced, like error handling with exceptions and multithreading for responsive user interfaces. You'll also learn about what makes Java special and why it's been so successful.

Chapter 2: Everything is an Object
This chapter gets you to the point that you can write your first Java program, so it must cover the essentials, including: the concept of a "handle" to an object; how to create an object; an introduction to primitive types and arrays; scoping and the way objects are destroyed by the garbage collector; how everything in Java is a new data type (class) and how to create your own classes; functions, arguments, and return values; name visibility and using components from other libraries; the **static** keyword; comments and embedded documentation.

Chapter 3: Controlling Program Flow
This chapter begins with all the operators that come to Java from C and C++. In addition, you'll discover common operator pitfalls, casting, promotion and precedence. This is followed by the basic control-flow and selection operations that you get with virtually any programming language: choice with if-else; looping with for and while; quitting a loop with break and continue as well as Java's labeled break and labeled continue (which account for the "missing goto" in Java); and selection using switch. Although much of this material has common threads with C and C++ code, there are some differences. In addition, all the examples will be full Java examples so you'll be getting more comfortable with what Java looks like.

Chapter 4: Initialization & Cleanup
This chapter begins by introducing the constructor to guarantee proper initialization. The definition of the constructor leads into the concept of function overloading (since you may want several constructors). This is followed by a discussion of the process of cleanup, which is not always as simple as it seems. Normally you just drop an object when you're done with it and the garbage collector eventually comes along and releases the memory. This portion explores the garbage collector and

some of its idiosyncrasies. The chapter concludes with a closer look at how things are initialized: automatic member initialization, specifying member initialization, the order of initialization, **static** initialization, and array initialization.

Chapter 5: Hiding The Implementation
This chapter covers the way that code is packaged together, and how some parts of a library are exposed and other parts are hidden. It begins by looking at the **package** and **import** keywords, which perform file-level packaging and allow you to build libraries of classes. The subject of directory paths and file names is examined. The remainder of the chapter looks at the **public**, **private**, and **protected** keywords and what they mean when used in various contexts.

Chapter 6: Reusing Classes
The concept of inheritance is standard in virtually all OOP languages. It's a way to take an existing class and add to its functionality (as well as change it, the subject of the next chapter), so inheritance is often a way to re-use code by leaving the "base class" the same, and just patching things here and there to produce what you want. However, inheritance isn't the only way to make new classes from existing ones; you can also embed an object inside your new class with *composition*. In this chapter you'll learn about these two ways to reuse in Java and how to apply them.

Chapter 7: Polymorphism
On your own, you might take nine months to discover and understand this cornerstone of OOP. Through small, simple examples you'll see how to create a family of types with inheritance and manipulate objects in that family through their common base class. Java's polymorphism allows you to treat all objects in this family generically, which means the bulk of your code doesn't rely on specific type information. This makes your programs extensible, so building programs and code maintenance is easier and cheaper. In addition, Java provides a third way to set up a reuse relationship: through the *interface*, which is a pure abstraction of the interface of an object. Once you've seen polymorphism, the interface can be clearly understood.

Chapter 8: Holding Your Objects
It's a fairly simple program that only ever has a fixed quantity of objects with known lifetimes. In general your programs will always be creating new objects at a variety of times that will only be known at the time the program is running. In addition, you won't know until run-time the quantity or even the exact type of the objects you need. To solve the general programming problem, you need to create any number of objects, anytime, anywhere. This chapter explores in depth the tools that Java supplies to hold objects while you're working with them: the simple arrays and more sophisticated containers (data structures) like **Vector** and **Hashtable**.

Chapter 9: Error Handling With Exceptions
The basic philosophy of Java is that "badly-formed code will not be run." As much as possible, the compiler catches problems, but sometimes the problems – either programmer error or a natural error condition that occurs as part of the normal execution of the program – can only be detected and dealt with at run-time. Java has *exception handling* to deal with any problems that arise while the program is running. This chapter examines how the keywords **try**, **catch**, **throw**, **throws**, and **finally** work in Java, when you should throw exceptions, and what to do when you catch them. In addition, you'll see Java's standard exceptions, how to create your own, what happens with exceptions in constructors, and how exception handlers are located.

Chapter 10: The Java IO System
Theoretically you can divide any program into 3 parts: input, process, and output. This implies that IO (input/output) is a pretty important part of the equation. In this chapter you'll learn about the different classes that Java provides for reading and writing files, blocks of memory, and the console. The distinction between "old" IO and "new" Java 1.1 IO will be shown. In addition, this section examines the process of taking an object, "streaming" it (so that it can be placed on disk or sent across a network) and reconstructing it, which is handled for you in Java version 1.1. In addition, Java 1.1's compression libraries, which are used in the Java ARchive file format (JAR), are examined.

Chapter 11: Run-time type identification
Java run-time type identification (RTTI) lets you find the exact type of an object when you only have a pointer or reference to the base type. Normally, you'll want to intentionally ignore the exact type of an object and let Java's dynamic binding mechanism (polymorphism) implement the correct behavior for

that type. But occasionally it is very helpful to know the exact type of an object for which you only have a base pointer; often this information allows you to perform a special-case operation more efficiently. This chapter explains what RTTI is for, how to use it, and how to get rid of it when it doesn't belong there. In addition, the Java 1.1 *reflection* feature is introduced.

Chapter 12: Passing & Returning Objects
Since the only way you talk to objects in Java is through "handles," the concepts of passing an object into a function, and returning an object from a function, have some interesting consequences. This explains what you need to know to manage objects when you're moving in and out of functions, and also shows the **String** class, which uses a different approach to the problem.

Chapter 13: Creating Windows and Applets
Java comes with the *Abstract Window Toolkit* (AWT), which is a set of classes that handle windowing in a portable fashion; these windowing programs can either be applets or stand-alone applications. This chapter is an introduction to the AWT and the creation of World-Wide-Web "applets." We'll also look at pros and cons of the AWT, and the GUI improvements introduced in Java 1.1. Finally, the very important "JavaBeans" technology is introduced, which is fundamental for the creation of Rapid-Application Development (RAD) application building tools.

Chapter 14: Multiple Threads
Java provides a built-in facility to support multiple concurrent processes running within a single program (unless you have multiple processors on your machine, this is actually the *appearance* of several processes). Although these can be used anywhere, they are most powerful when trying to create a responsive user interface so, for example, a user isn't prevented from pressing a button or entering data while some processing is going on. This chapter looks at the syntax and semantics of multithreading in Java.

Chapter 15: Network Programming
All the Java features and libraries seem to really come together when you start writing programs to work across networks. This chapter explores communication across the Internet, and the classes that Java provides to make this easier. It also covers Java 1.1's *Java DataBase Connectivity* (JDBC) and *Remote Method Invocation* (RMI).

Chapter 16: Design patterns
This chapter introduces the very important and yet non-traditional "patterns" approach to program design. An example of the design evolution process will be studied, starting with an initial solution and moving through the logic and process of evolving the solution to more appropriate designs. You'll see one way that a design can materialize over time.

Chapter 17: Projects
This chapter includes a set of projects that build on the material presented in this book, or otherwise didn't fit in earlier chapters. These projects are significantly more complex than the examples in the rest of the book, and they often demonstrate new techniques and uses of class libraries.

In addition, there are subjects that didn't seem to fit within the core of the book, and yet I find that I discuss them during seminars. These are placed in the appendices:

Appendix A: Using non-Java Code
A totally portable Java program has serious drawbacks: speed and the inability to access platform-specific services. When you know the platform that you're running on it's possible to dramatically speed up certain operations by making them *native methods*, which are functions that are written in another programming language (currently, only C/C++ is supported). There are other ways that Java supports non-Java code, including CORBA. This appendix contains pointers to other resources for connecting Java to non-Java code.

Appendix B: Comparing C++ and Java
If you're a C++ programmer you already have the basic idea of object-oriented programming, and the syntax of Java no doubt looks very familiar to you. This makes sense since Java was derived from C++. However, there are a surprising number of differences between C++ and Java. These differences are intended to be significant improvements, and if you understand the differences you'll see why Java is such a beneficial programming language. This appendix takes you through the important features that make Java distinct from C++.

Appendix C: Java programming guidelines
This appendix contains suggestions to help guide you while performing low-level program design and also while writing code.

Appendix D: A bit about garbage collection
Describes the operation and approaches for garbage collection.

Appendix E: Recommended reading
There are a lot of Java books out there, and a lot of them simply take the online documentation downloadable from JavaSoft and format those docs into a book, with some hasty prose added. They're not all like that, however, and these are some of the Java books I've found particularly useful.

exercises

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

These 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 all the students are absorbing the material. Some exercises are more advanced to prevent boredom on the part of experienced students. They're all designed to be solved in a short time and are only there to test and polish your knowledge rather than present major challenges (presumably, you'll find those on your own — or more likely they'll find you).

source code

All the source code for this book is available as copyrighted freeware, distributed as a single package, by visiting the Web site http://www.EckelObjects.com/Eckel. To make sure that you get the most current version, this is the only official site for distribution of the code and it cannot be redistributed from other sites. You can, however, distribute the code in classroom and other educational situations. The reason for the sole distribution point is that mirror sites often don't update to the most recent version, and this will prevent old copies from drifting around on the net.

The copyright prevents you from republishing the code in print media without permission.

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

```
//////////////////////////////////////////////////
// Copyright (c) Bruce Eckel, 1997
// Source code file from the book "Thinking in Java"
// All rights reserved EXCEPT as allowed by the
// following statements: You may freely use this file
// for your own work (personal or commercial),
// including modifications and distribution in
// executable form only. You may not copy and
// distribute this file, but instead the sole
// distribution point is
// http://www.EckelObjects.com/Eckel where it is
// freely available. You may not remove this
// copyright and notice. You may not distribute
// modified versions of the source code in this
// package. You may not use this file in printed
// media without the express permission of the
// author. Bruce Eckel makes no representation about
// the suitability of this software for any purpose.
// It is provided "as is" without express or implied
// warranty of any kind, including any implied
```

```
// warranty of merchantability, fitness for a
// particular purpose or non-infringement. The entire
// risk as to the quality and performance of the
// software is with you. Bruce Eckel and the
// publisher shall not be liable for any damages
// suffered by you or any third party as a result of
// using or distributing software. In no event will
// Bruce Eckel or the publisher be liable for any
// lost revenue, profit or data, or for direct,
// indirect, special, consequential, incidental or
// punitive damages, however caused and regardless of
// the theory of liability, arising out of the use of
// or inability to use software, even if Bruce Eckel
// and the publisher have been advised of the
// possibility of such damages. Should the software
// prove defective, you assume the cost of all
// necessary servicing, repair, or correction. If you
// think you've found an error, please email all
// modified files with loudly commented changes to:
// Bruce@EckelObjects.com. (please use the same
// address for non-code errors found in the book).
///////////////////////////////////////////////
```

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

coding standards

In the text of this book, identifiers (function, variable, and class names) will be set in **bold**. Most keywords will also be set in bold, except for those keywords which 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. This style seems to be supported by most Java development environments. It was developed over a number of years, and was inspired by Bjarne Stroustrup's style in his original *The C++ Programming Language* (Addison-Wesley, 1991; 2nd ed.). 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.

The programs in this book are files that are included by the word processor in the text, directly from compiled files. Thus, the code files printed in the book should all work without compiler errors. The errors that *should* cause compile-time error messages are commented out with the comment //**!** so they can be easily discovered and tested using automatic means. Errors discovered and reported to the author will appear first in the distributed source code and later in updates of the book (which will also appear on the Web site http://www.EckelObjects.com/Eckel)

java versions

Although I test the code in this book with several different vendor implementations of Java, I generally rely on the Sun implementation as a reference when determining whether behavior is correct.

Sun has released two major versions of Java, 1.0 and (about a year later) 1.1. The latter version represents a very significant change to the language and should probably have been labeled 2.0 (and if 1.1 is such a big change from 1.0, I shudder to think what will justify the number 2.0).

This book covers both version 1.0 and version 1.1, although in places where the 1.1 approach is clearly superior to the 1.0 approach, I definitely favor the new version, often choosing to teach the better approach and completely ignore the 1.0 approach (there are plenty of other books that teach 1.0).

One thing you'll notice is that I don't use the sub-revision numbers. At this writing, the released version of 1.0 from Sun/Javasoft was 1.02 and the released version of 1.1 was 1.1.1. In this book I will only refer to Java 1.0 and Java 1.1, to guard against typographical errors produced by further sub-revisioning of these products.

seminars & mentoring

My company provides five-day, hands-on, public & in-house training seminars based on the material in this book. Selected material from each chapter represents a lesson, which is followed by a monitored exercise period so each student receives personal attention. The lectures and slides for the introductory seminar is also captured on CD-ROM to provide at least some of the experience of the seminar without the travel and expense. For more information, go to

http://www.EckelObjects.com/Eckel

or email:

Bruce@EckelObjects.com

My company also provides consulting services to help guide your project through its development cycle, especially your company's first Java project.

errors

No matter how many tricks 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 send the original source file (which you can find at **http://www.EckelObjects.com/Eckel**) with a clearly-commented error and suggested correction via electronic mail to **Bruce@EckelObjects.com** so it may be fixed in the electronic version (on the Web site) and the next printing of the book. Also, suggestions for additional exercises or requests to cover specific topics in the next edition are welcome. Your help is appreciated.

acknowledgements

First of all, thanks to the Doyle Street Cohousing Community for putting up with me for the 1½ years that it took me to write this book.

Thanks to Paul Tyma (co-author of *Java Primer Plus*, 1996 The Waite Group), D'arcy Smith (Symantec), Cay Horstmann (co-author of *Core Java* 1996 Prentice Hall*)* for helping me clarify concepts in the language.

Thanks to people who have spoken in my Java track at the Software Development Conference, and students in my seminars, who ask the questions I need to hear in order to make the material clearer.

Special thanks to Larry and Tina O'Brien, who turned this book and my seminar into a teaching CD ROM (you can find out more at **http://www.EckelObjects.com/Eckel)**.

Lots of people sent in corrections and I am indebted to them all, but particular thanks go to: Kevin Raulerson (tons of great bugs found), Bob Resendes (ditto), Dr. Robert Stephenson, Franklin Chen, David Karr, Leander A. Stroschein, Steve Clark, Austin Maher, Dennis P. Roth, Roque Oliveira, Douglas Dunn, Dejan Ristic, Neil Galarneau, David B. Malkovsky, and others.

There have been a spate of smart technical people in my life who have become friends and have also been both influential and unusual in that they're vegetarians, do Yoga and practice other forms of spiritual enhancement, which I find quite inspirational and instructional: Kraig Brockshmidt, Gen Kiyooka, Andrea Provaglio (who helps in the understanding of Java and programming in general in Italy).

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: Neil Rubenking (who used to do the Yoga/Vegetarian/Zen thing but discovered computers), Marco Cantu (another Italian – perhaps being steeped in Latin gives one aptitude for programming languages?) and of course Zack Urlocker, long-time pal whom I've traveled the world with.

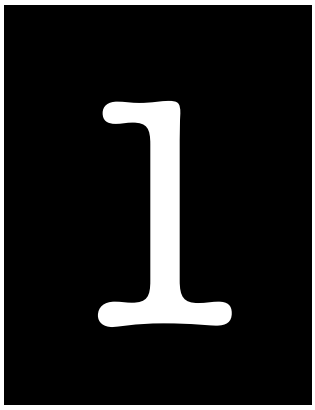
My friend Richard Hale Shaw's insights and support have been very helpful (and Kim's, too). Thanks also to KoAnn Vikoren, Eric Faurot, Lisa Monson, Julie Shaw, Nicole Freeman, Cindy Blair, Barbara Hanscome, Regina Ridley, Alex Dunne, and the rest of the cast and crew at MFI.

The book design, cover design, and cover photo were created by my friend Daniel Will-Harris, noted author and designer (http://www.Will-Harris.com), who used to play with rub-on letters in junior high school while he awaited the invention of computers and desktop publishing, and complained of me mumbling over my algebra problems. However, I produced the camera-ready pages myself, so the typesetting errors are mine. Microsoft® Word for Windows 7 was used to write the book and to create camera-ready pages. The body typeface is ITC Cheltenham and the headlines are in ITC American Typewriter.

Thanks to the vendors who supplied me with compilers: Borland, Microsoft, Symantec, Sybase/Powersoft/Watcom, and of course Sun.

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

The supporting cast of friends includes, but is not limited to: Andrew Binstock, Steve Sinofsky, JD Hildebrandt, Tom Keffer, Brian McElhinney, Brinkley Barr, Bill Gates at Midnight Engineering Magazine, Larry Constantine & Lucy Lockwood, Greg Perry, Dan Putterman, Christi Westphal, Gene Wang, Dave Mayer, David Intersimone, Andrea Rosenfield, Claire Sawyers, more Italians (Laura Fallai, Corrado, Ilsa and Cristina Giustozzi), Chris & Laura Strand, The Almquists, Brad Jerbic, Marilyn Cvitanic, The Mabrys, The Haflingers, The Pollocks, Peter Vinci, The Robbins Families, The Moelter Families (& the McMillans), The Wilks, Dave Stoner, Laurie Adams, The Penneys, The Cranstons, Larry Fogg, Mike & Karen Sequeira, Gary Entsminger & Allison Brody, Kevin Donovan & Sonda Eastlack, Chester & Shannon Andersen, Joe Lordi, Dave & Brenda Bartlett, Robert Herald, The Rentschlers, The Sudeks, Dick, Patty, and Lee Eckel, Lynn & Todd, and their families. And of course, Mom & Dad.



1: introduction to objects

Why has object-oriented programming had such a sweeping impact on the software development community?

Object-oriented programming appeals at multiple levels. For managers it promises faster and cheaper development and maintenance. For analysts and designers the modeling process becomes simpler and produces a clear, manageable design. For programmers the elegance and clarity of the object model and the power of object-oriented tools and libraries makes programming a much more pleasant task, and programmers experience an increase in productivity. Everybody wins, it would seem.

If there's a downside it is the expense of the learning curve. Thinking in objects is a dramatic departure from thinking procedurally, and the process of *designing* objects is much more challenging than procedural design, especially if you're trying to create reusable objects. In the past, a novice practitioner of object-oriented programming was faced with a choice of daunting tasks:

1. Choose a language like Smalltalk where you had to learn a large library before becoming productive

2. Choose C++ with virtually no libraries at all[1], and struggle through the depths of the language in order to write your own libraries of objects.

It is, in fact, difficult to design objects well – for that matter, it's hard to design *anything* well. But the intent is that a relatively few experts design the best objects for others to consume. Successful OOP languages incorporate not just language syntax and a compiler, but an entire development environment *including* a significant library of well-designed, easy to use objects. Thus, the primary job of most programmers is not to produce new types of objects but to utilize existing types to solve their application problems. The goal of this chapter is to show you what object-oriented programming is and how simple it can be.

This chapter will introduce many of the ideas of Java and object-oriented programming on a conceptual level, but keep in mind that you're not expected to be able to write full-fledged Java programs after reading this chapter. All the detailed descriptions and examples will follow throughout the course of this book.

the progress of abstraction

All programming languages provide abstractions. It can be argued that the complexity of the problems you can solve is directly related to the kind and quality of abstraction. By "kind" I mean: what is it you are abstracting? Assembly language is a small abstraction of the underlying machine. Many so-called "imperative" languages that followed (like FORTRAN, BASIC, and C) were abstractions of assembly language. These languages are big improvements over assembly language, but their primary abstraction still requires you to think in terms of the structure of the computer rather than the structure of the problem you are trying to solve. The programmer is required to establish the association between the machine model (in the "solution space") and the model of the problem that is actually being solved (in the "problem space"). The effort required to perform this mapping, and the fact that it is extrinsic to the programming language, produces programs that are difficult to write and expensive to maintain, and as a side effect created the entire "programming methods" industry.

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

The object-oriented approach takes a further step by providing tools for the programmer to represent elements in the problem space. This representation is general enough that the programmer is not constrained to any particular type of problem. We refer to the elements in the problem space and their representations in the solution space as "objects" (of course, you will also need other objects that don't have problem-space analogs). The idea is that the program is allowed to adapt itself to the lingo of the problem by adding new types of objects, so when you read the code describing the solution, you're reading words that also express the problem. This is a more flexible and powerful language abstraction than what we've had before.

There's still a connection back to the computer, though. Each object looks quite a bit like a little computer: it has a state, and it has operations you can ask it to perform. However, this doesn't seem like such a bad analogy to objects in the real world: they all have characteristics and behavior.

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

---

[1] Fortunately, this has changed significantly with the advent of third-party libraries and the Standard C++ library.

1. **Everything is an object.** Think of an object as a fancy variable: it stores data, but you can also ask it to perform operations on itself by making requests. In theory, you can take any conceptual component in the problem you're trying to solve (dogs, buildings, services, etc.) and represent it as an object in your program.
2. **A program is a bunch of objects telling each other what to do by sending messages**. To make a request of an object, you "send a message" to that object. More concretely, you can think of a message as a request to call a function for a particular object.
3. **Each object has its own memory made up of other objects**. Or, you make a new kind of object by making a package containing existing objects. Thus, you can build up complexity in a program while hiding it behind the simplicity of objects.
4. **Every object has a type**. Using the parlance, each object is an *instance* of a *class*, where "class" is synonymous with "type." The most important distinguishing characteristic of a class is "what messages can you send to it?"
5. **All objects of a particular type can receive the same messages**. This is actually a very loaded statement, as you will see later: because an object of type circle is also an object of type shape, a circle is guaranteed to receive shape messages. This means you can write code that talks to shapes, and automatically handle anything that fits the description of a shape. This *substitutability* is one of the most powerful concepts in OOP.

Some language designers have decided that object-oriented programming itself is not adequate to easily solve all programming problems, and advocate the combination of various approaches into *multiparadigm* programming languages[2].

an object has an interface

Aristotle was probably the first to begin a careful study of the concept of type. He was known to speak of "the class of fishes and the class of birds." The concept that all objects, while being unique, are also part of a set of objects that have characteristics and behaviors in common was directly used in the first object-oriented language, Simula-67, with its fundamental keyword **class** that introduces a new type into a program (thus *class* and *type* are often used synonymously[3]).

Simula, as its name implies, was created for the purpose of developing simulations such as the classic "bank-teller problem." In this, you have a bunch of tellers, customers, accounts, transactions, etc. The members of each class share some commonality: every account has a balance, every teller can accept a deposit, etc. At the same time, each member has its own state: each account has a different balance, each teller has a name. Thus the tellers, customers, accounts, transactions, etc. can each be represented with a unique entity in the computer program. This entity is the object, and each object belongs to a particular class that defines its characteristics and behaviors.

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

Once a type is established, you can make as many objects of that type as you like, and then manipulate those objects as if they were themselves the elements that exist in the problem you are trying to solve. Indeed, one of the challenges of object-oriented programming is to create a one-to-one mapping between the elements in the *problem space* (the place where the problem actually exists) and the *solution space* (the place where you're modeling that problem, e.g. the computer).

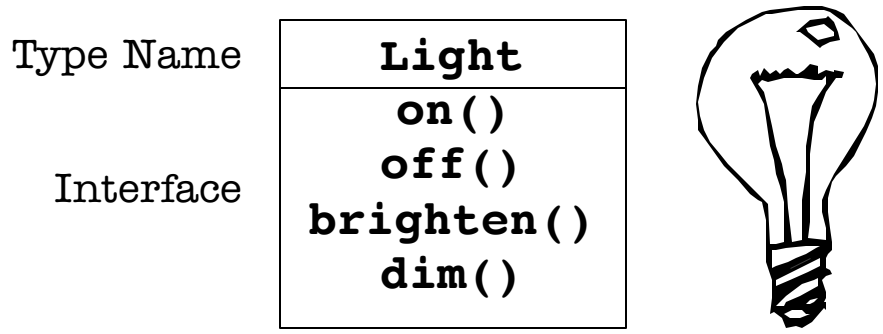
---

[2] See *Multiparadigm Programming in Leda* by Timothy Budd (Addison-Wesley 1995).

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

But how do you get an object to do useful work for you? There must be some way of making a request of that object so that it will do something (complete a transaction, draw something on the screen, turn on a switch, etc.). In addition, each object can satisfy only certain requests. The requests you can make of an object are defined by its *interface*, and the type is what determines the interface. The idea of type being equivalent to interface is fundamental in object-oriented programming.

A simple example might be a representation of a light bulb:




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

The name of the type/class is **Light**, and the requests that you can make of a **Light** object are to turn it on, turn it off, make it brighter or make it dimmer. You create a "handle" for a **Light** simply by declaring a name (**lt**) for that identifier, and you make an object of type **Light** with the **new** keyword, assigning it to the handle with the **=** sign. To send a message to the object, you state the handle name and connect it to the message name with a period (dot). From the standpoint of the user of a pre-defined class, that's pretty much all there is to programming with objects.

the hidden implementation

It is helpful to break up the playing field into *class creators* (those who create new data types) and *client programmers*[4] (the class consumers who use the data types in their applications). The goal of the client programmer is to collect a toolbox full of classes to use for rapid application development. The goal of the class creator is to build a class that exposes only what's necessary to the client programmer, and keeps everything else hidden. Why? Because if it's hidden, the client programmer can't use it, which means that the class creator can change the hidden portion at will, without worrying about the impact to anyone else.

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

In any relationship it's important to have boundaries that are respected by all parties involved. When you create a library, you establish a relationship with the client programmer, who is another programmer, but one putting together an application or using your library to build a bigger library.

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

If all the members of a class are available to everyone, then the client programmer can do anything they want with that class and there's no way to force any particular behaviors. Even though you might really prefer that the client programmer not directly manipulate some of the members of your class, without access control there's no way to prevent it. Everything's naked to the world.

There are two reasons for controlling access to members. The first is to keep client programmers' hands off portions they shouldn't touch, parts that are necessary for the internal machinations of the data type, but not part of the interface that users need to solve their particular problems. This is actually a service to users because they can easily see what's important to them and what they can ignore.

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

Java uses three keywords to set the boundaries in a class: **public**, **private**, and **protected**. Their use and meaning are remarkably straightforward. These *access specifiers* determine who can use the definition that follows. **public** means the following definition is available to everyone. The **private** keyword, on the other hand, means no one can access that definition except you, the creator of the type, inside function members of that type. **private** is a brick wall between you and the client programmer. If someone tries to access a private member, they'll get a compile-time error. **protected** acts just like **private**, with the exception that inherited classes have access to **protected** members, but not **private** members. Inheritance will be discussed shortly.

reusing the implementation

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

The simplest way to reuse a class is to place an object of that class inside a new class: we call this "creating a member object." Your new class can be made up of any number and type of other objects, whatever is necessary to achieve the functionality desired in your new class. This concept is called *composition*, since you are composing a new class from existing classes. Sometimes composition is referred to as a "has-a" relationship, as in "a car has a trunk."

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

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

inheritance: reusing the interface

By itself, the concept of an object is a very convenient tool, since it allows you to package data and functionality together by *concept*, so you can represent an appropriate problem-space idea rather than being forced to use the idioms of the underlying machine. In addition, these concepts are expressed in the primary idea of the programming language: as a data type (using the **class** keyword).

However, it seems a pity to go to all the trouble to create a data type and then be forced to create a brand new one that might have very similar functionality. It would be nicer if we could take the existing data type, clone it and make additions and modifications to the clone. This is effectively what you get with *inheritance*, with the exception that if the original class (called the *base* or *super* or *parent* class) is changed, the modified "clone" (called the *derived* or *inherited* or *sub* or *child* class) also reflects the appropriate changes.

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

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

You have two ways to differentiate your new derived class from the original base class it inherits from. The first is quite straightforward: you simply add brand new functions to the derived class. These new functions are not part of the base class interface. This means that the base class simply didn't do as much as you wanted it to, so you add more functions. This very simple and primitive use for inheritance is, at times, the perfect solution to your problem. However, you should look closely for the possibility that your base class may need these additional functions.

The second way, discussed in the following section, is to *change* the behavior of an existing base-class function by *overriding* it.

overriding base-class functionality

Inheritance is implemented in Java with the **extends** keyword: you make a new class and you say that it **extends** an existing class. Although this implies that you are going to add new functions to the interface, that's not necessarily true. You may also want to *change* the behavior of an existing interface function: this is referred to as *overriding* that function.

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

is-a vs. is-like-a relationships

There's a certain debate that can occur about inheritance: should inheritance *only* override base-class functions? This means that the derived type is *exactly* the same type as the base class since it has exactly the same interface. As a result, you can exactly substitute an object of the derived class for an object of the base-class. This can be thought of as *pure substitution*. In a sense, this is the ideal way to treat inheritance. We often refer to the relationship between the base class and derived classes in this

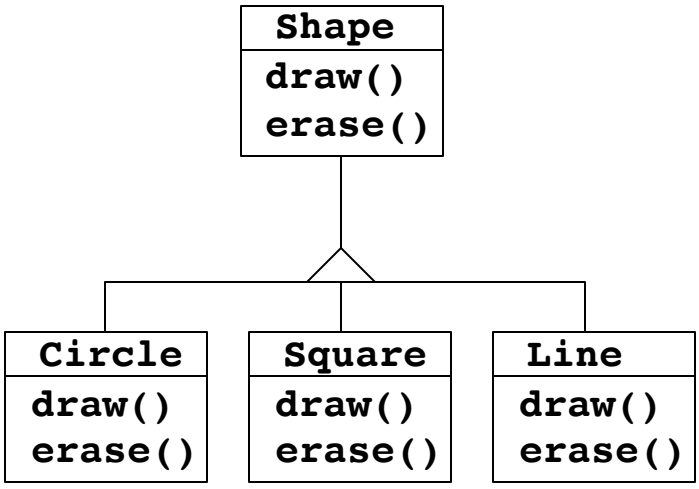
case as an *is-a* relationship, because you can say "a circle *is a* shape." A test for inheritance is whether you can state the is-a relationship about the classes and have it make sense.

However, there are times when you must add new interface elements to a derived type, thus extending the interface and creating a new type. The new type can still be substituted for the base type, but the substitution isn't perfect in a sense, since your new functions are not accessible from the base type. This can be described as an *is-like-a* relationship: the new type has the interface of the old type but it also contains other functions so you can't really say it's exactly the same.

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

interchangeable objects with polymorphism

Inheritance usually ends up creating a family of classes, all based on the same uniform interface. We express this with an inverted tree diagram[5]:



One of the most important things you do with such a family of classes is to treat an object of a derived class as an object of the base class. Why is this important? It means we can write a single piece of code that ignores the specific details of type, and just talks to the base class. That code is then *decoupled* from type-specific information, and thus is simpler to write and easier to understand. In addition, if a new type is added through inheritance, say a **Triangle**, the code you write will work just as well for the new type of **Shape** as it did on the existing types. Thus the program is *extensible*.

Consider the above example. If you write a function in Java:

```
void doStuff(Shape s) {
  s.erase();
  // ...
  s.draw();
}
```

[5] This uses the *Unified Notation*, which will primarily be used in this book.

This function is independent of the specific type of object it's drawing and erasing. If in some other program we use the **doStuff( )** function:

```
Circle c = new Circle();
Triangle t = new Triangle();
Line l = new Line();
doStuff(c);
doStuff(t);
doStuff(l);
```

The calls to **doStuff( )** work just right regardless of the exact type of the object.

This is actually a pretty amazing trick. Consider the line:

```
doStuff(c);
```

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

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

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

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

Not "if you're a **Circle**, do this, if you're a **Square**, do that, etc." Just "you're a shape, I know you can **erase( )** yourself, do it and take care of the details correctly." If you had to write code that checked for all the possible types a **Shape** could actually be, it would be messy and you'd have to change it every time you added a new kind of **Shape**.

dynamic binding

What's amazing about the code in **doStuff( )** is that somehow the right thing happens. Drawing a **Circle** causes different code to be executed than drawing a **Square** or a **Line**, but when the **draw( )** message is sent to an anonymous **Shape**, the correct behavior occurs based on the actual type that **Shape** handle happens to be connected to. This is amazing because when the Java compiler is compiling the code for **doStuff( )**, it cannot know what exact types it is dealing with. So ordinarily, you'd just expect it to end up calling the version of **erase( )** for **Shape**, and **draw( )** for **Shape**, and not for the specific **Circle**, **Square** or **Line**. And yet the right thing happens. How can this be?

When you send a message to an object even though you don't know what specific type it is, and the right thing happens, that's called *polymorphism*. The process used by object-oriented programming languages to implement polymorphism is called *dynamic binding*. The compiler and run-time system handle the details; all you need to know is that it happens and more importantly how to design with it.

Some languages require that you use a special keyword to enable dynamic binding. In C++ this keyword is **virtual**. In Java, you never have to remember to add a keyword, since all functions are automatically dynamically bound. So you can always expect that, when you send a message to an object, the object will do the right thing, even when upcasting is involved.

the abstract base class

Very often in a design, you want the base class to present *only* an interface for its derived classes. That is, you don't want anyone to actually create an object of the base class, only to upcast to it so that its interface can be used. This is accomplished by making that class *abstract* using the **abstract** keyword.

If anyone tries to make an object of an **abstract** class, the compiler prevents them. Thus this is a tool for design, to enforce a particular design.

You can also use the **abstract** keyword to describe a method that hasn't been implemented yet, as a stub saying "here is an interface function for all types inherited from this class, but I don't have any implementation for it at this point." An **abstract** method may only be created inside an **abstract** class. When the class is inherited, that method must be implemented, or the inherited class becomes **abstract** as well. Creating an **abstract** method allows you to put a method in an interface without being forced to provide a (possibly meaningless) body of code for that method.

object landscapes and lifetimes

Technically, OOP is just about abstract data typing, inheritance and polymorphism, but other issues can be at least as important. The remainder of this section will discuss these issues.

One of the most important factors concerns the way objects are created and destroyed: where is the data for an object and how is the lifetime of the object controlled? There are different philosophies at work here. C++ takes the approach that control of efficiency is the most important issue, so the programmer has a choice. For maximum run-time speed, the storage and lifetime can be determined while the program is being written, by placing the objects on the stack (these are sometimes called *automatic* or *scoped* variables) or in the static storage area. This places a priority on the speed of storage allocation and release, the control of which can be very valuable in some situations. However, you sacrifice flexibility: you must know the exact quantity, lifetime and type of objects *while* you're writing the program. If you are trying to solve a more general problem like computer-aided design, package management or air-traffic control, this is too restrictive.

The second approach is to create objects dynamically, in a pool of memory called the *heap*. In this approach you don't know until run time how many objects you need, what their lifetime is or what their exact type is. All that is determined at the spur of the moment while the program is running. If you need a new object, you simply make it on the heap at the point that you need it. Because the storage is managed dynamically, at run time, the amount of time required to allocate storage on the heap is significantly longer than creating storage on the stack (which is often a single assembly instruction to move the stack pointer down, and another to move it back up). The dynamic approach makes the generally logical assumption that objects tend to be complicated, so the extra overhead of finding storage and releasing that storage will not have an important impact on the creation of an object. In addition the greater flexibility is essential to solve the general programming problem.

C++ allows you to determine whether the objects are created while you write the program or at run time to allow the control of efficiency. You'd normally think that since it's more flexible, you'd always want to create objects on the heap rather than the stack. There's another issue, however, and that's the lifetime of an object. If you create an object on the stack or in static storage, the compiler determines how long the object lasts and can automatically destroy it. However, if you create it on the heap the compiler has no knowledge of its lifetime. How does the object get destroyed? This produces two more options: the programmer can determine programmatically when to destroy the object, or the environment can provide a process called a *garbage collector* that automatically discovers when an object is no longer in use and destroys it. Of course, a garbage collector is much more convenient, but it requires that all systems have some kind of multithreading support and that all applications be able to tolerate the existence of the garbage collector and the other overhead for garbage collection. This does not meet the design requirements of the C++ language and so it was not included.

Some languages, like Object Pascal (as seen in Delphi), Java and Smalltalk require that all objects be created on the heap, so there is no option for the optimization allowed in C++. These languages have narrower scopes of problems they can solve, but they provide an easier way to solve those problems. In addition, Java and Smalltalk have built-in garbage collectors (Delphi has the necessary wiring to easily add garbage collection, so it may happen sometime after this writing).

The rest of this section looks at additional factors concerning object lifetimes and landscapes.

containers and iterators

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

The solution to most problems in object-oriented design seems flippant: you create another type of object. The job of this object is to hold handles to other objects. Of course, you could do this with the array, which is available in most languages. But there's more: this new object, generally called a *container*, will expand itself whenever necessary to accommodate everything you place inside it. Thus you don't need to know how many objects you're going to hold in a container. Just create a container object and let it take care of the details.

Fortunately, a good OOP language comes with a set of containers as part of the package. In C++, it's the Standard Template Library (STL). Object Pascal has containers in its VCL. Java also has containers in its standard library. In some libraries, a generic container is considered good enough for all needs, and in others (C++ in particular) the library has different types of containers for different needs: a vector for consistent access to all elements, and a linked list for consistent insertion at all elements, for example, so you can choose the particular type that fits your needs. These may include sets, queues, hash tables, trees, stacks, etc.

All containers have in common some way to put things in and get things out. The way you place something into a container is fairly obvious: there's a function called "push" or "add" or a similar name. Fetching things out of a container is not always as apparent: if it's an array-like entity such as a vector, you may be able to use an indexing operator or function. But in many situations this doesn't make sense. In addition, a single-selection function is restrictive: what if you want to manipulate or compare a set of elements in the container instead of just one?

The solution is called an *iterator*, which is an object whose job is to select the elements within a container and present them to the user of the iterator. However, there's more to an iterator: as a class, it also provides a level of abstraction. This abstraction can be used to separate the details of the container from the code that's accessing that container. The container, via the iterator, is abstracted to be simply a sequence. The iterator allows you to traverse that sequence without worrying about the underlying structure – that is, whether it's a vector, a linked list, a stack, or something else. This gives you the flexibility to easily change the underlying data structure without disturbing the code in your program. Java provides a standard iterator (called **Enumeration**) for all its container classes.

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

But in the end, remember that a container is only a storage cabinet to put objects in. If that cabinet solves all your needs it doesn't really matter *how* it is implemented (a basic concept with most types of objects). If you're working in a programming environment that has built-in overhead due to other factors (running under Windows, for example, or the cost of a garbage collector), then the cost difference between a vector and a linked list may not matter, so you may only need one type of

sequence (the standard Java library makes this assumption: it only provides a vector). You could even imagine the "perfect" container abstraction, which could automatically change its underlying implementation according to the way it was used.

the singly-rooted hierarchy

One of the issues in OOP that has become especially prominent since the introduction of C++ is: should all classes be ultimately inherited from a single base class? In Java the answer is "yes" and the name of this ultimate base class is simply **Object**. It turns out that the benefits of the *singly-rooted hierarchy* are many.

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

All objects in a singly-rooted hierarchy such as Java provides can be guaranteed to have certain functionality. Thus you're guaranteed that you can perform certain basic operations on every object in your system.

It's possible to make all objects have the same size by forcing them to be created on the heap and passing them around as handles, instead of copying the object. This is the way Java works, and it greatly simplifies argument passing (one of the more complex topics in C++).

A singly-rooted hierarchy allows the implementation of a garbage collector. The necessary support can be installed in the base class, and the garbage collector can thus send the appropriate messages to every object in the system. Without a singly-rooted hierarchy and a system to manipulate an object via a handle, it is very difficult to implement a garbage collector.

Since run-time type information is guaranteed to be in all objects, you'll never end up with an object whose type you cannot determine. This is especially important with system level operations like exception handling, and to allow greater flexibility in programming.

So, if the use of a singly-rooted hierarchy is so beneficial, why isn't it in C++? It's the old bugaboo of efficiency and control. A singly-rooted hierarchy puts constraints on your program designs, and in particular it was perceived to put constraints on the use of existing C code. These constraints only cause problems in certain situations, but for maximum flexibility there is no requirement for a singly-rooted hierarchy in C++. In Java, which started from scratch and has no backward-compatibility issues with any existing language, it was a logical choice to use a singly-rooted hierarchy in following with most other object-oriented programming languages.

container libraries and support for easy container use

Since a container is a tool that you'll use on a very frequent basis, it makes sense to have a library of containers that are built in a reusable fashion, so you can take one off the shelf and plug it into your program. Java provides such a library, although it is fairly limited. And yet, it may satisfy most of your needs. More extensive libraries have been appearing on the Internet.

downcasting vs. Templates/Generics

To make these containers reusable, they contain the one universal type in Java that was previously mentioned: **Object**. Since the singly-rooted hierarchy means that everything is an **Object**, a container that holds **Object**s can hold anything. Thus it's easy to reuse.

To use such a container, you simply add object handles to it, and later ask for them back. But, since the container holds only **Object**s, when you add your object handle into the container it is upcast to **Object**, thus losing its identity. When you fetch it back out, you get an **Object** handle, and not a handle to the type that you put in. So how do you turn it back into something that has the useful interface of the object that you put into the container?

Here, the cast is used again, but this time you're not casting *up* the inheritance hierarchy to a more general type, but instead *down* the hierarchy to a more specific type. Therefore this manner of casting is called *downcasting*. But with upcasting, you know for example that a **Circle** is a type of **Shape** so it's safe to upcast, but you don't know that an **Object** is necessarily a **Circle** or a **Shape** so it's hardly safe to downcast unless you know that's what you're dealing with.

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

Downcasting and the run-time checks require extra time for the running program, and extra effort on the part of the programmer. Wouldn't it make sense to somehow create the container so that it knows the types that it holds, thus eliminating the need for the downcast and possible mistake? The solution is *parameterized types*, which are classes that the compiler can automatically customize to work with particular types. For example, with a parameterized container, the compiler could customize that container so it would only accept **Shape**s and fetch **Shape**s.

Parameterized types are an important part of C++ because C++ has no singly-rooted hierarchy. In C++, the keyword that implements parameterized types is **template**. Java currently has no parameterized types, since it is possible for it to get by – however awkwardly – using the singly-rooted hierarchy. At one point the word **generic** (the keyword used by ADA for its templates) was on a list of keywords that were "reserved for future implementation." Some of these seemed to have mysteriously slipped into a kind of "keyword Bermuda Triangle" and it's quite difficult to know what may eventually happen.

the housekeeping dilemma: who should clean up?

Each object requires resources in order to exist, most notably memory. When an object is no longer needed it must be cleaned up in order that these resources are released so they can be reused. In simple programming situations the question of how an object is cleaned up doesn't seem too challenging: you create the object, use it for as long as it's needed, and then it should be destroyed. However, it's not too hard to encounter situations where the situation is more complex.

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

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

Now the problem is more difficult: how can you possibly know when to destroy the objects? When you're done with the object, some other part of the system might not be. This same problem can arise

in a number of other situations, and in programming systems (like C++) where you must explicitly delete an object when you're done with it this can become quite complex[6].

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

garbage collectors vs. efficiency and flexibility

If all this is such a good idea, why didn't they do the same thing in C++? Well of course there's a price you pay for all this programming convenience, and its run-time overhead. As mentioned before, in C++ you can create objects on the stack, and in this case they're automatically cleaned up (but you don't have the flexibility of creating as many as you want at run-time). Creating objects on the stack is the most efficient way to allocate storage for objects, and also to free that storage. Creating objects on the heap is much more expensive. Always inheriting from a base class, and making all function calls polymorphic also exacts a toll. But the garbage collector is a particular problem, because you never quite know when it's going to start up nor how long it will take. This means that there's an inconsistency in the rate of execution of a Java program, so you can't use it in certain situations: where the rate of execution of a program is uniformly critical (these are generally called *real time* programs, although not all real-time programming problems are this stringent).

The designers of the C++ language, trying as they were to woo C programmers (and most successfully, at that), did not want to add any features to the language that would impact the speed or the use of C++ in any situation where C might be used. This goal was realized, but at the price of greater complexity when programming in C++. Java is simpler than C++, but the tradeoff is in efficiency and applicability. For a significant portion of programming problems, however, Java will often be the superior choice.

exception handling: dealing with errors

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

Exception handling wires error handling directly into the programming language itself (and sometimes even the operating system). An exception is an object that is "thrown" from the site of the error, and can be "caught" by an appropriate *exception handler* that is designed to handle that particular type of error. It's as if exception handling is a different, parallel path of execution that may be taken when things go wrong. And because it uses a separate execution path, it doesn't need to interfere with your normally-executing code, which makes that code simpler to write (since you aren't constantly forced to check for errors). In addition, a thrown exception is unlike an error value that's returned from a function, or a flag that's set by a function to indicate an error condition – these can be ignored. An exception cannot be ignored, thus it's guaranteed to be dealt with at some point. Finally, exceptions

[6] Note that this is only true for objects that are created on the heap, with **new**. However, the problem described, and indeed any general programming problem, requires objects to be created on the heap.

provide a way to reliably recover from a bad situation, so instead of just exiting you are often able to set things right and restore the execution of a program, which produces much more robust programs.

Java's exception handling stands out among programming languages, because in Java exception-handling was wired in from the beginning and you're *forced* to use it. If you don't write your code to properly handle exceptions, you'll get a compile-time error message. This guaranteed consistency makes error-handling a much easier issue to deal with.

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

multithreading

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

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

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

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

Java's threading is built into the language, which makes a complicated subject much simpler. The threading is supported on an object level, so one thread of execution is represented by one object. Java also provides limited resource locking: it can lock the memory of any object (which is, after all, one kind of shared resource) so that only one thread can use it at a time. This is accomplished with the **synchronized** keyword. Other types of resources must be locked explicitly by the programmer, typically by creating an object to represent the lock that all threads must check before accessing that resource.

persistence

When you create an object, it exists for as long as you need it, but under no circumstances does it exist when the program terminates. While this makes sense at first, there are situations where it would be incredibly useful if an object were to exist and hold its information even while the program *wasn't* running. Then the next time you started the program up, the object would be there and it would have the same information it had the previous time the program was running. Of course you can get a similar effect now by writing the information to a file or a database, but in the spirit of making

everything an object it would be quite convenient to be able to declare an object *persistent* and have all the details taken care of for you.

Java 1.1 provides support for "lightweight persistence," which means you can easily store objects on disk and later retrieve them. The reason it's "lightweight" is that you're still forced make explicit calls to do the storage and retrieval. In some future release more complete support for persistence may appear.

Java and the Internet

If Java is, in fact, yet another computer programming language, why is it so important and why is it being promoted as a revolutionary step in computer programming? The answer isn't immediately obvious if you're coming from a traditional programming perspective. Although Java will solve traditional stand-alone programming problems, the reason it is important is that it will also solve programming problems on the world-wide web ("the Web").

what is the Web?

The Web can seem a bit of a mystery at first, with all this talk of "surfing" and "presence" and "home pages." There has even been a growing reaction against "Internet-mania," questioning the economic value and outcome of such a sweeping movement. It's helpful to step back and see what it really is, but to do this you must understand client/server systems (another aspect of computing that's full of confusing issues).

client/server computing

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

The basic concept of client/server computing, then, is not so complicated. The problems arise because you have a single server trying to serve many clients at once. Generally a database management system is involved which that the designer "balance" the layout of data into tables for optimal use. In addition, systems often allow a client to insert new information into a server, and so you have the issue of making sure that one client's new data doesn't walk over another client's new data, or that data isn't lost in the process of adding it to the database (this is called *transaction processing*). As client software changes, it must be built, debugged and installed on the client machines, which turns out to be more complicated and expensive than you might think; it's especially problematic to support multiple types of computers and operating systems. Finally there's the all-important performance issue: you might have hundreds of clients making requests of your server at any one time, and so any small delay is crucial. To minimize latency, programmers work hard to offload processing tasks, often to the client machine but sometimes to other machines at the server site using so-called *middleware*.

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

the Web as a giant server

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

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

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

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

client-side programming[7]

The Web's initial server-browser design provided for interactive content, but the interactivity was completely provided by the server. The server produced static pages for the client browser, which would simply interpret and display them. Basic HTML contains very simple mechanisms for data gathering: text-entry boxes, check boxes, radio boxes, lists and drop-down lists, as well as a button which can be programmed to do only two things: reset the data on the form or "submit" the data on the form back to the server. This submission passes through the *Common Gateway Interface* (CGI) provided on all Web servers. The text within the submission tells CGI what to do with it; the most common action is to run a program located in the cgi-bin directory of the server (if you watch the address window at the top of your browser when you push a button on a Web page, you can sometimes see "cgi-bin" within all the gobbledygook there). These programs can be written in most languages, but Perl is a common choice because it is designed for text manipulation and is interpreted, and so can be installed on any server regardless of processor or operating system.

Many powerful Web sites today are built strictly on CGI, and you can in fact do nearly anything with it. The problem is response time. The response of a CGI program depends on how much data must be sent as well as the load on both the server and the Internet (on top of this, starting a CGI program tends to be slow). The initial designers of the Web did not foresee how rapidly this bandwidth would be exhausted for the kinds of applications people developed. For example, any sort of dynamic graphing is nearly impossible to perform with consistency, since a GIF file must be created and moved from the server to the client for each version of the graph. And you've no doubt had direct experience with something as simple as validating the data on an input form: you press the submit button on a page, the data is shipped back to the server which starts a CGI program that discovers an error, formats an HTML page informing you of the error and sends the page back to you, at which point you must back up a page and try again. Not only is this slow, it's inelegant.

---

[7] The material in this section is adapted from an article by the author that originally appeared on Mainspring, at www.mainspring.com. Used with permission.

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

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

plug-ins

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

scripting languages

Plug-ins resulted in an explosion of scripting languages. With a scripting language you embed the source code for your client-side program directly into the HTML page, and the plug-in that interprets that language is automatically activated while the HTML page is being displayed. Scripting languages tend to be reasonably simple to understand, and because they are simply text that is part of an HTML page they load very quickly, as part of the single server hit required to procure that page. The trade-off is that your code is exposed for everyone to see (and steal) but generally you aren't doing amazingly sophisticated things with scripting languages so it's not too much of a hardship.

This points out that scripting languages are really intended to solve specific types of problems, primarily the creation of richer and more interactive graphical user interfaces (GUIs). However, a scripting language may solve 80% of the kinds of problems encountered in client-side programming. Your problems may very well fit completely within that 80%, and since scripting languages tend to be easier and faster to develop you should probably consider a scripting language before looking at a more involved solution such as Java or ActiveX programming.

The most commonly-discussed scripting languages are JavaScript (nothing to do with Java; it's just named that way to grab some of Java's marketing momentum), VBscript (which looks like Visual Basic) and Tcl/Tk which comes from the popular cross-platform GUI-building language. There are others out there and no doubt more in development.

JavaScript is probably the most commonly supported; it comes built into both Netscape Navigator and the Microsoft Internet Explorer (IE). In addition, there are probably more JavaScript books out than for the others, and some tools automatically create pages using JavaScript. However, if you're already fluent in Visual Basic or Tcl/Tk, you'll be more productive using those scripting languages rather than learning a new one (you'll have your hands full dealing with the Web issues already).

Java

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

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

One advantage a Java applet has over a scripted program is that it's in compiled form, so the source code isn't available to the client. On the other hand, a Java applet can be decompiled without too much trouble, and hiding your code is often not an important issue anyway. Two other factors can be important: as you will see later in the book, a compiled Java applet can comprise many modules and take multiple server "hits" (accesses) to download (In Java 1.1 this is minimized by Java archives, called JAR files, that allow all the required modules to be packaged together for a single download). A scripted program will just be integrated into the Web page as part of its text (and will generally be smaller as well as not requiring any extra server hits). This may or may not be important to the responsiveness of your Web site. Finally, there's the all-important learning curve. Regardless of what you've heard, Java is not a trivial language to learn. If you're a Visual Basic programmer, moving to VBscript will be your fastest solution and since it will probably solve most typical client/server problems you may be hard pressed to justify learning Java. If you're experienced with a scripting language you will certainly benefit from looking at JavaScript or VBscript before committing to Java, since they may fit your needs handily and you'll be more productive sooner.

ActiveX

In effect, the competitor to Java is Microsoft's ActiveX, although it takes a completely different approach. ActiveX is originally a Windows-only solution, although it is now being developed via an independent consortium to become cross-platform. Effectively, ActiveX says "if your program connects to its environment just so, it can be dropped into a Web page and run under a browser that supports ActiveX" (IE directly supports ActiveX and Netscape does so using a plug-in). Thus, ActiveX does not constrain you to a particular language. If, for example, you're already an experienced Windows programmer using a language like C++, Visual Basic or Borland's Delphi, you can create ActiveX components with almost no changes to your programming knowledge. ActiveX also provides a path for the use of legacy code in your Web pages.

Security

Automatically downloading and running programs across the Internet can sound like a virus-builders dream. ActiveX especially brings up the thorny issue of security in client-side programming. If you click on a Web site, you may automatically download any number of things along with the HTML page: GIF files, script code, compiled Java code, and ActiveX components. Some of these are benign: GIF files can't do any harm, and scripting languages are generally very limited in what they can do. Java was also designed to run its applets within a "sandbox" of safety, which prevents it from writing to disk or accessing memory outside the sandbox.

ActiveX is at the opposite end of the spectrum. Programming with ActiveX is like programming Windows – you can do anything you want. So if you click on a page which downloads an ActiveX component, that component may cause damage to the files on your disk. Of course, any program that you load onto your computer using non-Internet means can do the same thing and viruses downloaded from BBSs have long been a problem, but the speed of the Internet amplifies the difficulty.

The solution seems to be "digital signatures," whereby code is verified to show who the author is. This is based on the idea that a virus works because its creator can be anonymous, so if you remove the

anonymity individuals will be forced to be responsible for their actions. This seems like a good plan because it allows programs to be much more functional, and I suspect it will in fact eliminate malicious mischief. However, if a program has an unintentional bug that's destructive it will still cause problems.

The Java approach is to prevent any of these problems from occurring via the sandbox. The Java interpreter that lives on your local Web browser examines the applet for any untoward instructions as the applet is being loaded. In particular, the applet cannot write files to disk nor erase files (one of the mainstays of the virus). Applets are generally considered to be very safe, and since this is essential for reliable client-server systems, any bugs that allow viruses are rapidly repaired (it's worth noting that the browser software actually enforces these security restrictions, and some browsers allow you to select different security levels to provide varying degrees of access to your system).

You may be skeptical of this rather draconian restriction against writing files to your local disk. What if you want to build a local database or save any other kind of data for later use, offline? The initial vision seemed to be that eventually everyone would be online to do anything important, but that was soon seen to be impractical (although low-cost "Internet appliances" may someday satisfy the needs of a significant segment of users). The solution is the "signed applet" which uses public-key encryption to verify that an applet does indeed come from where it claims it does. A signed applet can then go ahead and trash your disk, but the theory is that since you can now hold the applet creator accountable they won't do vicious things. Java 1.1 provides a framework for digital signatures so you will eventually be able to allow an applet to step outside the sandbox if necessary.

I think digital signatures have missed an important issue, which is the speed that people move around on the Internet. If you do in fact download a buggy program and it does something untoward, how long will it be before you discover the damage? It could be days or even weeks. And by then, how will you track down the program that's done it (and what good will it do at that point?).

Internet vs. Intranet

Since the Web is the most general solution to the client/server problem, it makes sense that you can use the same technology to solve a subset of the problem, in particular the classic client/server problem within a company. With traditional client/server approaches you have the problem of multiple different types of client computers, as well as the difficulty of installing new client software, both of which are handily solved with Web browsers and client-side programming. When Web technology is used this way, it is referred to as an *Intranet*. Intranets provide much greater security than the Internet, since you can physically control access to the servers within your company. In terms of training, it seems that once people understand the general concept of a browser it's much easier for them to deal with differences in the way pages and applets look, so the learning curve for new kinds of systems would seem to be reduced.

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

If you're running on an Intranet you may have a different set of constraints. It's not uncommon that all your machines could be Wintel platforms. On an Intranet, you're responsible for the quality of your own code, and can repair bugs when they're discovered. In addition, you may already have a body of legacy code that you've been using in a more traditional client/server approach, whereby you must physically install client programs every time you do an upgrade. The time wasted in this last activity is the most compelling reason to move to browsers since upgrades are invisible and automatic. If you are involved in such an Intranet, the most sensible approach to take is ActiveX rather than trying to re-code your programs in a new language.

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

server-side programming

This whole discussion has ignored the issue of server-side programming. What happens when you make a request of a server? Most of the time the request is simply "send me this file." Your browser then interprets the file in some appropriate fashion: as an HTML page, a graphic image, a Java applet, a script program, etc. A more complicated request to a server generally involves a database transaction. A common scenario involves a request for a complex database search, which the server then formats into an HTML page and sends to you as the result (of course, if the client has more intelligence via Java or a scripting language, the raw data can be sent and formatted at the client end, which will be faster and less load on the server). Or you may want to register your name in a database when joining a group, or place an order, which will involve changes to that database. These database requests must be processed via some code on the server side, which is generally referred to as *server-side programming*. Traditionally server-side programming has been performed using Perl and CGI scripts, but more sophisticated systems have been appearing, including Java-based Web servers that allow you to perform all your server-side programming in Java.

a separate arena: applications

Most of the brouhaha over Java has been about applets. But Java is actually a general-purpose programming language that can solve any type of problem, at least in theory. And as pointed out previously, there may be more effective ways to solve most client/server problems. When you move out of the applet arena (and simultaneously release the restrictions, such as the one against writing to disk) you enter the world of general-purpose applications that run standalone, without a Web browser, just like any ordinary program does. Here, Java's strength is not only in its portability, but its programmability. As you'll see throughout this book, Java has many features that allow you to create robust programs in a shorter period than with previous programming languages.

Be aware this is a mixed blessing, though. You pay for the improvements through slower execution speed. Although there is significant work going on in this area, it would appear that Java has built-in limitations that may make it inappropriate to solve certain types of programming problems.

online documentation

The Java language and libraries from Sun Microsystems (downloadable for free) come with documentation in electronic form, readable using a Web browser, and virtually every 3rd party implementation of Java has this or an equivalent documentation system. Almost all the books published on Java have duplicated this documentation. So you either already have it or you can get it for free, and unless necessary, this book will not repeat that documentation because (although the Sun documentation at this writing could only be described as "weak") you'll generally find it more useful to find the class descriptions with your Web browser than you will to look them up in a book (plus it will be up-to-date). Thus this book will provide extra descriptions of the classes only when it's necessary to supplement the documentation so you can understand a particular example.

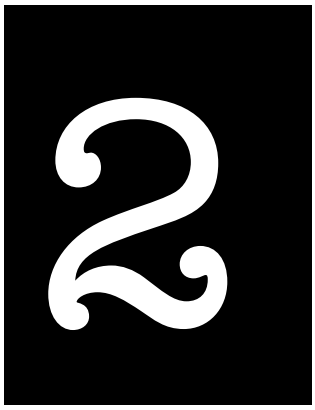
summary: Java vs. C++?

Should you use Java instead of C++ for your project? Other than Web applets, there are two issues to consider. First, if you want to use a lot of existing libraries (and you'll certainly get a lot of productivity gains there) or you have an existing C or C++ code base, then Java will probably slow you down rather than speed you up. If you're developing all your code primarily from scratch, then the simplicity of Java over C++ will shorten your development time.

The biggest issue is speed. Interpreted Java, which can be very slow, even on the order of 20-50 times slower than C in the original Java interpreters. This has improved quite a bit over time, but it will still remain an important number. Computers are about speed; if it wasn't significantly faster to do something on a computer then you'd do it by hand.

Thus the key to making Java feasible for most non-Web development projects is the appearance of speed improvements, possibly even native code compilers (two of which already exist at this writing). Of course, these will eliminate the touted cross-platform execution of the compiled programs, but they will also bring the speed of the executable closer to that of C and C++. In addition, cross-compiling programs in Java should be a lot easier than doing so in C or C++ (in theory, you just recompile, but that promise has been made before, for other languages).

You can find comparisons of Java and C++, observations about Java realities and practicality, and coding guidelines in the appendices.



2: everything is an object

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

Both C++ and Java are hybrid languages, but in Java the designers felt that the hybridization was not so important as it was in C++. A hybrid language allows multiple programming styles; the reason C++ is hybrid is to support backwards compatibility with the C language. Because C++ is a superset of the C language, it includes many of that language's undesirable features. The resulting language then becomes overly complicated and rife with impenetrable details.

The Java language assumes you want to do only object-oriented programming. This means that just to begin you must shift your mindset into an object-oriented world (unless it's already there). The benefit for this initial effort is the ability to program in a language that is simple to learn and to use. In this chapter we'll see the basic components of a Java program, and we'll learn that everything in Java is an object, even a Java program.

you manipulate objects through handles

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

All this is simplified in Java: you treat everything as an object, so there is a single consistent syntax that you use everywhere. Although you *treat* everything as an object, the identifier you manipulate is actually a "handle" (you may see this called a *reference* or even a pointer in other discussions of Java) to an object. You might imagine this scene as a television (the object) with your remote control as a handle. As long as you're holding this handle, you have a connection to the television, but when someone says "change the channel" or "lower the volume" what you're manipulating is the handle, which in turn modifies the object. If you want to move around the room and still control the television, you take the handle with you, not the whole television.

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

```
String s;
```

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

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

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

you must create all the objects

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

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

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

Of course, **String** is not the only type that exists: Java comes with a plethora of ready-made types. But what's more important is that you can create your own types. In fact, that's the fundamental activity in Java programming, and it's what you'll be learning about in the rest of the book.

where storage lives

It's useful to be able to visualize some aspects of the way things are laid out while the program is running, in particular how memory is arranged. There are 6 different places to store data:

1. **Registers**. This is the fastest of all storage because it exists in a different place than the other storage: inside the processor itself. However, the number of registers is severely limited and so registers are allocated by the compiler according to its needs and you

don't have direct control, nor do you see any evidence in your programs that registers even exist.

2. **The stack**. This lives in the general RAM (Random-access memory) area, but has direct support from the processor via its *stack pointer*. The stack pointer is moved down to create new memory and moved up to release that memory. This is an extremely fast and efficient way to allocate storage, slower only than registers. The Java compiler must know, while it is creating the program, the exact size and lifetime of all the data that is stored on the stack, because it must generate the code to move the stack pointer up and down. This constraint places limits on the flexibility of your programs, so while some Java storage exists on the stack – in particular, object handles – Java objects are not placed on the stack.

3. **The heap**. This is a general-purpose pool of memory (also in the RAM area) where all Java objects live. The nice thing about the heap is that, unlike the stack, the compiler doesn't need to know how much storage it needs to allocate from the heap or how long that storage must stay on the heap. Thus there's a great deal of flexibility in using storage on the heap. Whenever you need to create an object, you just write the code to create it using **new** and the storage is allocated on the heap right when that code is executed. And of course there's a price you pay for this flexibility: it takes more time to allocate heap storage.

4. **Static storage**. "Static" is used here in the sense of "in a fixed location" (although it's also in RAM). Static storage contains data that is available for the entire time a program is running. You can make certain elements of an object static, but Java objects themselves are never placed in static storage.

5. **Constant storage**. Constant values are often placed directly in the program code itself, which is safe since they can never change. Sometimes constants are cordoned off by themselves so they can be optionally placed in ROM – read-only memory.

6. **Non-RAM storage**. If data lives completely outside a program it can exist while the program is not running, outside the control of the program. The two primary examples of this are *streamed objects* where objects are turned into streams of bytes, generally to be sent to another machine, and *persistent objects* where the objects are placed on disk so they will hold their state even when the program is terminated. The trick with these types of storage is turning the objects into something that can exist on the other medium, and yet can be resurrected into a regular RAM-based object when necessary. Java 1.1 provides support for *lightweight persistence*, and future versions of Java may provide more complete solutions for persistence.•

special case: primitive types

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

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

Primitive type	Size	Minimum	Maximum	Wrapper type
boolean	1-bit	–	–	**Boolean**
char	16-bit	Unicode 0	Unicode 2^{16}- 1	**Character**

Primitive type	Size	Minimum	Maximum	Wrapper type
byte	8-bit	-128	+127	**Byte**[1]
short	16-bit	-2^{15}	$+2^{15}-1$	**Short**[1]
int	32-bit	-2^{31}	$+2^{31}-1$	**Integer**
long	64-bit	-2^{63}	$+2^{63}-1$	**Long**
float	32-bit	IEEE754	IEEE754	**Float**
double	64-bit	IEEE754	IEEE754	**Double**
void	–	–	–	**Void**[1]

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

Most of the primitive data types also have "wrapper" classes for them. That means if you want to make a class object on the heap to represent that primitive type, you use the associated wrapper. For example:

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

or you could also just say:

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

The reasons for doing this will be shown in a later chapter.

high-precision numbers

Java 1.1 has added two classes for performing high-precision arithmetic: **BigInteger** and **BigDecimal**. Although these approximately fit into the same category as the above "wrapper" classes, neither one has a primitive analogue.

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

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

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

Consult your on-line documentation for details about the constructors and methods you can call for these two classes.

arrays in Java

Virtually all programming languages support arrays. Using arrays in C and C++ is perilous because arrays are just blocks of memory, and if a program accesses the array outside of its memory block or

[1] In Java version 1.1 only, not in 1.0.

uses the memory before initialization (common programming errors) there will be unpredictable results.[2]

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

When you create an array of objects, you are really creating an array of handles, and each of those handles is automatically initialized to **null**. You must assign an object to each handle before you use it, and if you try to use a handle that's still **null** the problem will be reported at run-time. Thus, typical array errors are prevented in Java.

You can also create an array of primitives. Again, the compiler guarantees initialization because it forces you to provide initial values for those primitives.

Arrays will be covered in detail in later chapters.

you never have to destroy an object

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

scoping

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

```
{
  int x = 12;
  /* only x available */
  {
    int q = 96;
    /* both x & q available */
  }
  /* only x available */
  /* q "out of scope" */
}
```

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

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

Note that you *cannot* do the following, even though it is legal C and C++:

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

[2] In C++ you should often use the safer containers in the Standard Template Library as an alternative to arrays.

```
  {
    int x = 96; /* illegal */
  }
}
```

The compiler will announce that the variable **x** has already been defined. Thus the C/C++ ability to "hide" a variable in a larger scope is disallowed because the Java designers felt it led to confusing programs.

scope of objects

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

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

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

It turns out that because objects created with **new** stay around for as long as you want them, a whole slew of programming problems just vanish (in C++ and Java). The hardest problems seem to occur in C++ because you don't get any help from the language in making sure the objects are available when they're needed. And more importantly, in C++ you must make sure that you destroy the objects when you're done with them.

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

creating new data types: class

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

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

This introduces a new type , so you can now create an object of this type using **new**:

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

In **ATypeName**, the class body only consists of a comment (the stars and slashes and what is inside, which will be discussed later in this chapter) so there is not too much you can do with it. In fact, you cannot tell it to do much of anything (that is, you cannot send it any interesting messages) until you define some methods for it.

fields and methods

When you define a class (and all you do in Java is define classes, make objects of those classes, and send messages to those objects) you can put two types of elements in your class: data members (sometimes called *fields*) and member functions (typically called *methods*). A data member is an object (that you communicate with via its handle) of any type, or it can be one of the primitive types (which isn't a handle). If it is a handle to an object, you must initialize that handle to connect it to an actual object (using **new**, as seen earlier) in a special function called a *constructor* (described fully in Chapter 4). If it is a primitive type you can initialize it directly at the point of definition in the class. (As you'll see later, handles may also be initialized at the point of definition).

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

```
class DataOnly {
  int i;
  float f;
  boolean b;
}
```

This class doesn't *do* anything, but you can create an object:

```
DataOnly d = new DataOnly();
```

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

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

It is also possible that your object may contain other objects for which you would like to call a function. For this, you just keep "connecting the dots." The most common example is printing:

```
System.out.println("Howdy!");
```

This means "select **out** that lives inside of **System** and call the **println( )** method."

The **DataOnly** class cannot do much of anything except hold data, because it has no member functions (methods). To understand how those work, you must first understand *arguments* and *return values*.

default values for primitive members

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

Primitive type	Default
boolean	**false**
char	'\u000' (null)
byte	(byte)0
short	(short)0
int	0
long	0L

Primitive type	Default
float	0.0f
double	0.0d

Note carefully that the default values are what Java guarantees when the variable is used *as a member of a class*. This ensures that member variables of primitive types will always be initialized (something C++ doesn't do), reducing a source of bugs.

However, this guarantee doesn't apply to "local" variables – those that are not fields of a class. Thus, if within a function definition you have:

```
int x;
```

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

methods, arguments, and return values

Up until now, the term *function* has been used to describe a named subroutine. However, the term that is more commonly used in Java is *method* as in "a way to do something." If you want, you can go on thinking in terms of functions. It's really only a syntactic difference, but from now on "method" will be used in this book rather than "function."

Methods in Java determine the messages an object can receive. In this section you will learn how simple it is to define a method.

The fundamental parts of a method are the name, the arguments, the return type, and the body. Here is the basic form:

```
returnType methodName( /* argument list */ ) {
  /* Method body */
}
```

The return type is the type of the value that pops out of the method after you call it. The method name, as you might imagine, identifies the method. The argument list gives the types and names for the information you want to pass into the method.

Methods in Java can only be created as part of a class. A method can only be called for an object[3], and that object must be able to perform that method call. If you try to call the wrong method for an object, you'll get an error message at compile time. You call a method for an object by naming the object followed by a period (dot), followed by the name of the method and its argument list. For example, suppose you have a method **f( )** that takes no arguments and returns a value of type **int**. Then, if you have an object called **a** for which **f( )** can be called, you can say this:

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

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

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

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

the argument list

The method argument list specifies what information you pass into the method. As you might guess, this information – like everything else in Java – takes the form of objects. So, what you must specify in the argument list are the types of the objects to pass in and the name to use for each one. As in any situation in Java where you seem to be handing objects around, you are actually passing handles[4]. The type of the handle must be correct, however: if the argument is supposed to be a **String**, what you pass in must be a string.

Consider a method which takes a string as its argument. Here is the definition, which must be placed within a class definition for it to compile:

```
int size(String s) {
  return s.length();
}
```

The argument is of type **String** and is called **s**. Once **s** is passed into the method, you can treat it just like any other object (you can send messages to it). Here, the **length( )** method is called, which is one of the methods for **string**s – it returns the number of characters in a string.

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

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

At this point, it can look like a program is just a bunch of objects with methods that take other objects as arguments, and send messages to those other objects. That is indeed much of what goes on, but in the following chapter you'll learn how to do the detailed low-level work by making decisions within a method. But for this chapter, sending messages will suffice.

building a Java program

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

name visibility

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

Java was able to avoid all this by taking a fresh approach. To produce an unambiguous name for a library, the specifier used is not unlike an Internet domain name; in fact, the Java creators want you to use your Internet domain name in reverse since those are guaranteed to be unique. Since my domain

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

name is **EckelObjects.com**, then my utility library of foibles would be named **COM.EckelObjects.utility.foibles**. The COM extension is capitalized by convention. After your reversed domain name the dots are intended to represent subdirectories.

This mechanism in Java means that all your files automatically live in their own namespaces, and each class within a file automatically has a unique identifier (class names within a file must be unique, of course). Thus you do not need to learn special language features to solve this problem – the language takes care of it for you.

using other components

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

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

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

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

```
import java.util.Vector;
```

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

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

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

the "static" keyword

Normally, when you create a class you are describing how objects of that class look and how they will behave. You don't actually get anything until you create an object of that class with **new**, and at that point data are created and methods become available.

But there are two situations where this approach is not sufficient. What if you only want to have one piece of storage for a particular piece of data, regardless of how many objects are created, or even if no objects at all are created? And similarly, what if you need a method that isn't associated with any particular object of this class? That is, a method that you can call even if no class objects are created. Both these effects are achieved with the **static** keyword. When you say something is **static**, it means that data or method is not tied to any particular object instance of that class. Thus, even if you've never created an object of that class you can call a **static** method or access a piece of **static** data. With ordinary, non-**static** data and methods you must create an object, and use that object, to access the data or method since non-**static** data and methods must know the particular object they are working with. Of course, since **static** methods don't need any objects to be created before they are used, they *cannot* access non-**static** members or methods (since non-**static** members and methods must be tied to a particular object).

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

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

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

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

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

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

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

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

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

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

```
class StaticFun {
  static void incr() { StaticTest.i++; }
}
```

You can see that the **StaticFun** method **incr( )** increments the **static** data **i**. You can call **incr( )** in the typical way, through an object:

```
StaticFun sf = new StaticFun();
sf.incr();
```

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

```
StaticFun.incr();
```

While **static**, when applied to a data member, definitely changes the way the data is created (only one vs. the non-**static** one for each object), when applied to a method it's not so dramatic. The use of **static** for methods is to allow you to call that method without creating an object. This is essential, as we shall see, in defining the **main( )** method which is the entry point for running an application.

your first Java program

Finally, here's the program. It prints out information about the system that it's running on using various methods of the **System** object from the Java standard library. Note that an additional style of comment is introduced here: the '//' which is a comment until the end of the line:

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

public class Property {
  public static void main(String args[]) {
    System.out.println(new Date());
    System.getProperties().list(System.out);
    System.out.println("--- Memory Usage:");
```

```
    Runtime rt = Runtime.getRuntime();
    System.out.println("Total Memory = "
                       + rt.totalMemory()
                       + " Free Memory = "
                       + rt.freeMemory());
  }
}
```

At the beginning of each program file, you must place the **import** statement to bring in any extra classes you'll need for the code in that file. Notice that I said "extra." That's because there's a certain library of classes that are automatically brought into every java file: **java.lang**. Start up your web browser and look at the documentation from Javasoft (if you haven't downloaded it from www.Javasoft.com or otherwise installed the Java documentation, do so now). If you look at the **packages.html** file, you'll see a list of all the different class libraries that come with Java. Select **java.lang**. Under "Class Index" you'll see a list of all the classes that are part of that library. Since **java.lang** is implicitly included in every Java code file, these classes are automatically available. In the list, you'll see **System** and **Runtime**, which are used in **Property.java**. However, there's no **Date** class listed in **java.lang**, which means you'll need to import another library to use that. If you don't know the library where a particular class is (or if you want to see all the classes) you can select "Class Hierarchy" in the Java documentation. In a web browser, this takes a while to construct, but you can find every single class that comes with Java. Then you can use the browser's "find" function to find **Date**, and when you do you'll see it listed as **java.util.date**, which tells you it's in the **util** library and that you must **import java.util.*** in order to use **Date**.

Again looking at the documentation starting from the **packages.html** file (which I've set in my web browser as the starting page), if you select **java.lang** and then **System**, you'll see that the **System** class has several fields, and if you select **out** you'll discover that it's a **static PrintStream** object. Since it's **static** you don't have to create anything, the **out** object is always there and you can just use it. But what can you do with this **out** object? That is determined by what type it is – it's a **PrintStream**. Conveniently, **PrintStream** is shown in the description as a hyperlink, so if you click on that you'll see a list of all the methods you can call for **PrintStream**. There are quite a few and these will be covered later in the book, but for now all we're interested in is **println( )**, which in effect means "print out what I'm giving you to the console, and end with a new line). Thus in any Java program you write you can just say **System.out.println("things")** whenever you want to print things to the console.

The name of the class is the same as the name of the file. When you're creating a stand-alone program like this one, one of the classes in the file must have the same name as the file (the compiler complains if you don't do this) and that class must contain a method called **main( )** with the signature shown:

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

The **public** keyword means the method is available to the outside world (described in detail in Chapter 5). The argument to **main( )** is an array of **String** objects. The **args** won't be used in this program, but they have to be there because they hold the arguments invoked on the command line.

The first line of the program is quite interesting:

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

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

The second line calls **System.getProperties( )**. Again consulting the on-line documentation using your web browser, you'll see that **getProperties( )** is a **static** method of class **System**. Since it's **static**, you don't have to create any objects in order to call the method; a **static** method is always available whether an object of its class exists or not. When you call **getProperties( )**, it produces the system properties as an object of class **Properties**. **Properties**, in turn, has a method called **list( )** that sends its entire contents to a **PrintStream** object that you pass as an argument.

The third and fifth lines in **main( )** are typical print statements. Notice that to print multiple **String** values, we simply separate them with '**+**' signs. However, there's something strange going on here: the '**+**' sign doesn't mean "addition" when it's used with **String** objects. Normally you wouldn't ascribe any meaning at all to '**+**' when you think of strings. However, the Java **String** class is blessed with something called "operator overloading." That is, the '**+**' sign, only when used with **String** objects, behaves differently than it does with everything else. For **String**s, it means: "concatenate these two strings."

But that's not all. If you look at the statement:

```
System.out.println("Total Memory = "
                   + rt.totalMemory()
                   + " Free Memory = "
                   + rt.freeMemory());
```

totalMemory( ) and **freeMemory( )** return *numerical values*, and not **String** objects. What happens when you "add" a numerical value to a **String**? Well, the compiler sees the problem and magically calls a method that turns that numerical value (**int**, **float**, etc.) into a **String**, which can then be "added" with the plus sign. This *automatic type conversion* also falls into the category of operator overloading.

Much of the Java literature states vehemently that operator overloading (a feature in C++) is bad, and yet here it is! However, this is wired into the compiler, only for **String** objects, and you can't overload operators for any of the code you write.

The fourth line in **main( )** creates a **Runtime** object by calling the **static** method **getRuntime( )** for the class **Runtime**. What's returned is a handle to a **Runtime** object; whether this is a static object or one created with **new** doesn't need to concern you, since you only need to use the objects and not worry about who's responsible for cleaning them up. As shown, the **Runtime** object can tell you information about memory usage.

comments & embedded documentation

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

```
/* This is
*  A comment that continues
*  Across lines
*/
```

Remember, however, that everything inside the /* and */ is ignored so it's no different to say:

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

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

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

comment documentation

One of the thoughtful parts of the Java language is that the designers didn't just think about writing code, they also thought about documenting it. Possibly the biggest problem with documenting code has been maintaining that documentation. If the documentation and the code are separate, it becomes a hassle to change the documentation every time you change the code. The solution seems simple: link the code to the documentation. The easiest way to do this is to put everything in the same file. To complete the picture, however, you need a special comment syntax to mark special documentation and a tool to extract those comments and put them in a useful form. This is what Java has done.

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

The output of javadoc is an HTML file that you can view with your Web browser. This tool allows you to create and maintain a single source file and automatically generate useful documentation. Because of javadoc we have a standard for creating documentation, and it's easy enough that we can expect or even demand documentation with all Java libraries.

syntax

All of the javadoc commands occur only within /** comments. The comments end with */ as usual. There are two primary ways to use javadoc: embed HTML, or use "doc tags." Doc tags are commands that start with a '@' and are placed at the beginning of a comment line (a leading '*', however, is ignored).

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

```
/** A class comment */
public class docTest {
  /** A variable comment */
  public int i;
  /** A method comment */
  public void f() {}
}
```

Note that javadoc will process comment documentation for **public** and **protected** members only. Comments for **private** and "friendly" (see Chapter 5) members are ignored and you'll see no output. This makes sense, since only **public** and **protected** members are available outside the file, which is the client programmer's perspective. However, all **class** comments are included in the output.

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

embedded HTML

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

```
/**
* <pre>
* System.out.println(new Date());
```

```
* </pre>
*/
```

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

```
/**
* You can <em>even</em> insert a list:
* <ol>
* <li> Item one
* <li> Item two
* <li> Item three
* </ol>
*/
```

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

All types of comment documentation: class, variable and method, can support embedded HTML.

@see: referring to other classes

All three types of comment documentation can contain **@see** tags, which allow you to refer to the documentation in other classes. javadoc will generate HTML with the **@see** tags hyperlinked to the other documentation. The forms are:

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

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

class documentation tags

Along with embedded HTML and **@see** references, class documentation can also include tags for version information and the author's name. Class documentation can also be used for *interfaces* (described later in the book).

@version

This is of the form:

```
@version version-information
```

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

@author

This is of the form:

```
@author author-information
```

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

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

variable documentation tags

Variable documentation can only include embedded HTML and **@see** references.

method documentation tags

As well as embedded documentation and **@see** references, methods allow documentation tags for parameters, return values, and exceptions:

@param

This is of the form:

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

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

@return

This is of the form:

```
@return description
```

where **description** tells you the meaning of the return value. It can continue on subsequent lines.

@exception

Exceptions will be described in Chapter 9, but briefly they are objects that can be "thrown" out of a method if that method fails. Although only one exception object can emerge when you call a method, a particular method may produce any number of different types of exceptions, all of which need descriptions. So the form for the exception tag is:

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

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

@deprecated

This is new in Java 1.1. It is used to tag features which have been superseded by an improved feature. The deprecated tag is a suggestion that you no longer use this particular feature, since sometime in the future it is likely to be removed from the language.

documentation example

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

```
//: Property.java
import java.util.*;

/** The first example program in "Thinking in Java."
 * Lists system information on current machine.
 * @author Bruce Eckel
 * @author http://www.EckelObjects.com/Eckel
 * @version 1.0
*/
public class Property {
  /** Sole entry point to class & application
   * @param args Array of string arguments
   * @return No return value
   * @exception No exceptions are thrown
```

```
   */
   public static void main(String args[]) {
     System.out.println(new Date());
     System.getProperties().list(System.out);
     System.out.println("--- Memory Usage:");
     Runtime rt = Runtime.getRuntime();
     System.out.println("Total Memory = "
                        + rt.totalMemory()
                        + " Free Memory = "
                        + rt.freeMemory());
   }
} ///:~
```

The first line:

```
    //: Property.java
```

uses my own technique of putting a '**:**' as a special marker for the comment line containing the source file name. The last line also finishes with a comment, and this one indicates the end of the source-code listing, which allows it to be automatically extracted from the text of the book and checked with a compiler. This is described in detail in Chapter 17.

coding style

The unofficial standard in Java is to capitalize the first letter of a class name. If the class name consists of several words, they are run together (that is, you don't use underscores to separate the names) and the first letter of each embedded word is capitalized, such as:

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

For almost everything else: methods, fields (member variables) and object handle names, the accepted style is just like for classes *except* that the first letter of the identifier is lower case. For example:

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

Of course, you should remember that the user must also type all these long names, and be merciful.

summary

In this chapter you have seen enough of Java programming to understand how to write a simple program, and you have gotten an overview of the language and some of its basic ideas. However, the examples so far have all been of the form "do this, then do that, then do something else." What if you want the program to make choices, such as "if the result of doing this is red, do that, otherwise do something else?" The support in Java for this fundamental programming activity will be covered in the next chapter.

exercises

1. Following the first example in this chapter, create a “Hello, World” program that simply prints out that statement. You need to create only a single method in your class (the “main” one that gets executed when the program starts). Remember to make it **static** and to put the argument list in, even though you don’t use the argument list. Compile the program with **javac** and run it using **java**.

2. Write a program that prints three arguments taken from the command line.

3. Find the code for the second version of **Property.java**, which is the simple comment documentation example. Execute **javadoc** on the file and view the results with your Web browser.

4. Take the program in Exercise 1 and add comment documentation to it. Extract this comment documentation into an HTML file using **javadoc** and view it with your Web browser.

3: controlling program flow

Like a sentient creature, a program must manipulate its world and make choices during execution.

In Java you manipulate objects and data using operators, and you make choices with execution control statements. As Java was inherited from C++, most of these statements and operators will be familiar to C and C++ programmers. Java has also added some improvements and simplifications.

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. You should be reasonably comfortable with the general concept of operators from your previous programming experience. Addition (**+**), 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. Additionally, an operator can 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.

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 to remember is that multiplication and division happen before addition and subtraction. The other precedence rules are often forgotten by programmers, so you should use parentheses to make the order of evaluation explicit. For example:

```
A = X + Y - 2/2 + Z;
```

has a very different meaning from the same statement with a particular grouping of parentheses:

```
A = X + (Y - 2)/(2 + Z);
```

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 can produce a value, but an lvalue must be a distinct, named variable (that is, there must be a physical space to store a value). For instance, you can assign a constant value to a variable (**A = 4;**), but you cannot assign anything to constant value — it cannot be an lvalue (you can't say **4 = A;**).

mathematical operators

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

Java also uses a shorthand notation to perform 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**, you say: **x += 4;**.

This example shows the use of the mathematical operators:

```
//: MathOps.java
// Demonstrates the mathematical operators
import java.util.*;

public class MathOps {
  // Create a shorthand to save typing:
  static void prt(String s) {
    System.out.println(s);
  }
  // shorthand to print a string and an int:
  static void pInt(String s, int i) {
    prt(s + " = " + i);
  }
  // shorthand to print a string and a float:
  static void pFlt(String s, float f) {
    prt(s + " = " + f);
  }
  public static void main(String args[]) {
    // Create a random number generator,
    // seeds with current time by default:
    Random rand = new Random();
    int i, j, k;
```

```
    // '%' limits size to 100:
    j = rand.nextInt() % 100;
    k = rand.nextInt() % 100;
    pInt("j",j);  pInt("k",k);
    i = j + k; pInt("j + k", i);
    i = j - k; pInt("j - k", i);
    i = k / j; pInt("k / j", i);
    i = k * j; pInt("k * j", i);
    i = k % j; pInt("k % j", i);
    j %= k; pInt("j %= k", j);
    // Floating-point number tests:
    float u,v,w;  // applies to doubles, too
    v = rand.nextFloat();
    w = rand.nextFloat();
    pFlt("v", v); pFlt("w", w);
    u = v + w; pFlt("v + w", u);
    u = v - w; pFlt("v - w", u);
    u = v * w; pFlt("v * w", u);
    u = v / w; pFlt("v / w", u);
    // the following also works for
    // char, byte, short, int, long,
    // and double:
    u += v; pFlt("u += v", u);
    u -= v; pFlt("u -= v", u);
    u *= v; pFlt("u *= v", u);
    u /= v; pFlt("u /= v", u);
  }
} ///:~
```

The first thing you will see are some shorthand methods for printing: the **prt( )** method prints a **String**, the **pInt( )** prints a **String** followed by an **int**, and the **pFlt( )** prints a **String** followed by a **float**. Of course, they all ultimately end up using **System.out.println( )**.

To generate numbers the program first creates a **Random** object. Because no arguments are passed during creation, Java uses the current time as a seed for the random number generator . The program generates a number of different types of random numbers with the **Random** object simply by calling different methods: **nextInt( )**, **nextLong( )**, **nextFloat( )** or **nextDouble( )**.

The modulus operator, when used with the result of the random number generator, limits the result to an upper bound (100, in this case).

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 usage 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 clearer to say:

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

The unary minus produces the negative of the value. Unary plus provides symmetry with unary minus, although it doesn't do much.

auto increment and decrement

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

Two of the nicer shortcuts are the increment and decrement operators (often referred to as the auto-increment and auto-decrement operators). The decrement operator is **--** and means "decrease by one unit." The increment operator is **++** and means "increase by one unit." If **A** is an **int**, for example, the expression **++A** is equivalent to (**A = A + 1**). Increment and decrement operators 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 or expression, and post-increment means the **++** operator appears after the variable or expression. Similarly, pre-decrement means the **--** operator appears before the variable or expression, and post-decrement means the **--** operator appears after the variable or expression. 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:

```
//: AutoInc.java
// Demonstrates the ++ and -- operators

public class AutoInc {
  public static void main(String args[]) {
    int i = 1;
    prt("i : " + i);
    prt("++i : " + ++i); // Pre-increment
    prt("i++ : " + i++); // Post-increment
    prt("i : " + i);
    prt("--i : " + --i); // Pre-decrement
    prt("i-- : " + i--); // Post-decrement
    prt("i : " + i);
  }
  static void prt(String s) {
    System.out.println(s);
  }
} ///:~
```

The output for this program is:

```
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 (that is, they change the operand rather than just using its value).

The increment operator is one explanation for the name C++, implying "one step beyond C." So Java could be thought of as 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 works with all built-in data types, but the other comparisons won't work with type **boolean**.

logical operators

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

```
//: Bool.java
// Relational and logical operators
import java.util.*;

public class Bool {
  public static void main(String args[]) {
    Random rand = new Random();
    int i = rand.nextInt() % 100;
    int j = rand.nextInt() % 100;
    prt("i = " + i);
    prt("j = " + j);
    prt("i > j is " + (i > j));
    prt("i < j is " + (i < j));
    prt("i >= j is " + (i >= j));
    prt("i <= j is " + (i <= j));
    prt("i == j is " + (i == j));
    prt("i != j is " + (i != j));

    // Treating an int as a boolean is
    // not legal Java
//! prt("i && j is " + (i && j));
//! prt("i || j is " + (i || j));
//! prt("!i is " + !i);

    prt("(i < 10) && (j < 10) is "
       + ((i < 10) && (j < 10)) );
    prt("(i < 10) || (j < 10) is "
       + ((i < 10) || (j < 10)) );
  }
  static void prt(String s) {
    System.out.println(s);
  }
} ///:~
```

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 //**!** comment marker. The subsequent expressions, however, produce **boolean** values using relational comparisons, then use logical operations on the results.

One output listing looked like this:

```
i = 85
j = 4
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 true
```

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

You can replace the definition for **int** in the above 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.

short-circuiting

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

```
//: ShortCircuit.java
// Demonstrates short-circuiting behavior
// with logical operators.

public class ShortCircuit {
  static boolean test1(int val) {
    System.out.println("test1(" + val + ")");
    System.out.println("result: " + (val < 1));
    return val < 1;
  }
  static boolean test2(int val) {
    System.out.println("test2(" + val + ")");
    System.out.println("result: " + (val < 2));
    return val < 2;
  }
  static boolean test3(int val) {
    System.out.println("test3(" + val + ")");
    System.out.println("result: " + (val < 3));
    return val < 3;
  }
  public static void main(String args[]) {
    if(test1(0) && test2(2) && test3(2))
      System.out.println("expression is true");
    else
      System.out.println("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:

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

You might naturally think that all three tests would be executed, but the output shows otherwise:

```
test1(0)
result: true
test2(2)
result: false
expression is false
```

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 could be expensive. The reason for short-circuiting, in fact, is precisely that: you can get a potential performance increase if all the parts of a logical expression do not need to be evaluated.

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; you were often manipulating hardware directly and had to set the bits in hardware registers. Java was originally designed to be embedded in TV set-top boxes and so this low-level orientation still made sense. However, you probably won't use the bitwise operators that 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 only produces a zero 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 only takes 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.

Since the bitwise operators and logical operators use the same characters, it is helpful to have a mnemonic device to help you remember the meanings: since 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). You're also prevented from using **boolean**s in shift expressions (described next).

shift operators

The shift operators also manipulate bits. They can only be used on primitive, integral types. The left-shift operator (**<<**) produces the operand to the left of the operator shifted to the left by the number of bits specified after the operator (inserting zeroes at the lower-order bits). The signed right-shift operator (**>>**) produces the operand to the left of the operator shifted to the right by the number of bits specified after 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**, **long** will be the 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.

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

```
//: BitManipulation.java
// Using the bitwise operators
import java.util.*;

public class BitManipulation {
  public static void main(String args[]) {
    Random rand = new Random();
    int i = rand.nextInt();
    int j = rand.nextInt();
```

```
    pBinInt("-1", -1);
    pBinInt("+1", +1);
    int maxpos = 2147483647;
    pBinInt("maxpos", maxpos);
    int maxneg = -2147483648;
    pBinInt("maxneg", maxneg);
    pBinInt("i", i);
    pBinInt("~i", ~i);
    pBinInt("-i", -i);
    pBinInt("j", j);
    pBinInt("i & j", i & j);
    pBinInt("i | j", i | j);
    pBinInt("i ^ j", i ^ j);
    pBinInt("i << 5", i << 5);
    pBinInt("i >> 5", i >> 5);
    pBinInt("(~i) >> 5", (~i) >> 5);
    pBinInt("i >>> 5", i >>> 5);
    pBinInt("(~i) >>> 5", (~i) >>> 5);

    long l = rand.nextLong();
    long m = rand.nextLong();
    pBinLong("-1L", -1L);
    pBinLong("+1L", +1L);
    long ll = 9223372036854775807L;
    pBinLong("maxpos", ll);
    long lln = -9223372036854775808L;
    pBinLong("maxneg", lln);
    pBinLong("l", l);
    pBinLong("~l", ~l);
    pBinLong("-l", -l);
    pBinLong("m", m);
    pBinLong("l & m", l & m);
    pBinLong("l | m", l | m);
    pBinLong("l ^ m", l ^ m);
    pBinLong("l << 5", l << 5);
    pBinLong("l >> 5", l >> 5);
    pBinLong("(~l) >> 5", (~l) >> 5);
    pBinLong("l >>> 5", l >>> 5);
    pBinLong("(~l) >>> 5", (~l) >>> 5);
  }
  static void pBinInt(String s, int i) {
    System.out.println(
      s + ", int: " + i + ", binary: ");
    System.out.print("   ");
    for(int j = 31; j >=0; j--)
      if(((1 << j) &  i) != 0)
        System.out.print("1");
      else
        System.out.print("0");
    System.out.println();
  }
  static void pBinLong(String s, long l) {
    System.out.println(
      s + ", long: " + l + ", binary: ");
    System.out.print("   ");
    for(int i = 63; i >=0; i--)
      if(((1L << i) & l) != 0)
        System.out.print("1");
      else
        System.out.print("0");
```

```
    System.out.println();
  }
} ///:~
```

The two methods at the end, **pBinInt( )** and **pBinLong( )** take an **int** or a **long**, respectively, and print it out in binary format along with a descriptive string. You can ignore the implementation of these for now.

You'll notice the use of **System.out.print( )** instead of **System.out.println( )**. The **print( )** method does not put out a newline, so it allows you to output a line in pieces.

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 looks like this:

```
-1, int: -1, binary:
   11111111111111111111111111111111
+1, int: 1, binary:
   00000000000000000000000000000001
maxpos, int: 2147483647, binary:
   01111111111111111111111111111111
maxneg, int: -2147483648, binary:
   10000000000000000000000000000000
i, int: 59081716, binary:
   00000011100001011000001111110100
~i, int: -59081717, binary:
   11111100011110100111110000001011
-i, int: -59081716, binary:
   11111100011110100111110000001100
j, int: 198850956, binary:
   00001011110110100011100110001100
i & j, int: 58720644, binary:
   00000011100000000000000110000100
i | j, int: 199212028, binary:
   00001011110111111011101111111100
i ^ j, int: 140491384, binary:
   00001000010111111011101001111000
i << 5, int: 1890614912, binary:
   01110000101100000111111010000000
i >> 5, int: 1846303, binary:
   00000000000111000010110000011111
(~i) >> 5, int: -1846304, binary:
   11111111111000111101001111100000
i >>> 5, int: 1846303, binary:
   00000000000111000010110000011111
(~i) >>> 5, int: 132371424, binary:
   00000111111000111101001111100000
```

The binary representation of the numbers is referred to as *signed two's complement*.

ternary if-else operator

This operator is unusual because it has three operands. It is truly an operator because it produces a value, unlike the ordinary if-else statement which 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 prides itself on being a terse language and the ternary operator may 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 can be used for its side effects or for the value it produces, but generally you want the value since that's what makes the operator distinct from the **if-else**. Here's an example:

```
static int ternary(int i) {
  return i < 10 ? i * 100 : i * 10;
}
```

You can see that this code is more compact than what you'd have to write without the ternary operator:

```
static int alternative(int i) {
  if (i < 10)
    return i * 100;
  return i * 10;
}
```

The second form is easier to understand, and doesn't require a lot more typing.

the comma operator

The comma is used in C and C++ not only as a separator in function argument lists, but also as an operator for sequential evaluation. The only place that the comma *operator* is used in Java is in **for** loops, which will be described later in this chapter.

String operator +

There's one special usage of an operator in Java: the **+** operator can be used to concatenate strings, as you've already seen. It seems a very natural use of the **+** even though it doesn't fit with the traditional way that **+** is used. This capability seemed like a good idea in C++ and 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++, this feature was still considered too complex and thus is not part of Java.

The use of the **String +** has some interesting behavior: if an expression begins with a **String**, then all operands that follow must be **String**s:

```
int x = 0, y = 1, z = 2;
String sString = "x, y, z ";
System.out.println(sString + x + y + z);
```

Here, the Java compiler will convert **x**, **y**, and **z** into their **String** representations instead of adding them together first. However, if you say:

```
System.out.println(x + sString);
```

Java will signal an error. So if you're putting together a **String** with addition, make sure the first element is a **String** (or a quoted sequence of characters, which the compiler recognizes as a **String**).

common pitfalls when using operators

One of the pitfalls when using operators is trying to get away without 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 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** and 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 above case, probably an error).

A similar problem in C/C++ is using bitwise AND and OR instead of logical. 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 just type one character instead of two. In Java, this is again prevented by the compiler because it prevents you from cavalierly using 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 (including all modifiers) inside parentheses to the left of any value. Here's an example:

```
void casts() {
  int i = 200;
  long l = (long)i;
  long l2 = (long)200;
}
```

As you can see, it's possible to perform a cast on a numeric value as well as a variable. In both casts shown here, however, the cast is superfluous, since the compiler will automatically promote an **int** value to a **long** when necessary. You can still put a cast in to make a point or to make your code clearer. In other situations, a cast is 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 do 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 and thus 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 (**String** is a special case).

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:

```
//: Literals.java

class Literals {
  char c = 0xffff; // max char hex value
  byte b = 0x7f; // max byte hex value
```

```
    short s = 0x7fff; // max short hex value
    int i1 = 0x2f; // Hexadecimal (lowercase)
    int i2 = 0X2F; // Hexadecimal (uppercase)
    int i3 = 0177; // Octal (leading zero)
    // Hex and Oct also work with long.
    long n1 = 200L; // long suffix
    long n2 = 200l; // long suffix
    long n3 = 200;
    //! long l6(200); // not allowed
    float f1 = 1;
    float f2 = 1F; // float suffix
    float f3 = 1f; // float suffix
    float f4 = 1e-45f; // 10 to the power
    float f5 = 1e+9f; // float suffix
    double d1 = 1d; // double suffix
    double d2 = 1D; // double suffix
    double d3 = 47e47d; // 10 to the power
  } ///:~
```

Hexadecimal (base 16), which works with all the integral data types, is denoted by a leading **0x** or **0X** followed by 0–9 and a–f either in upper or lower case. 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 above 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. 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.

A trailing character after a literal value establishes its type. Upper or lowercase **L** means **long**, upper or lowercase **F** means **float**, and upper or lowercase **D** means **double**.

Exponents use a notation that I've always found rather dismaying: **1.39 e-47f**. In science and engineering, 'e' refers to Euler's constant which is the base of natural logarithms, approximately 2.718. This is used in exponentiation expressions such as $1.39 \times e^{-47}$, which means 1.39×2.719^{-47}. However, when FORTRAN was invented they decided that **e** would naturally mean "ten to the power," which is an odd decision since FORTRAN was designed for science and engineering and one would think its designers would be sensitive about introducing such an ambiguity. At any rate, this custom was followed in C, C++ and now Java. So if you're used to thinking in terms of Euler's constant, you must do a mental translation when you see an expression like **1.39 e-47f** in Java: it means 1.39×2.719^{-47}.

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

```
long n3 = 200;
```

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

```
float f4 = 1e-47f; // 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**.

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 may 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 satisfies a specific need: it tells you the number of bytes allocated for data items. The most compelling need for **sizeof( )** in C and C++ is for portability. Different data types may be different sizes on different machines, so the programmer must find out how big those types are when performing operations that may be sensitive to size. For example, one computer might store integers in 32 bits, whereas another might store integers as 16 bits, and therefore 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 have to think about portability on this level – it is designed into the language.

precedence revisited

Upon hearing me complain about the complexity of remembering operator precedence during one of my seminars, a student suggested a mnemonic that is simultaneously a commentary: "Ulcer Addicts Really Like C A lot."

Mnemonic	Operator type	Operators
Ulcer	Unary	+ - ++ – [[rest…]]
Addicts	Arithmetic (and shift)	* / % + - << >>
Really	Relational	> < >= <= == !=
Like	Logical (and bitwise)	&& || & | ^
C	Conditional (ternary)	A > B ? X : Y
A Lot	Assignment	= (and compound assignment like *=)

Of course, with the shift and bitwise operators distributed around it is not a perfect mnemonic, but for non-bit operations it works.

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 would cause errors are commented out with a **//!**.

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

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 b = (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 b = (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 b = (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 b = (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 b = (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 b = (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 b = (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 very limited: you can assign to it the values **true** and **false**, and you can test it for truth or falsehood. However, 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 results in an **int** result, which must be explicitly cast back to the original type (a narrowing conversion which may 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:

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

public class Overflow {
  public static void main(String args[]) {
    int big = 0x7fffffff; // max int value
    prt("big = " + big);
    int bigger = big * 4;
    prt("bigger = " + bigger);
  }
  static void prt(String s) {
    System.out.println(s);
  }
} ///:~
```

The output of this is:

```
big = 2147483647
bigger = -4
```

and 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 may unknowingly lose information during the cast.

execution control

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

true and false

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

if-else

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

if(*Boolean-expression***)**
statement

or

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

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

Pascal and BASIC programmers should notice that the "then" is implied in Java, which is a terse language. "Then" isn't essential, so it was left out.

To demonstrate the **if-else** and other program control keywords we'll develop, over the next few pages, a program that finds the zeroes in a mathematical function. First we need a test method that will only tell us whether a guess is above, below, or equivalent to a target number:

```
static int test(int testval) {
  int result = 0;
  if(testval > target)
    result = -1;
  else if(testval < target)
    result = +1;
  else
    result = 0; // match
  return result;
}
```

It is conventional to indent the body of a control flow statement so the reader may easily determine where it begins and ends.

return

The **return** keyword has two purposes: it specifies what value a method will return (if it doesn't have a **void** return value) and it also causes that value to be returned immediately. The **test( )** method above can be rewritten to take advantage of this:

```
static int test2(int testval) {
  if(testval > target)
    return -1;
  if(testval < target)
    return +1;
  return 0; // match
}
```

There's no need for **else** because the method will not continue after executing a **return**.

iteration

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

while(*Boolean-expression***)**
statement

The Boolean-expression is evaluated once at the beginning of the loop, and again before each further iteration of the statement. The *statement* can be a compound statement surrounded by curly braces, which is true with any statement.

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

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

public class WhileTest {
  public static void main(String args[]) {
    double r = 0;
    while(r < 0.99d) {
      r = Math.random();
      System.out.println(r);
    }
  }
} ///:~
```

This uses the **static** method **random( )** in the **Math** library, which generates a **double** value between 0 and 1. The conditional expression for the **while** says "keep doing this loop until the number is greater than 0.99. Each time you run this program you'll get a different-sized list of numbers.

do-while

The form for **do-while** is

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

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

for

The **for** allows you to select the path of execution and is thus called a *selection statement*. A **for** loop performs initialization before the first iteration. Then it performs conditional testing and, at the end of each iteration, some form of "stepping." The form of the **for** loop is:

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

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

for loops are usually used for "counting" tasks:

```
//: ListCharacters.java
// Demonstrates "for" loop by listing
// all the ASCII characters.

public class ListCharacters {
  public static void main(String args[]) {
  for( char c = 0; c < 128; c++)
    if (c != 26 )  // ANSI Clear screen
      System.out.println(
        "value: " + (int)c +
        " character: " + c);
  }
} ///:~
```

Notice that the variable **c** is defined at the point where it is used, inside the control expression of the **for** loop, rather than the beginning of the block denoted by the open curly brace. The scope of **c** is the expression controlled by the **for**.

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

You can define multiple variables within a **for** statement, but they must be of the same type:

```
for(int i = 0, j = 1;
    i < 10 && j != 11;
    i++, j++)
 /* body of for loop */;
```

The ability to define variables in the control expression is limited to the **for** loop. You cannot use this approach with any of the other selection or iteration statements.

the comma operator

Earlier in this chapter I stated that the comma *operator* (not the comma *separator*, which is used to separate function arguments) has only one use in Java: in the control expression of a **for** loop. In both the initialization and step portions of the control expression, you can have a number of statements separated by commas, and those statements will be evaluated sequentially. The previous bit of code uses this ability. Here's another example:

```
//: CommaOperator.java

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

Here's the output:

```
i= 1 j= 11
i= 2 j= 4
i= 3 j= 6
i= 4 j= 8
```

You can see that in both the initialization and step portions the statements are evaluated in sequential order. Also, the initialization portion can have any number of definitions *of one type*.

break and continue

Inside the body of any of the iteration statements you can control the flow of the loop using **break** and **continue**. **break** quits the loop without executing the rest of the statements in the loop. **continue** stops the execution of the current iteration and goes back to the beginning of the loop to begin a new iteration.

This program shows examples of **break** and **continue** within **for** and **while** loops:

```
//: BreakAndContinue.java
// Demonstrates break and continue keywords

public class BreakAndContinue {
  public static void main(String args[]) {
    for(int i = 0; i < 100; i++) {
      if(i == 74) break; // Out of for loop
      if(i % 9 != 0) continue; // Next iteration
      System.out.println(i);
    }
    int i = 0;
    // An "infinite loop":
    while(true) {
      i++;
      int j = i * 27;
      if(j == 1269) break; // Out of loop
      if(i % 10 != 0) continue; // Top of loop
      System.out.println(i);
    }
  }
} ///:~
```

In the **for** loop the value of **i** never gets to 100 because the **break** statement breaks out of the loop when **i** is 74. Normally you'd use a **break** like this only if you didn't know when the terminating

condition was going to occur. The **continue** statement causes execution to go back to the top of the iteration loop (thus incrementing **i**) whenever **i** is not evenly divisible by 9. When it is, the value is printed.

The second portion shows an "infinite loop" that would, in theory, continue forever. However, inside the loop there is a **break** statement that will break out of the loop. In addition, you'll see that the **continue** moves back to the top of the loop without completing the remainder (thus printing only happens when the value of **i** is divisible by 10). The output is:

```
0
9
18
27
36
45
54
63
72
10
20
30
40
```

The value 0 is printed because 0 % 9 produces 0.

the infamous "goto"

The **goto** keyword has been present in programming languages from the beginning; indeed, **goto** was the genesis of program control in assembly language: "if condition A, then jump here, otherwise jump there." If you read the assembly code that is ultimately generated by virtually any compiler, you'll see that program control generally resolves into jumps. However, **goto** is an *unconditional* jump, and that's what brought it into disrepute: if a program will always jump from one point to another, isn't there some way to reorganize the code so the flow of control is not so jumpy? **goto** fell into true disfavor with the publication of the famous "Goto considered harmful" paper by Edsger Dijkstra, and since then goto-bashing has been a popular sport, with advocates of the cast-out keyword scurrying for cover.

As is typical in situations like this, the middle ground is the most fruitful. The problem is not the use of **goto** but the overuse of **goto**, and in rare situations **goto** is the best way to structure control flow.

Although **goto** is a reserved word in Java, it is not used in the language; Java has no goto. However, it does have something that looks a bit like a jump tied in with the **break** and **continue** keywords. It's not really a jump, but a way to break out of an iteration statement. The reason it's often thrown in with discussions of **goto** is because it uses the same mechanism: a label.

A label is an identifier followed by a colon, like this:

```
label1:
```

The *only* place a label is useful in Java is right before an iteration statement. And that means *right* before – you can't put any other statement between the label and the iteration (well, you can, but it won't do you any good). And the only reason to put a label before an iteration is if you're going to nest another iteration or a switch inside it. That's because the **break** and **continue** keywords will normally interrupt only the current loop, but when used with a label they'll interrupt the loops up to where the label exists:

```
label:
outer-iteration {
 inner-iteration {
   //...
   break; // 1
   //...
   continue;  // 2
   //...
```

```
  continue label; // 3
  //...
  break label;  // 4
 }
}
```

In case 1, the **break** breaks out of the inner iteration and you end up in the outer iteration. In case 2, the **continue** moves back to the beginning of the inner iteration. But in case 3, the **continue label** breaks out of the inner iteration *and* the outer iteration, all the way back to the **label**. Then it does in fact continue the iteration, but starting at the outer iteration. In case 4, the **break label** also breaks all the way out to **label**, but it does not re-enter the iteration. It actually does break out of both iterations.

Here is an example using **for** loops:

```
//: LabeledFor.java
// Java's "labeled for loop"

public class LabeledFor {
  public static void main(String args[]) {
    int i = 0;
    outer: // Can't have statements here
    for(; true ;) { // infinite loop
      inner: // Can't have statements here
      for(; i < 10; i++) {
        prt("i = " + i);
        if(i == 2) {
          prt("continue");
          continue;
        }
        if(i == 3) {
          prt("break");
          i++; // Otherwise i never
               // gets incremented.
          break;
        }
        if(i == 7) {
          prt("continue outer");
          i++; // Otherwise i never
               // gets incremented.
          continue outer;
        }
        if(i == 8) {
          prt("break outer");
          break outer;
        }
        for(int k = 0; k < 5; k++) {
          if(k == 3) {
            prt("continue inner");
            continue inner;
          }
        }
      }
    }
    // Can't break or continue
    // to labels here
  }
  static void prt(String s) {
    System.out.println(s);
  }
} ///:~
```

This uses the **prt( )** method that has been defined in the other examples.

Notice that **break** breaks out of the **for** loop, and that the increment-expression doesn't occur until the end of the pass through the **for** loop. Since **break** skips the increment expression, the increment is performed by hand in the case of **i == 3**. The **continue outer** statement in the case of **I == 7** also goes to the top of the loop and also skips the increment, so it too is incremented by hand.

Here is the output:

```
i = 0
continue inner
i = 1
continue inner
i = 2
continue
i = 3
break
i = 4
continue inner
i = 5
continue inner
i = 6
continue inner
i = 7
continue outer
i = 8
break outer
```

If not for the **break outer** statement, there would be no way to get out of the outer loop from within an inner loop, since **break** by itself can only break out of the innermost loop (the same is true for **continue**).

Here is a demonstration of labeled **break** and **continue** statements with **while** loops:

```
//: LabeledWhile.java
// Java's "labeled while" loop

public class LabeledWhile {
  public static void main(String args[]) {
    int i = 0;
    outer:
    while(true) {
      prt("Outer while loop");
      while(true) {
        i++;
        prt("i = " + i);
        if(i == 1) {
          prt("continue");
          continue;
        }
        if(i == 3) {
          prt("continue outer");
          continue outer;
        }
        if(i == 5) {
          prt("break");
          break;
        }
        if(i == 7) {
          prt("break outer");
          break outer;
```

```
        }
      }
    }
  }
  static void prt(String s) {
    System.out.println(s);
  }
} ///:~
```

The same rules hold true for **while**:

1. A plain **continue** goes to the top of the innermost loop and continues
2. A labeled **continue** goes to the label and re-enters the loop right after that label
3. A **break** "drops out of the bottom" of the loop
4. A labeled **break** drops out of the bottom of the end of the loop denoted by the label.

The output of this method makes it clear:

```
Outer while loop
i = 1
continue
i = 2
i = 3
continue outer
Outer while loop
i = 4
i = 5
break
Outer while loop
i = 6
i = 7
break outer
```

It's important to remember that the *only* reason to use labels in Java is if you have nested loops, and you want to **break** or **continue** through more than one nested level.

switch

The **switch** is sometimes classified, along with **if**, as a *selection statement*. The **switch** statement selects from among pieces of code based on the value of an integral expression. Its form is:

switch(*integral-selector***) {**
case *integral-value1* **:** *statement***; break;**
case *integral-value2* **:** *statement***; break;**
case *integral-value3* **:** *statement***; break;**
case *integral-value4* **:** *statement***; break;**
case *integral-value5* **:** *statement***; break;**
// **...**
default: *statement***;**
}

Integral-selector is an expression that produces an integral value. The **switch** compares the result of *Integral-selector* to each *integral-value*. If it finds a match, the corresponding *statement* (simple or compound) executes. If no match occurs, the **default** *statement* executes.

You will notice in the above definition that each **case** ends with a **break**, which causes execution to jump to the end of the **switch** body. This is the conventional way to build a **switch** statement, but the **break** is optional. If it is missing, the code for the following case statements execute until a **break** is

encountered. Although you don't usually want this kind of behavior, it can be useful to an experienced programmer.

The **switch** statement is a very clean way to implement multi-way selection (i.e., selecting from among a number of different execution paths), but it requires a selector that evaluates to an integral value such as **int** or **char**. If you want to use, for example, a string or a floating-point number as a selector, it won't work in a **switch** statement. For non-integral types, you must use a series of **if** statements.

Here's an example that randomly creates letters and determines whether they're vowels or consonants:

```
//: VowelsAndConsonants.java
// Demonstrates the switch statement

public class VowelsAndConsonants {
  public static void main(String args[]) {
    for(int i = 0; i < 100; i++) {
      char c = (char)(Math.random() * 26 + 'a');
      System.out.print(c + ": ");
      switch(c) {
      case 'a':
      case 'e':
      case 'i':
      case 'o':
      case 'u':
                System.out.println("vowel");
                break;
      case 'y':
      case 'w':
                System.out.println(
                  "Sometimes a vowel");
                break;
      default:
                System.out.println("consonant");
      }
    }
  }
} ///:~
```

Since **Math.random( )** generates a value between 0 and 1, you only need to multiply it by the upper bound of the range of numbers you want to produce (26 for the letters in the alphabet) and add an offset to establish the lower bound.

Although it appears you're switching on a character here, the **switch** statement is actually using the integral value of the character. The singly-quoted characters in the **case** statements also produce integral values which are used for comparison.

Notice how the **case**s can be "stacked" on top of each other to provide multiple matches for a particular piece of code. You should also be aware that it's essential to put the **break** statement at the end of a particular case, otherwise control will simply drop through and continue processing on the next case.

calculation details

The statement:

```
char c = (char)(Math.random() * 26 + 'a');
```

deserves a closer look. **Math.random( )** produces a **double**, so the value 26 is converted to a **double** to perform the multiplication, which also produces a **double**. This means that **'a'** must be converted to a **double** to perform the addition. The **double** result is turned back into a **char** with a cast.

First, what does the cast to **char** do? That is, if you have the value 29.7 and you cast it to a **char**, is the resulting value 30 or 29? The answer to this can be seen with an example:

```
//: CastingNumbers.java
// What happens when you cast a float or double
// to an integral value?

public class CastingNumbers {
  public static void main(String args[]) {
    double
      above = 0.7,
      below = 0.4;
    System.out.println("above: " + above);
    System.out.println("below: " + below);
    System.out.println(
      "(int)above: " + (int)above);
    System.out.println(
      "(int)below: " + (int)below);
    System.out.println(
      "(char)('a' + above): " +
      (char)('a' + above));
    System.out.println(
      "(char)('a' + below): " +
      (char)('a' + below));
  }
} ///:~
```

The output is:

```
above: 0.7
below: 0.4
(int)above: 0
(int)below: 0
(char)('a' + above): a
(char)('a' + below): a
```

So the answer is that casting from a **float** or **double** to an integral value always truncates.

The second question has to do with **Math.random( )**. Does it produce a value from zero to one, inclusive or exclusive of the value '1'? In math lingo, is it (0,1) or [0,1] or (0,1] or [0,1)? (The square bracket means "includes" whereas the parenthesis means "doesn't include"). Again, a test program provides the answer:

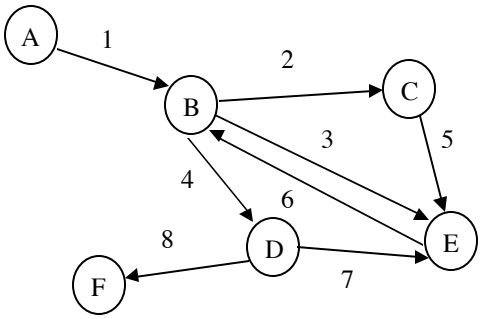
```
//: RandomBounds.java
// Does Math.random() produce 0.0 and 1.0?

public class RandomBounds {
  static void usage() {
    System.err.println("Usage: \n\t" +
      "RandomBounds lower\n\t" +
      "RandomBounds upper");
    System.exit(1);
  }
  public static void main(String args[]) {
    if(args.length != 1) usage();
    if(args[0].equals("lower")) {
      while(Math.random() != 0.0)
        ; // Keep trying
      System.out.println("Produced 0.0!");
    }
    else if(args[0].equals("upper")) {
```

```
      while(Math.random() != 1.0)
        ; // Keep trying
      System.out.println("Produced 1.0!");
    }
    else
      usage();
  }
} ///:~
```

In both cases, you are forced to break out of the program manually, so **Math.random( )** never produces either 0.0 or 1.0. Or, in math lingo, it is (0,1).

a state machine

As long as you're working with integral values, the **switch** can eliminate a lot of messy code that would normally require many **if** statements. Consider a more complicated example (which you can skip without damage if it doesn't interest you). A *state machine* moves through a series of states based on the input it receives. A number of types of computer programming problems can be modeled this way. The diagram for the states in this example looks like this:



Each state is shown as a circle containing the name of the state, and the input that causes the state to change from one circle to another is indicated next to the arrow showing the direction of change. Thus, if you're in state B and the input is 3, the machine changes to state E.

Using the **switch**, the solution can be implemented as follows:

```
//: StateMachine.java
// Demonstrates the use of switch
// statement by building a state machine.

public class StateMachine {
  static int input[] = {
    1, 2, 5, 6, 3, 6, 4, 7, 6, 4, 8, 1
  };
  static int i = 0;
  static char state = 'A';
  static void prtState() {
    System.out.println(
      "" + i +
      ": state = " + state +
      ", input = " + input[i]);
  }
  public static void main(String args[]) {
    outer:
    while(true) {
      switch(state) {
        case 'A':
          prtState();
          switch(input[i]) {
```

```
            case 1:
              state = 'B';
            default:
          }
          i++;
          continue outer;
        case 'B':
          prtState();
          switch(input[i]) {
            case 2:
              state = 'C';
              break;
            case 3:
              state = 'E';
              break;
            case 4:
              state = 'D';
            default:
          }
          i++;
          continue outer;
        case 'C':
          prtState();
          switch(input[i]) {
            case 5:
              state = 'E';
            default:
          }
          i++;
          continue outer;
        case 'D':
          prtState();
          switch(input[i]) {
            case 7:
              state = 'E';
              break;
            case 8:
              state = 'F';
            default:
          }
          i++;
          continue outer;
        case 'E':
          prtState();
          switch(input[i]) {
            case 6:
              state = 'B';
            default:
          }
          i++;
          continue outer;
        case 'F':
        default:
          prtState();
          break outer;
      }
    }
  }
} ///:~
```

This example also shows the use of labeled **continue** and **break** statements within **switch** statements (again, labeled **continue** and **break** statements can only be used within nested loops). The program enters an infinite **while** loop, meaning that it is supposed to continue moving through states in the machine forever. The outer **switch** is based on the current state of the machine, and each **case** contains its own **switch** statement based on the input that drives it to the next state. At the end it reaches state F, and the **break outer** gets it all the way out of the loop.

The input that drives the machine from one state to the next is provided by something you haven't seen yet: an *array*:

```
static int input[] = {
  1, 2, 5, 6, 3, 6, 4, 7, 6, 4, 8, 1
};
```

What makes this an array rather than just a single variable is the square brackets (**[ ]**) after the identifier **input**. In addition, the array is initialized with a set of values, using the form you see above. This form can be used for all the primitive types. You can also create arrays of objects but that requires a different form. Arrays will be covered in detail in a later chapter.

To access an element of the array, you simply select the number of that element inside the square brackets:

```
input[i]
```

This produces the value of the **i**th element. The indexing of elements starts at zero, which is something you must keep track of, even though Java provides error checking to keep you from running off the end of the array. You can use the same form to assign to an array element.

This is intended only as an example of the use of **switch** statements, not as the optimal way to implement a state machine. In fact, this is a rather primitive way to do it and a more elegant way is through the use of objects.

summary

This chapter concludes the study of fundamental features that appear in most programming languages: calculation, operator precedence, type casting, and selection and iteration. Now you're ready to begin taking steps that move you closer to the world of object-oriented programming. In the next chapter, the important issues of initialization and cleanup of objects will be covered, followed in the subsequent chapter by the essential concept of implementation hiding.

exercises

1. Write a program that prints values from one to 100.
2. Modify Exercise 1 so the program exits by using the **break** keyword at value 47. Try using **return** instead.
3. Draw a state machine diagram and implement the state machine, using **StateMachine.java** as a model.

4: initialization & cleanup

[[[Chapter 4 directory:c04]]]

As the computer revolution progresses, "unsafe" programming has become one of the major culprits that make programming expensive.

Two of these safety issues are *initialization* and *cleanup*. Many C bugs occur when the programmer forgets to initialize a variable. This is especially true with libraries, when users don't know how to initialize a library component, or even that they must. Cleanup is a special problem because it's easy to forget about an element when you're done with it, since it no longer concerns you. Thus, the resources used by that element are still retained, and you can easily end up running out of resources (most notably memory).

C++ introduced the concept of a *constructor*, a special method automatically called when an object is created. Java also adopted the constructor, and in addition has a garbage collector that automatically releases memory resources when they're no longer being used. This chapter examines the issues of initialization and cleanup and their support in Java.

guaranteed initialization with the constructor

You can imagine creating a method called **initialize( )** for every class you write. The name is a hint that it should be called before using the object. Unfortunately, this means the user must remember to call the method. In Java, the class designer can guarantee initialization of every object by providing a special method called a *constructor*. If a class has a constructor, the compiler automatically calls that constructor when an object is created, before users can even get their hands on it. So initialization is guaranteed.

The next challenge is what to name this method. There are two issues. The first is that any name you use could clash with a name you might like to use as a member in the class. The second is that because the compiler is responsible for calling the constructor, it must always know which method to call. The C++ solution seems the easiest and most logical, so it's also used in Java: The name of the constructor is the same as the name of the class. It makes sense that such a method will be called automatically on initialization.

Here's a simple class with a constructor:

```
//: SimpleConstructor.java
// Demonstration of a simple constructor

class Rock {
  Rock() { // This is the constructor
    System.out.println("Creating Rock");
  }
}

public class SimpleConstructor {
  public static void main(String args[]) {
    for(int i = 0; i < 10; i++)
      new Rock();
  }
} ///:~
```

Now, when an object is created:

```
new Rock();
```

storage is allocated and the constructor is called. It is guaranteed that the object will be properly initialized before you can get your hands on it.

Note that the coding style of making the first letter of all methods lower case must necessarily be lifted for constructors, since the name of the constructor must match the name of the class *exactly*.

Like any method, the constructor can have arguments to allow you to specify *how* an object is created. The above example can easily be changed so the constructor takes an argument:

```
class Rock {
  Rock(int i) {
    System.out.println(
      "Creating Rock number " + i);
  }
}

public class SimpleConstructor {
  public static void main(String args[]) {
    for(int i = 0; i < 10; i++)
      new Rock(i);
```

```
    }
}
```

Constructor arguments provide you with a way to guarantee that all parts of your object are initialized to appropriate values. For example, if the class **Tree** has a constructor that takes a single integer argument denoting the height of the tree, you would create a **Tree** object like this:

```
Tree t = new Tree(12);  // 12-foot tree
```

If **Tree(int)** is your only constructor, then the compiler won't let you create an object any other way.

Constructors eliminate a large class of problems and make the code easier to read. In the preceding code fragment, for example, you don't see an explicit call to some **initialize( )** method that is conceptually separate from definition. In Java, definition and initialization are unified concepts — you can't have one without the other.

The constructor is a very unusual type of method: it has no return value. This is distinctly different from a **void** return value, where the method returns nothing but you still have the option to make it return something else. Constructors return nothing and you don't have an option. If there were a return value, and if you could select your own, the compiler would somehow have to know what to do with that return value.

method overloading

One of the important features in any programming language is the use of names. When you create an object (a variable), you give a name to a region of storage. A method is a name for an action. By using names to describe your system, you create a program that is easier for people to understand and change. It's a lot like writing prose — the goal is to communicate with your readers.

You refer to all objects and methods by using names. Well-chosen names make it easier for you and others to understand your code.

A problem arises when mapping the concept of nuance in human language onto a programming language. Often, the same word expresses a number of different meanings – it's *overloaded*. This is very useful, especially when it comes to trivial differences. You say "wash the shirt," "wash the car," and "wash the dog." It would be silly to be forced to say, "shirtWash the shirt," "carWash the car," and "dogWash the dog" just so the listener doesn't have to make any distinction about the action performed. Most human languages are redundant, so even if you miss a few words, you can still determine the meaning. We don't need unique identifiers — we can deduce meaning from context.

Most programming languages (C in particular) require that you have a unique identifier for each function. Thus you could not have one function called **print( )** for printing integers and another called **print( )** for printing floats — each function requires a unique name.

In Java, another factor forces the overloading of method names: the constructor. Because the constructor's name is predetermined by the name of the class, there can be only one constructor name. But what if you want to create an object in more than one way? For example, suppose you build a class that can initialize itself in a standard way and also by reading information from a file. You need two constructors, one that takes no arguments (the *default* constructor) and one that takes a **String** as an argument, which is the name of the file from which to initialize the object. Both are constructors, so they must have the same name — the name of the class. Thus *method overloading* is essential to allow the same method name to be used with different argument types. And although method overloading is a must for constructors, it's a general convenience and can be used with any method.

Here's an example that shows both overloaded constructors and overloaded ordinary methods:

```
//: Overloading.java
// Demonstration of both constructor
// and ordinary method overloading.
```

```
import java.util.*;

class Tree {
  int height;
  Tree() {
    prt("Planting a seedling");
    height = 0;
  }
  Tree(int i) {
    prt("Creating new Tree that is "
        + i + " feet tall");
    height = i;
  }
  void info() {
    prt("Tree is " + height
        + " feet tall");
  }
  void info(String s) {
    prt(s + ": Tree is "
        + height + " feet tall");
  }
  static void prt(String s) {
    System.out.println(s);
  }
}

public class Overloading {
  static Random rand = new Random();
  static int pRand(int mod) {
    return Math.abs(rand.nextInt()) % mod;
  }
  public static void main(String args[]) {
    int i = 0;
    while(i != 9) {
      Tree t = new Tree(i = pRand(10));
      t.info();
      t.info("overloaded method");
    }
    // Overloaded constructor:
    new Tree();
  }
} ///:~
```

A **Tree** object may be created either as a seedling, with no argument, or as grown in a nursery, with an existing height. To support this, there are two constructors, one that takes no arguments (we call constructors that take no arguments *default constructors*) and one that takes the existing height.

You may also want to call the **info( )** method in more than one way: with a **String** argument if you have an extra message you want printed, and without if you have nothing more to say. It would seem strange to have to give two separate names to what is obviously the same concept. Fortunately, method overloading allows you to use the same name for both.

distinguishing overloaded methods

If the methods have the same name how can Java know which method you mean? There's a very simple rule: Each overloaded method must take a unique list of argument types.

If you think about this for a second, it makes sense: how else could you as a programmer tell the difference between two methods that have the same name, other than by the types of their arguments?

Even differences in the ordering of arguments is sufficient to distinguish two methods:

```
//: OverloadingOrder.java
// Overloading based on the order of
// the arguments.

public class OverloadingOrder {
  static void print(String s, int i) {
    System.out.println(
      "String: " + s +
      ", int: " + i);
  }
  static void print(int i, String s) {
    System.out.println(
      "int: " + i +
      ", String: " + s);
  }
  public static void main(String args[]) {
    print("String first", 11);
    print(99, "Int first");
  }
} ///:~
```

The two **print( )** methods have identical arguments, but the order is different, and that's what makes them distinct.

overloading on return values

It is common to wonder "Why just class names and method argument lists? Why not distinguish between methods based on their return values?" For example, these two methods, which have the same name and arguments, are easily distinguished from each other:

```
void f();
int f();
```

This works fine when the compiler can unequivocally determine the meaning from the context, as in **int x = f( )**. However, you can call a method and ignore the return value; this is often referred to as *calling a method for its side effect* since you don't care about the return value but instead want the other effects of the method call. So if you call the method this way:

```
f();
```

how can Java determine which f() should be called? And how could someone reading the code see it? Because of this sort of problem, you cannot use return value types to distinguish overloaded methods.

default constructors

A *default constructor* is one without arguments, used to create a "vanilla object." If you create a class that has no constructors, the compiler will automatically create a default constructor for you. For example:

```
//: DefaultConstructor.java

class Bird {
  int i;
}

public class DefaultConstructor {
  public static void main(String args[]) {
    Bird nc = new Bird(); // default!
  }
```

```
} ///:~
```

The line

```
new Bird();
```

Creates a new object and calls the default constructor, even though one was not explicitly defined. Without it we would have no method to call to build our object. Now, however, if you define any constructors (with or without arguments), the compiler will *not* synthesize one for you:

```
class Bush {
  Bush(int i) {}
  Bush(double d) {}
}
```

Now if you say:

```
new Bush();
```

The compiler will complain that it cannot find a constructor that matches. It's as if when you don't put in any constructors, it says: "you are bound to need *some* constructor, so let me make one for you." But if you write a constructor, it says "you've written a constructor so you know what you're doing; if you didn't put in a default it's because you meant to leave it out."

In these cases, the default constructor doesn't do anything in particular. However, when you learn about inheritance in Chapter 6, the constructor will be revisited and you'll see that some other functionality can occur with the default constructor.

the this keyword

If you have two objects of the same type called **a** and **b**, you may wonder how it is you can call a method **f( )** for both those objects:

```
class Banana { void f(int i) { /* ... */ } }
Banana a = new Banana(), b = new Banana();
a.f(1);
b.f(2);
```

If there's only one method called **f( )**, how can that method know whether it's being called for the object **a** or **b**?

To allow you to write the code in a convenient object-oriented syntax where you're "sending a message to an object," the compiler does some work for you under the covers. There's a secret first argument passed to the method **f( )**, and that argument is the handle to the object that's being manipulated. So the two method calls above become something like:

```
Banana.f(a,1);
Banana.f(b,2);
```

This is internal and you can't write these expressions and get the compiler to accept them, but it gives you an idea of what's happening.

Now suppose you're inside a method and you'd like to get the handle to the current object. Since that handle is passed *secretly* by the compiler, there's no identifier for it. However, for this purpose there's a keyword: **this**. The **this** keyword – which can only be used inside a method – produces the handle to the object the method has been called for. You can treat this handle just like any other object handle. Keep in mind that if you're calling a method of your class from within another method of your class, you don't need to use **this** – you simply call the method. The current **this** handle is automatically used for the other method. Thus you can say:

```
class Apricot {
  void pick() { /* ... */ }
  void pit() { pick(); /* ... */ }
```

```
}
```

Inside **pit( )**, you *could* say **this.pick( )** but there's no need to – the compiler does it for you automatically. Thus the **this** keyword is only used for special cases when you need to explicitly use the handle to the current object. For example, it's often used in **return** statements when you want to return the handle to the current object:

```
//: Leaf.java
// Simple use of the "this" keyword

public class Leaf {
  private int i = 0;
  Leaf increment() {
    i++;
    return this;
  }
  void print() {
    System.out.println("i = " + i);
  }
  public static void main(String args[]) {
    Leaf x = new Leaf();
    x.increment().increment().increment().print();
  }
} ///:~
```

Because **increment( )** returns the handle to the current object via the **this** keyword, multiple operations can easily be performed on the same object.

calling constructors from constructors

When you write several constructors for a class, there are times when you'd like to call one constructor from another, to avoid duplicating code. You can do this using the **this** keyword.

Normally, when you say **this**, it is in the sense of "this object" or "the current object," and by itself it produces the handle to the current object. In a constructor, the **this** keyword takes on a different meaning when you give it an argument list: it makes an explicit call to the constructor that matches that argument list. Thus you have a straightforward way to call other constructors:

```
//: Flower.java
// Calling constructors with "this"

public class Flower {
  private int i = 0;
  private String s = new String("null");
  Flower(int x) {
    i = x;
    System.out.println(
      "Constructor w/ int arg only, i = " + i);
  }
  Flower(String ss) {
    System.out.println(
      "Constructor w/ String arg only, s=" + ss);
    s = ss;
  }
  Flower(String s, int x) {
    this(x);
//!    this(s); // Can't call two!
    this.s = s; // Another use of "this"
    System.out.println("String & int args");
  }
  Flower() {
    this("hi", 47);
```

```
    System.out.println(
      "default constructor (no args)");
  }
  void print() {
//!    this(11); // Not inside non-constructor!
    System.out.println("i = " + i + " s = " + s);
  }
  public static void main(String args[]) {
    Flower x = new Flower();
    x.print();
  }
} ///:~
```

The constructor **Flower(String s, int x)** shows that, while you can call one constructor using **this**, you cannot call two. In addition, the constructor call must be the first thing you do, otherwise you'll get a compiler error message.

This example also shows another way you'll see **this** used. Since the name of the argument **s** and the name of the member data **s** are the same, there's an ambiguity. You can resolve it by saying **this.s** to refer to the member data. You'll often see this form used in Java code, and it's used in numerous places in this book.

In **print( )** you can see that the compiler won't let you call a constructor from inside any method other than a constructor.

the meaning of static

With the **this** keyword in mind, you can more fully understand what it means to make a method **static**. It means there is no **this** for that particular method. Thus you cannot call non-**static** methods from inside **static** methods (although the reverse is possible), and it's possible to call a **static** method for the class itself, without any object. In fact, that's primarily what a **static** method is for. It's as if you're creating the equivalent of a global function (from C). Except that global functions are not permitted in Java, and putting the **static** method inside a class allows it access to other **static** methods and to **static** fields.

Some people argue that **static** methods are not object-oriented, since they do have the semantics of a global function – with a **static** method you don't send a message to an object, since there's no **this**. This is probably a fair argument, and if you find yourself using a *lot* of static methods you should probably consider rethinking your strategy. However, **static**s are pragmatic and there are times you genuinely need them so whether or not they are "proper OOP" should be left to the theoreticians. Indeed, even SmallTalk has the equivalent in its "class methods."

cleanup: finalization and garbage collection

Programmers know about the importance of initialization, but often forget the importance of cleanup. After all, who needs to clean up an **int**? However, with libraries, just "letting go" of an object once you're done with it is not always safe. Of course, Java has the garbage collector to reclaim the memory of objects that are no longer used. But consider a very special (and unusual case). Suppose your object allocates memory without using **new**. The garbage collector only knows how to release memory allocated *with* **new**, so it won't know how to release the object's "special" memory. To handle this special case, Java provides a method called **finalize( )** that you can define for your class. Here's how it's *supposed* to work: when the garbage collector is ready to release the storage used for your object, it will first call **finalize( )**, and only on the next garbage-collection pass will it reclaim the object's memory. So if you choose to redefine **finalize( )**, it gives you the ability to perform some important cleanup *at the time of garbage collection*.

This is a potential programming pitfall, because some programmers, especially C++ programmers, may initially mistake **finalize( )** for the *destructor* in C++, which is a function that is always called when an object is destroyed. But it is very important to distinguish between C++ and Java here, because in C++, *objects always get destroyed* (in a bug-free program), whereas in Java objects do not always get garbage-collected. Or, put another way:

Garbage collection is not destruction.

If you remember this, you will stay out of trouble. What it means is that if there is some activity that must be performed before you no longer need an object, you must perform that activity yourself. Java has no destructor or similar concept, so you must create an ordinary method to perform this cleanup. For example, suppose in the process of creating your object it draws itself on the screen. If you don't explicitly erase its image from the screen, it may never get cleaned up. If you put some kind of erasing functionality inside **finalize( )**, then if an object is garbage-collected, the image will first be removed from the screen, but if it isn't the image will remain. So a second point to remember is:

Your objects may not get garbage collected.

You may find that the storage for an object never gets released because your program never nears the point of running out of storage. If your program completes and the garbage collector never gets around to releasing the storage for any of your objects, that storage will be returned to the operating system *en masse* as the program exits. This is a good thing, because garbage collection has some overhead, and if you never do it you never incur that expense.

what is finalize( ) for?

You might believe at this point that you should not use **finalize( )** as a general-purpose cleanup method. What good is it?

A third point to remember is:

Garbage collection is only about memory.

That is, the sole reason for the existence of the garbage collector is to recover memory that is no longer being used by your program. So any activity that is associated with garbage collection, most notably your **finalize( )** method, must also be only about memory.

Does this mean that if your object contains other objects **finalize( )** should explicitly release those objects? Well, no – the garbage collector takes care of the release of all object memory, regardless of how the object is created. It turns out that the need for **finalize( )** is limited to special cases, where your object may allocate some storage in some way other than creating an object. But, you may observe, everything in Java is an object so how can this be?

It would seem that **finalize( )** is in place because of the possibility that you'll do something C-like by allocating memory using a mechanism other than the normal one in Java. The way this can happen is primarily through *native methods*, which are a way to call non-Java code from Java (native methods are discussed in Appendix A). C and C++ are the only languages currently supported, but since they in turn can call subprograms in other languages, you can effectively call anything. Inside the non-Java code, C's **malloc( )** family of functions may be called to allocate storage, and unless you call **free( )** that storage will not be released, causing a memory leak. Of course, **free( )** is itself a C/C++ function, so you'd have to call that inside a native method inside your **finalize( )**.

After reading this, you probably get the idea that you won't use **finalize( )** very much. You're right – it is not the appropriate place for normal cleanup to occur. So where should normal cleanup be performed?

you must perform cleanup

The answer is this: To clean up an object, the user of that object must call a cleanup method at the point the cleanup is desired. This sounds pretty straightforward, but it collides a bit with the C++ concept of a *destructor*, a function that is automatically called when a C++ object is destroyed. In C++, all objects are destroyed. Or rather, all objects *should be* destroyed. If the C++ object is created

as a local (not possible in Java) then the destruction happens at the closing curly brace of the scope where the object was created. If the object was created using **new** (like in Java) the destructor is called when the programmer calls the C++ operator **delete** (which doesn't exist in Java). If the programmer forgets, the destructor is never called and you have a memory leak, plus the other parts of the object never get cleaned up.

In contrast, Java doesn't allow you to create local objects – you must always use **new**. But in Java, there's no "delete" to call for releasing the object since the garbage collector releases the storage for you. So from a simplistic standpoint you could say that because of garbage collection, Java has no destructor. You'll see as the book progresses, however, that the presence of a garbage collector does not remove the need or utility of destructors. If you want some kind of cleanup performed other than storage release you must *still* call a method in Java, which is the equivalent of a C++ destructor without the convenience.

One of the things **finalize( )** can be useful for is observing the process of garbage collection. The following example shows you what's going on and summarizes the previous descriptions of garbage collection:

```
//: Garbage.java
// Demonstration of the garbage
// collector and finalization

class Chair {
  static boolean gcrun = false;
  static boolean f = false;
  static int created = 0;
  static int finalized = 0;
  int i;
  Chair() {
    i = created++;
    if(created == 47)
      System.out.println("Created 47");
  }
  protected void finalize() {
    if(!gcrun) {
      gcrun = true;
      System.out.println(
        "Beginning to finalize after " +
        created + " Chairs have been created");
    }
    if(i == 47) {
      System.out.println(
        "Finalizing Chair #47, " +
        "Setting flag to stop Chair creation");
      f = true;
    }
    finalized++;
    if(finalized >= created)
      System.out.println(
        "All " + finalized + " finalized");
  }
}

public class Garbage {
  public static void main(String args[]) {
    if(args.length == 0) {
      System.err.println("Usage: \n" +
        "java Garbage before\n  or:\n" +
        "java Garbage after");
      return;
    }
```

```
    while(!Chair.f) {
      new Chair();
      new String("To take up space");
    }
    System.out.println(
      "After all Chairs have been created:\n" +
      "total created = " + Chair.created +
      ", total finalized = " + Chair.finalized);
    if(args[0].equals("before")) {
      System.out.println("gc():");
      System.gc();
      System.out.println("runFinalization():");
      System.runFinalization();
    }
    System.out.println("bye!");
    if(args[0].equals("after"))
      System.runFinalizersOnExit(true);
  }
} ///:~
```

The above program creates many **Chair** objects, and at some point after the garbage collector begins running, the program stops creating **Chair**s. Since the garbage collector can be run at any time, you don't know exactly when it will start up, so there's a flag called **gcrun** to indicate whether the garbage collector has started running yet. A second flag **f** is a way for **Chair** to tell the **main( )** loop that it should stop making objects. Both of these flags are set within **finalize( )**, which is called during garbage collection.

Two other **static** variables, **created** and **finalized**, keep track of the number of **obj**s created vs. the number that get finalized by the garbage collector. Finally, each **Chair** has its own (i.e.: non-**static**) **int i** so it can keep track of what number it is. When the magic number 47 is finalized, the flag is set to **true** to bring the process of **Chair** creation to a stop.

All this happens in **main( )**, in the loop

```
    while(!Chair.f) {
      new Chair();
      new String("To take up space");
    }
```

Normally you'd wonder how this loop could ever finish, since there's nothing inside that changes the value of **Chair.f**. However, the **finalize( )** process will, eventually, when it finalizes number 47.

The creation of a **String** object during each iteration is simply extra garbage being created to encourage the garbage collector to kick in, which it will do when it starts to get nervous about the amount of memory available.

When you run the program, you provide a command-line argument of "before" or "after." The "before" argument will call the **System.gc( )** method (to force execution of the garbage collector) along with the **System.runFinalization( )** method to run the finalizers. These methods were available in Java 1.0, but the **runFinalizersOnExit( )** method that is invoked by using the "after" argument is only available in Java 1.1.

Unfortunately, the implementations of the garbage collector in Java 1.0 would never call **finalize( )** correctly. As a result, **finalize( )** methods that were essential (such as those to close a file) often didn't get called. The documentation claimed that all finalizers would be called at the exit of a program, even if the garbage collector hadn't been run on those objects by the time the program terminated. This wasn't true, so as a result you couldn't reliably expect **finalize( )** to be called for all objects. Effectively, **finalize( )** was useless in Java 1.0.

The above program shows that, in Java 1.1, the promise that finalizers will always be run holds true, but only if you explicitly force it to happen yourself. If you use an argument that isn't "before" or "after" (such as "none") then neither finalization process will occur, and you'll get an output like this:

```
Created 47
Beginning to finalize after 8694 Chairs have been created
Finalizing Chair #47, Setting flag to stop Chair creation
After all Chairs have been created:
total created = 9834, total finalized = 108
bye!
```

Thus, not all finalizers get called by the time the program completes. To force finalization to happen, you can call **System.gc( )** followed by **System.runFinalization( )**. This will destroy all the objects that are no longer in use up to that point. The odd thing about this is that you call **gc( )** *before* you call **runFinalization( )**, which seems to contradict the JavaSoft documentation which claims that finalizers are run first, and then the storage is released. However, if you call **runFinalization( )** first, and then **gc( )**, the finalizers will not be executed.

One reason that Java 1.1 may default to skipping finalization for all objects is because it seems to be expensive. When you use either of the approaches that force garbage collection you may notice longer delays than without the extra finalization.

member initialization

Java goes out of its way to guarantee that any variable is properly initialized before it is used. In the case of variables that are defined locally to a method, this guarantee comes in the form of a compile-time error. So if you say:

```
void f() {
  int i;
  i++;
}
```

You'll get an error message that says that **i** may not have been initialized. Of course, the compiler could have given **i** a default value, but it's more likely that this is a programmer error and a default value would have covered that up. Forcing the programmer to provide an initialization value is more likely to catch a bug.

If a primitive is a data member of a class, however, things are a bit different. Since any method may initialize or use that data, it may not be practical to force the user to initialize it to its appropriate value before the data is used. However, it's unsafe to leave it with a garbage value, so each primitive data member of a class is guaranteed to get an initial value. Those values can be seen here:

```
//: InitialValues.java
// Shows default initial values

class Measurement {
  boolean t;
  char c;
  byte b;
  short s;
  int i;
  long l;
  float f;
  double d;
  void print() {
    System.out.println(
      "Data type      Inital value\n" +
      "boolean        " + t + "\n" +
      "char           " + c + "\n" +
      "byte           " + b + "\n" +
      "short          " + s + "\n" +
      "int            " + i + "\n" +
```

```
      "long           " + l + "\n" +
      "float          " + f + "\n" +
      "double         " + d);
  }
}

public class InitialValues {
  public static void main(String args[]) {
    Measurement d = new Measurement();
    d.print();
    /* In this case you could also say:
    new Measurement().print();
    */
  }
} ///:~
```

The output of this program is:

```
Data type      Inital value
boolean        false
char
byte           0
short          0
int            0
long           0
float          0.0
double         0.0
```

The **char** value is a null, which doesn't print.

You'll see later that when you define an object handle inside a class without initializing it to a new object, that handle is given a value of null.

You can see that even though the values are not specified, they automatically get initialized. So at least there's no threat of working with uninitialized variables.

specifying initialization

But what happens if you want to give an initial value? One very direct way to do this is simply assign the value at the point you define the variable in the class (note you cannot do this in C++, although C++ novices always try). Here the first lines in the default constructor of class **Measurement** are changed to provide initial values:

```
class Measurement {
  boolean b = true;
  char c = 'x';
  byte B = 47;
  short s = 0xff;
  int i = 999;
  long l = 1;
  float f = 3.14f;
  double d = 3.14159;
  //. . .
```

You can also initialize class objects in this same way. If **Depth** is a class, you can insert a variable and initialize it like so:

```
class Measurement {
  Depth o = new Depth();
  boolean b = true;
  // . . .
```

If you haven't given **o** an initial value and you go ahead and try to use it anyway, you'll get an exception at run-time.

You can even call a method to provide an initialization value:

```
class CInit {
  int i = f();
  //...
}
```

This method may have arguments, of course, but those arguments cannot be other class members that haven't been initialized yet. Thus, you can do this:

```
class CInit {
  int i = f();
  int j = g(i);
  //...
}
```

But you cannot do this:

```
class CInit {
  int j = g(i);
  int i = f();
  //...
}
```

This is one place where the compiler, appropriately, *does* complain about forward referencing, since this has to do with the order of initialization and not the way the program is compiled.

This approach to initialization is simple and straightforward. It has the limitation that *every* object of type **Measurement** will get these same initialization values. Sometimes this is exactly what you need, but at other times you need more flexibility.

constructor initialization

The constructor can be used to perform initialization, and this gives you greater flexibility in your programming, since you may call methods and perform actions at run time to determine the initial values. There's one thing to keep in mind, however: you aren't precluding the automatic initialization, which happens before the constructor is entered. So, for example, if you say:

```
class Counter {
  int i;
  Counter() { i = 7; }
  // . . .
```

then **i** will first be initialized to zero, then to 7. This is true with all the primitive types and with object handles, including those that are given explicit initialization at the point of definition. For this reason, the compiler doesn't try to force you to initialize elements in the constructor at any particular place, or before they are used – initialization is already guaranteed.[1]

order of initialization

Within a class, the order of initialization is strictly in the order that the variables are defined within the class. Even if the variable definitions are scattered throughout in between method definitions, the variables are initialized before any methods can be called, even the constructor. For example:

```
//: OrderOfInitialization.java
```

[1] In contrast, C++ has the *constructor initializer list* that causes initialization to occur before entering the constructor body, and is enforced for objects. See *Thinking in C++*.

```
// Demonstrates initialization order.

class Tag {
  Tag(int marker) {
    System.out.println(
      "Tag(" + marker + ")");
  }
}

class Card {
  Tag t1 = new Tag(1);
  Card() {
    System.out.println("Card()");
    t3 = new Tag(33);
  }
  Tag t2 = new Tag(2);
  void f() {
    System.out.println("f()");
  }
  Tag t3 = new Tag(3);
}

public class OrderOfInitialization {
  public static void main(String args[]) {
    Card t = new Card();
    t.f();
  }
} ///:~
```

In **Card**, the definitions of the **Tag** objects are intentionally scattered about to prove that they'll all get initialized before the constructor is entered or anything else can happen. In addition, **t3** is re-initialized inside the constructor. The output is:

```
Tag(1)
Tag(2)
Tag(3)
Card()
Tag(33)
f()
```

Thus, the **t3** handle gets initialized twice, once before the constructor call and once during (the first object is dropped, so it may be garbage-collected later). This may not seem very efficient at first, but it guarantees proper initialization – what would happen if an overloaded constructor were defined that did *not* initialize **t3** and there wasn't a "default" initialization for **t3** in its definition?

static data initialization

What happens when the data is **static**? Exactly the same thing: if it's a primitive and you don't initialize it, it gets the standard primitive initial values. If it's a handle to an object, it's null unless you create a new object and attach your handle to it.

If you want to place initialization at the point of definition, it looks the same as for non-statics. But since there's only one piece of storage for a **static**, regardless of how many objects are created, a question comes up: when does that storage get initialized? An example makes this question clear:

```
//: StaticInitialization.java
// Specifying initial values in a
// class definition.

class Bowl {
  Bowl(int marker) {
    System.out.println("Bowl(" + marker + ")");
```

```
  }
  void f(int marker) {
    System.out.println("f(" + marker + ")");
  }
}

class Table {
  static Bowl b1 = new Bowl(1);
  Table() {
    System.out.println("Table()");
    b2.f(1);
  }
  void f2(int marker) {
    System.out.println("f2(" + marker + ")");
  }
  static Bowl b2 = new Bowl(2);
}

class Cupboard {
  Bowl b3 = new Bowl(3);
  static Bowl b4 = new Bowl(4);
  Cupboard() {
    System.out.println("Cupboard()");
    b4.f(2);
  }
  void f3(int marker) {
    System.out.println("f3(" + marker + ")");
  }
  static Bowl b5 = new Bowl(5);
}

public class StaticInitialization {
  public static void main(String args[]) {
    System.out.println(
      "Creating new Cupboard() in main");
    new Cupboard();
    System.out.println(
      "Creating new Cupboard() in main");
    new Cupboard();
    t2.f2(1);
    t3.f3(1);
  }
  static Table t2 = new Table();
  static Cupboard t3 = new Cupboard();
} ///:~
```

Bowl allows you to view the creation of a class, and **Table** and **Cupboard** create **static** members of **Bowl** scattered through their class definitions. Notice that **Cupboard** creates a non-**static Bowl b3** prior to the **static** definitions. The output shows what happens:

```
Bowl(1)
Bowl(2)
Table()
f(1)
Bowl(4)
Bowl(5)
Bowl(3)
Cupboard()
f(2)
Creating new Cupboard() in main
```

```
Bowl(3)
Cupboard()
f(2)
Creating new Cupboard() in main
Bowl(3)
Cupboard()
f(2)
f2(1)
f3(1)
```

The **static** initialization only occurs if it's necessary – if you don't create a **Table** object, for example, the **static Bowl b1** and **b2** will never be created. However, they are created only when the *first* **Table** object is created. After that, the **static** object is not re-initialized.

The order of initialization is: **static**s first, if they haven't already been initialized by a previous object creation, and then the non-**static** objects. You can see the evidence of this in the output.

explicit static initialization

Java allows you to group other **static** initializations inside a special "**static** construction clause" in a class. It looks like this:

```
class Spoon {
  static int i;
  static {
    i = 47;
  }
  // . . .
```

So it looks like a method, but it's just the **static** keyword followed by a method body. This code, like the other **static** initialization, is only executed once, the first time you make an object of that class *or* you access a **static** member of that class (even if you never make an object of that class). For example:

```
//: ExplicitStatic.java
// Explicit static initialization
// with the "static" clause.

class Cup {
  Cup(int marker) {
    System.out.println("Cup(" + marker + ")");
  }
  void f(int marker) {
    System.out.println("f(" + marker + ")");
  }
}

class Cups {
  static Cup c1;
  static Cup c2;
  static {
    c1 = new Cup(1);
    c2 = new Cup(2);
  }
  Cups() {
    System.out.println("Cups()");
  }
}

public class ExplicitStatic {
  public static void main(String args[]) {
    System.out.println("Inside main()");
```

```
    Cups.c1.f(99);  // (1)
  }
  static Cups x = new Cups();  // (2)
  static Cups y = new Cups();  // (2)
} ///:~
```

The **static** initializers for **Cups** will be run when either the access of the **static** object **c1** occurs on the line marked (1), or if line (1) is commented out and the lines marked (2) are un-commented. If both (1) and (2) are commented out, the **static** initialization for **Cups** never occurs.

non-static instance initialization

Java 1.1 provides a similar syntax for initializing non-static variables for each object. Here's an example:

```
//: Mugs.java
// Java 1.1 "Instance Initialization"

class Mug {
  Mug(int marker) {
    System.out.println("Mug(" + marker + ")");
  }
  void f(int marker) {
    System.out.println("f(" + marker + ")");
  }
}

public class Mugs {
  Mug c1;
  Mug c2;
  {
    c1 = new Mug(1);
    c2 = new Mug(2);
    System.out.println("c1 & c2 initialized");
  }
  Mugs() {
    System.out.println("Mugs()");
  }
  public static void main(String args[]) {
    System.out.println("Inside main()");
    Mugs x = new Mugs();
  }
} ///:~
```

You can see that the instance initialization clause:

```
  {
    c1 = new Mug(1);
    c2 = new Mug(2);
    System.out.println("c1 & c2 initialized");
  }
```

looks just like the static initialization clause except for the missing **static** keyword. This syntax is necessary to support the initialization of *anonymous inner classes* (see Chapter 7).

array initialization

Initializing arrays in C is error-prone and tedious. C++ uses *aggregate initialization* to make it much safer[2]. Java has no "aggregates" like C++, since everything is an object in Java. However it does have arrays, and these are supported with array initialization.

An array is simply a sequence of objects, all the same type and packaged together under one identifier name. Arrays are defined and used with the square-brackets *indexing operator* **[ ]**. To define an array you just follow your identifier with empty square brackets:

```
int a1[];
```

However, you can also put the square brackets after the type name to produce exactly the same meaning:

```
int[] a1;
```

This might be considered a more sensible syntax, since it says that the type is "an **int** array." But the former style of definition conforms to expectations from C and C++ programmers.

The compiler doesn't allow you to tell it how big the array is. But we're back to that issue of "handles": all you have at this point is a handle to an array, and there's been no space allocated for the array itself. To create storage for the array you must write an initialization expression. For arrays, initialization can appear anywhere in your code, but you can also use a special kind of initialization expression that must occur at the point the array is created. This special initialization is a set of values surrounded by curly braces. The storage allocation (the equivalent of using **new**) will be taken care of by the compiler in this case. For example:

```
int a1[] = { 1, 2, 3, 4, 5 };
```

So why would you ever define an array handle without an array?

```
int a2[];
```

Well, it's possible to assign one array to another in Java, so you can say:

```
a2 = a1;
```

What you're really doing is copying a handle, as demonstrated here:

```
//: Arrays.java
// Arrays of primitives.

public class Arrays {
  public static void main(String args[]) {
    int a1[] = { 1, 2, 3, 4, 5 };
    int a2[];
    a2 = a1;
    for(int i = 0; i < a2.length; i++)
      a2[i]++;
    for(int i = 0; i < a1.length; i++)
      prt("a1[" + i + "] = " + a1[i]);
  }
  static void prt(String s) {
    System.out.println(s);
  }
} ///:~
```

[2] See *Thinking in C++* for a complete description of aggregate initialization.

You can see that **a1** is given an initialization value while **a2** is not; **a2** is assigned later – in this case, to another array.

There's something new here: all arrays have an intrinsic member (whether they're arrays of objects or arrays of primitives) that you can query – but not change – to tell you how many elements there are in the array: **length**. Since arrays in Java, like C and C++, start counting from element zero the largest element you can index is **length - 1**. If you go out of bounds C and C++ quietly accept this and allow you to stomp all over your memory, the source of many infamous bugs. However, Java protects you against such problems by causing a run-time error (called an *exception*, the subject of Chapter 9) if you step out of bounds. Of course, checking every array access costs time and code and there's no way to turn it off, which means that array accesses may be a source of inefficiency in your program if they occur at a critical juncture. For Internet security and programmer productivity, the Java designers felt this was a worthwhile tradeoff.

What if you don't know how many elements you're going to need in your array while you're writing the program? You simply use **new** to create the elements in the array. Here, **new** works even though it's creating an array of primitives (**new** won't create a non-array primitive):

```
//: ArrayNew.java
// Creating Arrays with new.
import java.util.*;

public class ArrayNew {
  static Random rand = new Random();
  static int pRand(int mod) {
    return Math.abs(rand.nextInt()) % mod;
  }
  public static void main(String args[]) {
    int a[];
    a = new int[pRand(20)];
    prt("length of a = " + a.length);
    for(int i = 0; i < a.length; i++)
      prt("a[" + i + "] = " + a[i]);
  }
  static void prt(String s) {
    System.out.println(s);
  }
} ///:~
```

Since the size of the array is chosen at random (using the **pRand( )** method defined earlier), it's clear that array creation is actually happening at run-time. In addition, you'll see from the output of this program that array elements of primitive types are automatically initialized to zero.

Of course, the array could also have been defined and initialized in the same statement:

```
int a[] = new int[pRand(20)];
```

If you're dealing with an array of class objects (non-primitives) you must always use **new**. Here, the handle issue comes up again because what you create is an array of handles. Consider the wrapper type **Integer** which is a class and not a primitive:

```
//: ArrayClassObj.java
// Creating an array of class objects.
import java.util.*;

public class ArrayClassObj {
  static Random rand = new Random();
  static int pRand(int mod) {
    return Math.abs(rand.nextInt()) % mod;
  }
  public static void main(String args[]) {
    Integer a[] = new Integer[pRand(20)];
```

```
    prt("length of a = " + a.length);
    for(int i = 0; i < a.length; i++) {
      a[i] = new Integer(pRand(500));
      prt("a[" + i + "] = " + a[i]);
    }
  }
  static void prt(String s) {
    System.out.println(s);
  }
} ///:~
```

Here, even after **new** is called to create the array:

```
Integer a[] = new Integer[pRand(20)];
```

it's just an array of handles, and not until the handle itself is initialized by creating a new **Integer** object is the initialization complete:

```
a[i] = new Integer(pRand(500));
```

If you forget to create the object, however, you'll get an exception at run-time.

Take a look at the formation of the **String** object inside the print statements. You can see that the handle to the **Integer** object is automatically converted to produce a **String** representing the value inside the object.

It's also possible to initialize arrays of objects using the curly-brace-enclosed list. There are two forms, the first of which is the only one allowed in Java 1.0. The second (equivalent) form is allowed in Java 1.1:

```
//: ArrayInit.java
// Array initialization

public class ArrayInit {
  public static void main(String args[]) {
    Integer[] a = {
      new Integer(1),
      new Integer(2),
      new Integer(3),
    };

    Integer[] b = new Integer[] {
      new Integer(1),
      new Integer(2),
      new Integer(3),
    };
  }
} ///:~
```

This is useful at times, but it's more limited since the size of the array is determined at compile time. The last comma in the list of initializers is optional (this feature makes for easier maintenance of long lists).

Java allows you to easily create multidimensional arrays:

```
//: MultiDimArray.java
// Creating multi-dimensional arrays.
import java.util.*;

public class MultiDimArray {
  static Random rand = new Random();
  static int pRand(int mod) {
```

```
    return Math.abs(rand.nextInt()) % mod;
  }
  public static void main(String args[]) {
    int a1[][] = {
      { 1, 2, 3, },
      { 4, 5, 6, },
    };
    for(int i = 0; i < a1.length; i++)
      for(int j = 0; j < a1[i].length; j++)
        prt("a1[" + i + "][" + j +
            "] = " + a1[i][j]);
    // 3-D array with fixed length:
    int a2[][][] = new int[2][2][4];
    for(int i = 0; i < a2.length; i++)
      for(int j = 0; j < a2[i].length; j++)
        for(int k = 0; k < a2[i][j].length;
            k++)
          prt("a2[" + i + "][" +
              j + "][" + k +
              "] = " + a2[i][j][k]);
    // 3-D array with varied-length vectors:
    int a3[][][] = new int[pRand(7)][][];
    for(int i = 0; i < a3.length; i++) {
      a3[i] = new int[pRand(5)][];
      for(int j = 0; j < a3[i].length; j++)
        a3[i][j] = new int[pRand(5)];
    }
    for(int i = 0; i < a3.length; i++)
      for(int j = 0; j < a3[i].length; j++)
        for(int k = 0; k < a3[i][j].length;
            k++)
          prt("a3[" + i + "][" +
              j + "][" + k +
              "] = " + a3[i][j][k]);
    // Array of class objects:
    Integer[][] a4 = {
      { new Integer(1), new Integer(2)},
      { new Integer(3), new Integer(4)},
      { new Integer(5), new Integer(6)},
    };
    for(int i = 0; i < a4.length; i++)
      for(int j = 0; j < a4[i].length; j++)
        prt("a4[" + i + "][" + j +
            "] = " + a4[i][j]);
    Integer[][] a5;
    a5 = new Integer[3][];
    for(int i = 0; i < a5.length; i++) {
      a5[i] = new Integer[3];
      for(int j = 0; j < a5[i].length; j++)
        a5[i][j] = new Integer(i*j);
    }
    for(int i = 0; i < a5.length; i++)
      for(int j = 0; j < a5[i].length; j++)
        prt("a5[" + i + "][" + j +
            "] = " + a5[i][j]);
  }
  static void prt(String s) {
    System.out.println(s);
  }
} ///:~
```

The code used for printing uses **length** so that it doesn't depend on fixed array sizes.

The first example shows a multi-dimensional array of primitives. You delimit each vector in the array with curly braces:

```
int a1[][] = {
  { 1, 2, 3, },
  { 4, 5, 6, },
};
```

Each set of square brackets moves you into the next level of the array.

The second example shows a three-dimensional array allocated with **new**. Here, the whole array is allocated all at once:

```
int a2[][][] = new int[2][2][4];
```

But the third example shows that each vector in the arrays that make up the matrix can be of any length:

```
int a3[][][] = new int[pRand(7)][][];
for(int i = 0; i < a3.length; i++) {
  a3[i] = new int[pRand(5)][];
  for(int j = 0; j < a3[i].length; j++)
    a3[i][j] = new int[pRand(5)];
}
```

The first **new** creates an array with a random-length first element and the rest undetermined. The second **new** inside the for loop fills out the elements but leaves the third index undetermined until you hit the third **new**.

You will see from the output that array values are automatically initialized to zero if you don't give them an explicit initialization value.

You can deal with arrays of class objects in a similar fashion, which is shown in the fourth example, demonstrating the ability to collect many **new** expressions with curly braces:

```
Integer[][] a4 = {
  { new Integer(1), new Integer(2)},
  { new Integer(3), new Integer(4)},
  { new Integer(5), new Integer(6)},
};
```

The fifth example shows how an array of class objects can be built up piece by piece:

```
Integer[][] a5;
a5 = new Integer[3][];
for(int i = 0; i < a5.length; i++) {
  a5[i] = new Integer[3];
  for(int j = 0; j < a5[i].length; j++)
    a5[i][j] = new Integer(i*j);
}
```

The **i*j** is just to put an interesting value into the **Integer**.

summary

The seemingly elaborate mechanism for initialization – the constructor – provided by Java should give you a strong hint about the critical importance placed on initialization in the language. As Stroustrup was designing C++, one of the first observations he made about productivity in C was that a very significant portion of programming problems are caused by improper initialization of variables. These

kinds of bugs are very hard to find, and similar issues apply to improper cleanup. Because constructors allow you to *guarantee* proper initialization and cleanup (the compiler will not allow an object to be created without the proper constructor calls), you get complete control and safety.

In C++, destruction is quite important because objects created with **new** must be explicitly destroyed. In Java, memory for all objects is automatically released by the garbage collector, so the equivalent cleanup method in Java isn't necessary much of the time. Thus (in cases where you don't need destructor-like behavior) Java's garbage collector greatly simplifies programming, and adds much-needed safety in managing memory. Some garbage collectors are even cleaning up other resources like graphics and file handles. However, the garbage collector does add a run-time cost, the expense of which is difficult to put into perspective because of the overall slowness of Java interpreters at this writing. As this changes, we'll be able to discover if the overhead of the garbage collector will preclude the use of Java for certain types of programs (one of the issues is the unpredictability of the garbage collector).

Because of the guarantee that all objects will be constructed, there's actually more to the constructor than what is shown here. In particular, when you create new classes using either *composition* or *inheritance* the guarantee of construction also holds, and some additional syntax is necessary to support this. You'll learn about composition, inheritance and how they affect constructors in future chapters.

exercises

1. Create a class with a default constructor (one that takes no arguments) that prints a message. Create an object of this class.

2. Add an overloaded constructor to exercise 1 that takes a String argument and prints it along with your message.

3. Create an array of object handles only of the class you created in exercise two. When you run the program, notice whether the initialization messages from the constructor calls are printed.

4. Complete exercise 3 by creating objects to attach to the array of handles.

5. Experiment with **Garbage.java** by running the program using the arguments "before," "after" and "none." Repeat the process and see if you detect any patterns in the output. Change the code so that **System.RunFinalization( )** is called before **System.gc( )** and observe the results.

5: hiding the implementation

[[[Chapter 5 directory:c05]]]

A primary consideration in object-oriented design is "separating the things that change from the things that stay the same"

This is particularly important for libraries: the user of that library (also called the *client programmer*) must be able to rely on the part they are using, and know that they won't have to rewrite code if a new version of the library comes out. And on the flip side, the library creator must have the freedom to make modifications and improvements with the certainty that the client programmer's code won't be affected by those changes.

This can be achieved through convention. For example, The library programmer must agree not to remove existing methods when modifying a class in the library, since that would break the client programmer's code. The reverse situation is thornier, however. In the case of a data member, how can the library creator know which data members have been accessed by client programmers? This is also true with methods that are only part of the implementation of a class, and not meant to be used directly by the client programmer. But what if the library creator wants to rip out an old implementation and put in a new one? Changing any of those members may break some client programmer's code. Thus the library creator is in a straight jacket and can't change anything.

To solve this problem, Java provides *access specifiers* to allow the library creator to say: this is available to the client programmer, this is not. The levels of access control from "most access" to "least access" are **public**, "friendly" (which has no keyword), **protected**, and **private**. From the previous paragraph you may think that, as a library designer, you'll want to keep everything as "private" as possible, and only expose the methods that you want the client programmer to use. This is exactly right, even though it's often counterintuitive for people who program in other languages (especially C) and are used to accessing everything without restriction. By the end of this chapter you should be convinced of the value of access control in Java.

The concept of a library of components and the control over who can access the components of that library is not complete, however. There's still the question of how the components are bundled together into a cohesive library unit. This is controlled with the **package** keyword in Java, and the access specifiers are affected by whether a class is in the same package or in a separate package. So to begin this chapter, you'll learn how library components are placed into packages. Then you'll be able to understand the complete meaning of the access specifiers.

package: the library unit

A package is what you get when you use the **import** keyword to bring in an entire library, such as

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

This brings in the entire utility library that's part of the standard Java distribution. Since **Vector** is in **java.util**, you can now either specify the full name **java.util.Vector** (which you can do without the **import** statement) or you can just say **Vector** (because of the **import**).

If you want to bring in a single class, you can just name that class in the **import** statement

```
import java.util.Vector;
```

Now you can use **Vector** with no qualification. However, none of the other classes in **java.util** are available.

The reason for all this importing is to provide a mechanism to manage "name spaces." The names of all your class members are insulated from each other: a method **f( )** inside a class **A** will not clash with an **f( )** that has the same signature (argument list) in class **B**. But what about the class names themselves? Suppose you create a **stack** class which is installed on a machine that already has a **stack** class that's written by someone else? With Java on the Internet, this can happen without the user knowing it since classes can get downloaded automatically in the process of running a Java program.

This potential clashing of names is why it's important to have complete control over the name spaces in Java, and to be able to create a completely unique name regardless of the constraints of the Internet.

So far, the examples in this book have existed in a single file and have been designed for local use, and haven't bothered with package names (in this case the class name is placed in the "default package"). This is certainly an option, and for simplicity's sake this approach will be used whenever possible throughout the rest of the book. However, if you're planning to create a program that is "Internet friendly" you'll need to think about preventing class name clashes.

When you create a source-code file for Java, it's commonly called a *compilation unit* (sometimes a *translation unit*). Each compilation unit must have a name followed by **.java**, and inside the compilation unit there can be a public class that must have the same name as the file (including capitalization, but excluding the **.java** filename extension). If you don't do this, the compiler will complain. There can only be *one* **public** class in each compilation unit (or the compiler will complain). The rest of the classes in that compilation unit, if there are any, are hidden from the world outside that package because they're *not* **public**, and they comprise "support" classes for the main **public** class.

When you compile a **.java** file you get an output file with exactly the same name but an extension of **.class** *for each class in the* **.java** file. Thus you can end up with quite a few **.class** files from a small number of **.java** files. If you've programmed with a compiled language, you may be used to the compiler spitting out an intermediate form (usually an "Obj" file) that is then packaged together with others of its kind using a linker (to create an executable file) or a librarian (to create a library). That's not how Java works. A working program is a bunch of **.class** files, which may be packaged and compressed into a JAR file (using the **jar** utility in Java 1.1). The Java interpreter is responsible for finding, loading and interpreting these files[1].

A library is also a bunch of these class files – each file has one class that is **public**, so there's one component for each file. But how do you say that all these components (that are in their own separate **.java** and **.class** files) belong together? That's where the **package** keyword comes in.

When you say:

```
package mypackage;
```

You're stating that this compilation unit is part of a library named **mypackage**. Or, put another way, you're saying that the **public** class name within this compilation unit is under the umbrella of the name **mypackage**, and if anyone wants to use the name they'll either have to fully specify the name or use the **import** keyword in combination with **mypackage** (using the choices given previously). Note that the convention for Java packages is to use all lowercase letters, even for intermediate words.

For example, suppose the name of the file is **MyClass.java**. This means there can be one and only one **public** class in that file, and the name of that class must be **MyClass** (including the capitalization):

```
package mypackage;

public class MyClass {
  // . . .
```

Now, if someone wants to use **MyClass** or, for that matter, any of the other **public** classes in **mypackage**, they must use the **import** keyword to make the name or names in **mypackage** available. The alternative is to give the fully-qualified name:

```
mypackage.MyClass m = new mypackage.MyClass();
```

The **import** keyword can make this much cleaner:

```
import mypackage.*;
// . . .

MyClass m = new MyClass();
```

It's worth keeping in mind that what the **package** and **import** keywords allow you to do, as a library designer, is to divide up the single global name space so you won't have clashing names, no matter how many people get on the Internet and start writing classes in Java.

creating unique package names

You may observe that, since a package never really gets "packaged" into a single file, a package might be made up of many **.class** files, and things could get a bit cluttered. To prevent this, a logical thing to do is to place all the **.class** files for a particular package into a single directory; that is, to use the hierarchical file structure of the operating system to advantage. This is how Java handles the problem of clutter.

It also solves two other problems: creating unique package names, and finding those classes that may be buried in a directory structure someplace. This is accomplished, as was introduced in Chapter 2, by

[1] There's nothing in Java that forces the use of an interpreter. There exist native-code Java compilers that generate a single executable file.

encoding the path of the location of the **.class** file into the name of the **package**. The compiler enforces this, but in addition, by convention, the first part of the **package** name is the Internet domain name of the creator of the class, reversed. Since Internet domain names are guaranteed to be unique (by Internic[2], who controls their assignment) *if* you follow this convention it's guaranteed that your **package** name will be unique and thus you'll never have a name clash. Of course, if you don't have your own domain name then you'll need to fabricate an unlikely combination (such as your first and last name) to create unique package names. However, if you've decided to start publishing Java code it's worth the relatively small effort to get a domain name.

The second part of this trick is resolving the **package** name into a directory on your machine, so when the Java program is running and it needs to load the **.class** file (which it may do dynamically, at the point in the program where it needs to create an object of that particular class), it can locate the directory where the **.class** file resides.

The Java interpreter proceeds as follows: first, it finds the environment variable CLASSPATH (set via the operating system when Java, or a tool like a Java-enabled browser, is installed on a machine). CLASSPATH contains one or more directories that may be used as roots for a search for **.class** files. Starting at that root, the interpreter will take the package name and replace each dot with a slash to generate a path name from the CLASSPATH root (so **package foo.bar.baz** becomes **foo\bar\baz** or **foo/bar/baz** depending on your OS. This is then concatenated to the various entries in the CLASSPATH). That's where it looks for the **.class** file with the name corresponding to the class you're trying to create.

To understand this, you'll need to study an example. Consider my domain name, which is **EckelObjects.com**. By reversing this, **COM.EckelObjects** (the COM, EDU, ORG, etc. extension is capitalized by Java convention) establishes my unique global name for my classes. I can further subdivide this by deciding I want to create a library named **util**, so I'll end up with a package name:

```
package COM.EckelObjects.util;
```

Now this package name can be used as an umbrella name space for the following two files:

```
//: Vector.java
// Creating a package
package COM.EckelObjects.util;

public class Vector {
  public Vector() {
    System.out.println(
      "COM.EckelObjects.util.Vector");
  }
} ///:~
```

When you create your own packages, you'll discover that the **package** statement must be the first non-comment code in the file. The second file looks much the same:

```
//: List.java
// Creating a package
package COM.EckelObjects.util;

public class List {
  public List() {
    System.out.println(
      "COM.EckelObjects.util.List");
  }
} ///:~
```

Both of these files are placed in the subdirectory on my system:

[2] ftp://ftp.internic.net

```
C:\DOC\JavaT\COM\EckelObjects\util
```

If you walk back through this, you can see the package name **COM.EckelObjects.util**, but what about the first portion of the path? That's taken care of in the CLASSPATH environment variable, which is, on my machine:

```
CLASSPATH=.;D:\JAVA\LIB;C:\DOC\JavaT
```

You can see that the CLASSPATH can contain a number of alternative search paths.

Now, the following file can be placed in any directory:

```
//: LibTest.java
// Uses the library
import COM.EckelObjects.util.*;

public class LibTest {
  static public void main(String args[]) {
    Vector v = new Vector();
    List l = new List();
  }
} ///:~
```

When the compiler encounters the **import** statement, it begins searching at the directories specified by CLASSPATH, looking for a subdirectory COM\EckelObjects\util, then seeking the compiled files of the appropriate names (**Vector.class** for **Vector** and **List.class** for **List**). Note that both the classes and the desired methods in **Vector** and **List** must be **public**.

automatic compilation

The first time you create an object of an imported class, the compiler will go hunting for the **.class** file of the same name (so if you're creating an object of class **X**, it looks for **X.class**) in the appropriate directory. If it only finds **X.class**, that's what it must use. However, if it also finds an **X.java** in the same directory, the compiler will first compare the date stamp on the two files, and if **X.java** is more recent than **X.class**, it will *automatically recompile* **X.java** to generate an up-to-date **X.class**.

collisions

What happens if two libraries are imported via * and they include the same names? For example, suppose a program does this:

```
import COM.EckelObjects.util.*;
import java.util.*;
```

Since **java.util.*** also contains a **Vector** class, this causes a potential collision. However, as long as the collision doesn't actually occur, everything is OK – which is good because otherwise you might end up doing a lot of typing to prevent collisions that would never happen.

The collision *does* occur if you now try to make a **Vector**:

```
Vector v = new Vector();
```

Which **Vector** class does this refer to? The compiler can't know, and the reader can't know either. So the compiler complains and forces you to be explicit. If I want the standard Java **Vector**, for example, I must say:

```
java.util.Vector v = new java.util.Vector();
```

Since this (along with the CLASSPATH) completely specifies the location of that **Vector**, there's no need for the **import java.util.*** statement unless I'm using something else from **java.util**.

a custom tool library

With this knowledge in hand, you can now create your own libraries of tools to reduce or eliminate duplicate code. Consider, for example, creating an alias for **System.out.println( )** to reduce typing. This can be part of a package called **tools**:

```
//: P.java
// The P.rint & P.rintln shorthand
package COM.EckelObjects.tools;

public class P {
  public void rint(Object obj) {
    System.out.print(obj);
  }
  public static void rint(String s) {
    System.out.print(s);
  }
  public static void rint(char s[]) {
    System.out.print(s);
  }
  public static void rint(char c) {
    System.out.print(c);
  }
  public static void rint(int i) {
    System.out.print(i);
  }
  public static void rint(long l) {
    System.out.print(l);
  }
  public static void rint(float f) {
    System.out.print(f);
  }
  public static void rint(double d) {
    System.out.print(d);
  }
  public static void rint(boolean b) {
    System.out.print(b);
  }
  public static void rintln() {
    System.out.println();
  }
  public static void rintln(Object obj) {
    System.out.println(obj);
  }
  public static void rintln(String s) {
    System.out.println(s);
  }
  public static void rintln(char s[]) {
    System.out.println(s);
  }
  public static void rintln(char c) {
    System.out.println(c);
  }
  public static void rintln(int i) {
    System.out.println(i);
  }
  public static void rintln(long l) {
    System.out.println(l);
  }
  public static void rintln(float f) {
```

```
    System.out.println(f);
  }
  public static void rintln(double d) {
    System.out.println(d);
  }
  public static void rintln(boolean b) {
    System.out.println(b);
  }
} ///:~
```

All the different data types can now be printed out either with a newline (**P.rintln( )**) or without a newline (**P.rint( )**).

You can guess that the location of this file must be in a directory that starts at one of the CLASSPATH locations, then continues **COM/EckelObjects/tools**. After compiling, the **P.class** file can be used anywhere on your system with an **import** statement:

```
//: ToolTest.java
// Uses the tools library
import COM.EckelObjects.tools.*;

public class ToolTest {
  static public void main(String args[]) {
    P.rintln("Available from now on!");
  }
} ///:~
```

So from now on, whenever you come up with a useful new utility, you can add it to the **tools** directory (or to your own personal **util** or **tools** directory).

classpath pitfall

The **P.java** file brought up an interesting pitfall. Especially with early implementations of Java, setting the classpath correctly was generally quite a headache. During the development of this book, the **P.java** file was introduced and seemed to work fine, but at some point it began breaking. For a long time I was certain this was the fault of one implementation of Java or another, but finally I discovered that at one point I had introduced a program (**CodePackager.java**, shown in Chapter 17) that also used a different class **P**. Because it was used as a tool, it was *sometimes* placed in the classpath, and other times it wasn't. When it was, the **P** in **CodePackager.java** was found first by Java when executing a program where it was looking for the class in **COM.EckelObjects.tools**, and the compiler would say that a particular method couldn't be found. This was very frustrating, because you can see the method in the above class **P**, and no further diagnostics were reported to give you a clue that is was finding a completely different class (that wasn't even **public**).

At first this could seem like a compiler bug, but if you look at the **import** statement it just says "here's where you *might* find **P**." However, the compiler is supposed to look anywhere in its classpath so if it finds a **P** there it will use it, and if it finds the "wrong" one *first* during a search then it will stop looking.

To avoid a highly frustrating experience, make sure there's only one class of each name anywhere in your classpath.

package caveat

It's worth remembering that anytime you create a package, you implicitly specify a directory structure when you give the package a name. The package *must* live in the directory indicated by its name, which must be a directory that is searchable starting from the CLASSPATH. This means that experimenting with the **package** keyword can be a bit frustrating at first because unless you adhere to the package-name to directory-path rule, you'll get a lot of mysterious run-time messages about not being able to find a particular class, even if that class is sitting there in the same directory. If you get such a message, try commenting out the **package** statement, and if it runs you'll know where the problem lies.

Java access specifiers

The Java access specifiers **public**, **protected** and **private** are placed in front of each definition for each member in your class, whether it's a data member or a method. Each access specifier only controls the access for that particular definition. This is a distinct contrast with C++, where the access specifier controls all the definitions following it, until another access specifier comes along.

One way or another everything has some kind of access specified for it. In the following sections, you'll learn all about the various types of access, starting with the default access.

"friendly"

What if you give no access specifier at all, as in all the examples before this chapter? The default access has no keyword, but it is commonly referred to as "friendly." It means that all the other classes in the current package have access to the friendly member, but to all the classes outside of this package the member appears to be private. Since a compilation unit – a file – can only belong to a single package, all the classes within a single compilation unit are automatically friendly with each other.

Friendly access allows you to group related classes together in a package so they can easily interact with each other. When you put classes together in a package (thus granting mutual access to their friendly members; e.g. making them "friends") you "own" the code in that package. It makes sense that only code that you own should have friendly access to other code that you own. You could say that friendly access gives a meaning or a reason for grouping classes together in a package. In many languages the way you organize your definitions in files can be willy-nilly, but in Java you're compelled to organize them in a sensible fashion. In addition, you'll probably want to exclude classes that shouldn't have access to the classes being defined in the current package.

A very important rule in any relationship is "who can access my private implementation?" The class controls which code has access to its members. There's no magic way to "break in"; someone in another package can't declare a new class and say "hi, I'm a friend of **Bob**!" and expect to see the **protected**, friendly, and **private** members of **Bob**. Thus, the only way to grant access to a member is to

1. Make the member **public**. Then everybody, everywhere, can access it.
2. Make the member friendly by leaving off any access specifier, and put the other classes in the same package. Then the other classes can access the member.
3. As you'll see in a later chapter where inheritance is introduced, an inherited class can access a **protected** member as well as **public** member (but not **private** members). It can only access friendly members if the two classes are in the same package. But don't worry about that now.
4. Provide "accessor/mutator" methods (a.k.a. "get/set" methods) that read and change the value. This is the most civilized approach in terms of OOP, and it is fundamental, for example, to JavaBeans, as you'll see in Chapter 13.

public: interface access

When you use the **public** keyword, it means that the member declaration that immediately follows **public** is available to everyone, in particular to the client programmer who is using the library. Suppose you define a package **dessert** containing the following compilation unit:

```
//: Cookie.java
// Creates a library
package c05.dessert;

public class Cookie {
  public Cookie() {
```

```
    System.out.println("Cookie constructor");
  }
  void foo() { System.out.println("foo"); }
} ///:~
```

Remember, **Cookie.java** must reside in a subdirectory called **dessert**, in a directory under **C05** (indicating Chapter 5 of this book) which itself must be under one of the CLASSPATH directories. Don't make the mistake of thinking that Java will always look at the current directory as one of the starting points for searching: if you don't have a '**.**' as one of the paths in your CLASSPATH, Java won't look there.

Now if you create a program that uses **Cookie**:

```
//: Dinner.java
// Uses the library
import c05.dessert.*;

public class Dinner {
  public Dinner() {
   System.out.println("Dinner constructor");
  }
  public static void main(String args[]) {
    Cookie x = new Cookie();
    //! x.foo(); // Can't access
  }
} ///:~
```

You can create a **Cookie** object since its constructor is public and the class itself is public (we'll look more at the concept of a public class later). However, the **foo( )** member is inaccessible inside **Dinner.java** since **foo( )** is only friendly within package **dessert**.

the unnamed package

You may be surprised to discover that the following code compiles, even though it would appear that it breaks the rules:

```
//: Cake.java
// Accesses a class in a separate
// compilation unit.

class Cake {
  public static void main(String args[]) {
    Pie x = new Pie();
    x.f();
  }
} ///:~
```

In a second file, in the same directory:

```
//: Pie.java
// The other class

class Pie {
  void f() { System.out.println("Pie.f()"); }
} ///:~
```

You might initially view these as completely foreign files, and yet **Cake** is able to create a **Pie** object and call its **f( )** method! You'd normally think that **Pie** and **f( )** are friendly and therefore not available to **Cake**. They *are* friendly – that part is correct. The reason they are available in **Cake.java** is because they are in the same directory and have no explicit package name. Java treats files like this as implicitly part of the "unnamed package" for that directory, and therefore friendly to all the other files

in that directory. Note, however, if you try to compile the files by invoking **javac** from *outside* the directory where **Pie.java** and **Cake.java** live, you'll get error messages.

private: you can't touch that!

The **private** keyword means no one can access that member except that particular class, inside methods of that class. Other classes in the same package cannot access **private** members, so it's as if you're even insulating the class against yourself. On the other hand, it's not unlikely that a package might be created by several people collaborating together, so **private** allows you to freely change that member without concern that it will affect another class in the same package. The default "friendly" package access is often an adequate amount of hiding – remember, a "friendly" member is inaccessible to the user of the package. This is nice, since the default access is the one that you normally use. Thus, you'll typically think about access primarily for the members that you explicitly want to make **public** for the client programmer, and as a result you may not initially think you'll use the **private** keyword very often since it's tolerable to get away without it (this is a distinct contrast with C++). However, it turns out that the consistent use of **private** is very important, especially where multithreading is concerned (as you'll see in Chapter 14).

Here's an example of the use of **private**:

```
//: IceCream.java
// Demonstrates "private" keyword

class Sundae {
  private Sundae() {}
  static Sundae makeASundae() {
    return new Sundae();
  }
}

public class IceCream {
  public static void main(String args[]) {
    //! Sundae x = new Sundae();
    Sundae x = Sundae.makeASundae();
  }
} ///:~
```

This shows an example where **private** comes in handy: you may want to control how an object is created, and prevent anyone from directly accessing a particular constructor (or all of them). In the above example, you cannot create a **Sundae** object via its constructor – instead you must call the **makeASundae( )** method to do it for you.

Any method that you're certain is only a "helper" method for that class can be made **private** to ensure that you don't accidentally use it elsewhere in the package and thus prohibit you from changing or removing the method. Making a method **private** guarantees that you retain this option.

protected: "sort of private"

The **protected** access specifier requires a jump ahead to understand. So first, you should be aware that you don't need to understand this section to continue through the book up through the inheritance chapter. But for completeness an example using **protected** will be briefly described.

The **protected** keyword deals with a concept called *inheritance*, which takes an existing class and adds new members to that class without touching the existing class, which we refer to as the *base class*. You can also change the behavior of existing members of the class. To inherit from an existing class, you say that your new class **extends** an existing class, like this:

```
class Foo extends Bar {
```

The rest of the class definition looks the same.

If you create a new package and you inherit from a class in another package, the only members you have access to are the **public** members of the original package (of course, if you perform the inheritance in the *same* package you have the normal package access to all the "friendly" members). Sometimes the creator of the base class would like to take a particular member and grant access to derived classes but not the world in general. That's what **protected** does. If you refer back to the file **Cookie.java** on page 128, the following class *cannot* access the "friendly" member:

```
//: ChocolateChip.java
// Can't access friendly member
// in another class
import c05.dessert.*;

public class ChocolateChip extends Cookie {
  public ChocolateChip() {
   System.out.println(
     "ChocolateChip constructor");
  }
  public static void main(String args[]) {
    ChocolateChip x = new ChocolateChip();
    //! x.foo(); // Can't access foo
  }
} ///:~
```

One of the interesting things about inheritance is that if a method **foo( )** exists in class **Cookie**, then it also exists in any class inherited from **Cookie**. But since **foo( )** is "friendly" in a foreign package, it's unavailable to us in this one. Of course, you could make it **public**, but then everyone would have access and maybe that's not what you want. If we change the class **Cookie** as follows:

```
public class Cookie {
  public Cookie() {
    System.out.println("Cookie constructor");
  }
  protected void foo() {
    System.out.println("foo");
  }
}
```

Then **foo( )** still has "friendly" access within package **dessert**, but it is also accessible to anyone inheriting from **dessert**. However, it is *not* **public**.

interface & implementation

Access control is often referred to as *implementation hiding*. Wrapping data and methods within classes (combined with implementation hiding this is often called *encapsulation*) produces a data type with characteristics and behaviors, but access control puts boundaries within that data type, for two important reasons. The first is to establish what the client programmers can and can't use: you can build your internal mechanisms into the structure without worrying that the client programmers will think it's part of the interface they should be using.

This feeds directly into the second reason, which is to separate the interface from the implementation. If the structure is used in a set of programs, but users can't do anything but send messages to the **public** interface, then you can change anything that's *not* **public** (e.g. "friendly," **protected** or **private**) without requiring modifications to their code.

We're now in the world of object-oriented programming, where a **class** is actually describing "a class of objects," as you would describe a class of fishes or a class of birds. Any object belonging to this class will share these characteristics and behaviors. The class is a description of the way all objects of this type will look and act.

In the original OOP language, Simula-67, the keyword **class** was used to describe a new data type. The same keyword has been used for most object-oriented languages. This is the focal point of the whole language: the creation of new data types that are more than just boxes containing data and methods.

The **class** is the fundamental OOP concept in Java. It is one of the keywords that will *not* be set in bold in this book — it becomes annoying with a word repeated as often as "class."

For clarity, you may prefer a style of creating classes that places the **public** members at the beginning, followed by the **protected**, friendly, and **private** members. The advantage of this is that the user of the class can then read down from the top and see first what's important to them (the **public** members, because they can be accessed outside the file) and stop reading when they encounter the non-public members, which are part of the internal implementation. However, with the comment-documentation supported by javadoc (described in Chapter 2) the issue of code readability by the client programmer becomes less important.

```
public class X {
  public void pub1( ) { /* . . . */ }
  public void pub2( ) { /* . . . */ }
  public void pub3( ) { /* . . . */ }
  private void priv1( ) { /* . . . */ }
  private void priv2( ) { /* . . . */ }
  private void priv3( ) { /* . . . */ }
  private int i;
  // . . .
}
```

However, this will only make it partially easier to read because the interface and implementation are still mixed together. That is, you still see the source code – the implementation – because it's right there in the class. Displaying the interface to the consumer of a class is really the job of the *browser*, a tool whose job it is to look at all the available classes and show you what you can do with them (what members are available) in a useful fashion. By the time you read this, good browsers should be an expected part of any good Java development tool.

class access

In Java, the access specifiers can also be used to determine which classes *within* a library will be available to the users of that library. If you want a class to be available to a client programmer, you place the **public** keyword somewhere before the opening brace of the class body. This controls whether the client programmer can even create an object of the class.

To control the access of a class, the specifier must appear before the keyword **class**. Thus you can say:

```
public class Widget {
```

That is, if the name of your library is **mylib** any client programmer can access **Widget** by saying

```
import mylib.Widget;
```

or

```
import mylib.*;
```

However, there's an extra pair of constraints:

1. There can only be one **public** class per compilation unit (file). The idea is that each compilation unit has a single public interface represented by that public class. It can have as many supporting "friendly" classes as you want. If you have more than one **public** class inside a compilation unit, the compiler will give you an error message.

2. The name of the **public** class must exactly match the name of the file containing the compilation unit, including capitalization. So for **Widget**, the name of the file must be

Widget.java, not **widget.java** or **WIDGET.java**. Again, you'll get a compile-time error if they don't agree.

What if you've got a class inside **mylib** that you're just using to accomplish the tasks performed by **Widget** or some other **public** class in **mylib**? You don't want to go to the bother of creating documentation for the client programmer, and you think that sometime later you may want to completely change things and rip out your class altogether, substituting a different one. To give you this flexibility, you need to ensure that no client programmers become dependent on your particular implementation details hidden inside **mylib**. To accomplish this, you just leave the **public** keyword off the class, in which case it becomes friendly (that class can only be used within that package).

Note that a class cannot be **private** (that would make it accessible to no one but the class itself), or **protected**. So you only have two choices for class access: "friendly" or **public**. If you don't want anyone else to have access to that class, you can make all the constructors **private**, thereby preventing anyone but you, inside a **static** member of the class, from creating an object of that class. Here's an example:

```
//: Lunch.java
// Demonstrates class access specifiers
// Make a class effectively private
// with private constructors:

class Soup {
  private Soup() {}
  // (1) Allow creation via static method:
  public static Soup makeSoup() {
    return new Soup();
  }
  // (2) Create a static object and
  // return a reference upon request.
  // (The "Singleton" pattern):
  private static Soup ps1 = new Soup();
  public static Soup access() {
    return ps1;
  }
  public void f() {}
}

class Sandwich {
  void f() { new Lunch(); }
}

// Only one public class allowed per file:
public class Lunch {
  void test() {
    // Can't do this! Private constructor:
    //! Soup priv1 = new Soup();
    Soup priv2 = Soup.makeSoup();
    Sandwich f1 = new Sandwich();
    Soup.access().f();
  }
} ///:~
```

Up to now, most of the methods have been returning either **void** or a primitive type so the definition:

```
  public static Soup access() {
    return ps1;
  }
```

may look a little confusing at first. The word before the method name (**access**) tells what the method returns. So far this has most often been **void** which means it returns nothing, but you can also return a

handle to an object which is what happens here. This method returns a handle to an object of class **Soup**.

The **class Soup** shows how to prevent direct creation of a class by making all the constructors **private**. Remember that if you don't explicitly create at least one constructor, the default constructor (a constructor with no arguments) will be created for you. By writing the default constructor, it won't be created automatically. By making it **private**, no one can create an object of that class. But now how does anyone use this class? The above example shows two options. First, a **static** method is created that creates a new **Soup** and returns a handle to it. This could be useful if you want to do some extra operations on the **Soup** before returning it, or if you want to keep count of how many **Soup** objects to create (perhaps to restrict their population).

The second option uses what's called a *design pattern*, which will be discussed later in this book. This particular pattern is called a "singleton" because it only allows a single object to be created. The object of class **Soup** is created as a **static private** member of **Soup**, so there's one and only one, and you can't get at it except through the **public** method **access( )**.

As previously mentioned, if you don't put an access specifier for class access, it defaults to "friendly." This means an object of that class can be created by any other class in the package, but not outside the package (remember, all the files within the same directory that don't have explicit **package** declarations are implicitly part of the unnamed package for that directory). However, if a **static** member of that class is **public**, the client programmer can still access that **static** member, even though they cannot create an object of that class.

summary

In any relationship it's important to have boundaries that are respected by all parties involved. When you create a library, you establish a relationship with the user of that library – the client programmer – who is another programmer, but one putting together an application or using your library to build a bigger library.

Without rules, client programmers can do anything they want with all the members of a class, even if you might really prefer they don't directly manipulate some of the members. Everything's naked to the world.

This chapter looked at how classes are built to form libraries; first the way a group of classes is packaged within a library, and second the way the class itself controls access to its members.

It is estimated that a C programming project begins to break down somewhere between 50K - 100K lines of code because C has a single "name space" so names begin to collide, causing an extra management overhead. In Java, the **package** keyword, the package naming scheme and the **import** keyword give you complete control over names, so the issue of name collision is easily avoided.

There are two reasons for controlling access to members. The first is to keep users' hands off tools they shouldn't touch, tools that are necessary for the internal machinations of the data type, but not part of the interface that users need to solve their particular problems. So making methods and fields private is actually a service to users because they can easily see what's important to them and what they can ignore. It simplifies their understanding of the class.

The second and most important reason for access control is to allow the library designer to change the internal workings of the class without worrying about how it will affect the client programmer. You may build a class one way at first, and then discover that restructuring your code will provide much greater speed. If the interface and implementation are clearly separated and protected, you can accomplish this without forcing the user to rewrite their code.

Access specifiers in Java give valuable control to the creator of a class. The users of the class can clearly see exactly what they can use and what to ignore. More important, though, is the ability to ensure that no user becomes dependent on any part of the underlying implementation of a class. If you know this as the creator of the class, you can change the underlying implementation with the knowledge that no client programmer will be affected by the changes because they can't access that part of the class.

When you have the ability to change the underlying implementation, you can not only improve your design at some later time, but you also have the freedom to make mistakes. No matter how carefully you plan and design, you'll make mistakes. Knowing that it's relatively safe to make these mistakes means you'll be more experimental, you'll learn faster, and you'll finish your project sooner.

The public interface to a class is what the user *does* see, so that is the most important part of the class to get "right" during analysis and design. But even that allows you some leeway for change. If you don't get the interface right the first time, you can *add* more methods, as long as you don't remove any that client programmers have already used in their code.

exercises

1. Create a class with **public**, **private**, and **protected** data members and method members. Create an object of this class and see what kind of compiler messages you get when you try to access all the class members. Be aware that classes in the same directory are part of the "default" package.

2. Create a class with **protected** data. Create a second class in the same file with a method that manipulates the **protected** data in the first class.

6: reusing classes

One of the most compelling features about Java is code reuse. But to be revolutionary, you've got to be able to do a lot more than copy code and change it.

That's the approach used in procedural languages like C, and it hasn't worked very well. As with everything in Java, the solution revolves around the class. You reuse code by creating new classes, but instead of creating them from scratch, you use existing classes that someone has already built and debugged.

The trick is to use the classes without soiling the existing code. In this chapter you'll see two ways to accomplish this. The first is quite straightforward: You simply create objects of your existing class inside the new class. This is called *composition* because the new class is composed of objects of existing classes. Here, you're simply reusing the functionality of the code, not its form.

The second approach is subtler. It creates a new class as a *type of* an existing class. You literally take the form of the existing class and add code to it, without modifying the existing class. This magical act is called *inheritance*, and the compiler does most of the work. Inheritance is one of the cornerstones of object-oriented programming and has additional implications that will be explored in the next chapter.

It turns out that much of the syntax and behavior are similar for both composition and inheritance (which makes sense; they are both ways of making new types from existing types). In this chapter, you'll learn about these code reuse mechanisms.

composition syntax

Up till now composition has been used quite frequently. You simply place object handles inside new classes. For example, suppose you'd like an object that holds several **String** objects, a couple of primitives and an object of another class. For the class objects, just put handles inside your new class, and for the primitives just define them inside your class:

```
//: SprinklerSystem.java
// Composition for code reuse

class WaterSource {
  private String s;
  WaterSource() {
    System.out.println("WaterSource()");
    s = new String("Constructed");
  }
  public String toString() { return s; }
}

public class SprinklerSystem {
  private String valve1, valve2, valve3, valve4;
  WaterSource source;
  int i;
  float f;
  void print() {
    System.out.println("valve1 = " + valve1);
    System.out.println("valve2 = " + valve2);
    System.out.println("valve3 = " + valve3);
    System.out.println("valve4 = " + valve4);
    System.out.println("i = " + i);
    System.out.println("f = " + f);
    System.out.println("source = " + source);
  }
  public static void main(String args[]) {
    SprinklerSystem x = new SprinklerSystem();
    x.print();
  }
} ///:~
```

One of the methods defined in **WaterSource** is special: **toString( )**. You will learn later that every non-primitive object has a **toString( )** method, and it's called in special situations when the compiler wants a string but its got one of these objects. So in the expression:

```
System.out.println("source = " + source);
```

The compiler sees you trying to add a **String** object (**"source = "**) to a **WaterSource**. This doesn't make sense to it, because you can only "add" a **String** to another **String**, so it says "I'll turn **source** into a **String** by calling **toString( )**!" After doing this it can combine the two **String**s and pass the resulting **String** to **System.out.println( )**. Any time you want to allow this behavior with a class you create you only need to write a **toString( )** method.

At first glance, you might assume – Java being as safe and careful as it is – that the compiler would automatically construct objects for each of the handles in the above code, for example calling the default constructor for **WaterSource** to initialize **source**. The output of the print statement is in fact:

```
valve1 = null
valve2 = null
valve3 = null
valve4 = null
i = 0
```

```
f = 0.0
source = null
```

Primitives that are fields in a class are automatically initialized to zero, as noted in Chapter 2. But the object handles are initialized to **null**, and if you try to call methods for any of them you'll get an exception. It's actually pretty good (and useful) that you can still print them out without throwing an exception.

It makes sense that the compiler doesn't just create a default object for every handle because that would incur unnecessary overhead in many cases. If you want the handles initialized, you can do it:

1. At the point the objects are defined. This means they'll always be initialized before the constructor is called.
2. In the constructor for that class
3. Right before you actually need to use the object. This may reduce overhead, if there are situations where the object doesn't need to be created.

All three approaches are shown here:

```
//: Bath.java
// Constructor initialization with composition

class Soap {
  private String s;
  Soap() {
    System.out.println("Soap()");
    s = new String("Constructed");
  }
  public String toString() { return s; }
}

public class Bath {
  private String
    // Initializing at point of definition:
    s1 = new String("Happy"),
    s2 = "Happy",
    s3, s4;
  Soap castille;
  int i;
  float toy;
  Bath() {
    System.out.println("Inside Bath()");
    s3 = new String("Joy");
    i = 47;
    toy = 3.14f;
    castille = new Soap();
  }
  void print() {
    // Delayed initialization:
    if(s4 == null)
      s4 = new String("Joy");
    System.out.println("s1 = " + s1);
    System.out.println("s2 = " + s2);
    System.out.println("s3 = " + s3);
    System.out.println("s4 = " + s4);
    System.out.println("i = " + i);
    System.out.println("toy = " + toy);
    System.out.println("castille = " + castille);
  }
```

```
  public static void main(String args[]) {
    Bath b = new Bath();
    b.print();
  }
} ///:~
```

Notice that in the **Bath** constructor a statement is executed before any of the initializations take place. When you don't initialize at the point of definition, there's still no guarantee that you'll perform any initialization before you send a message to an object handle – except for the inevitable run-time exception.

inheritance syntax

Inheritance is such an integral part of Java (and OOP languages in general) that it was introduced in Chapter 1 and has been used occasionally in chapters before this one, since certain situations required it. In addition, you're always doing inheritance when you create a class, because if you don't say otherwise you inherit from Java's standard root class **Object**.

The syntax for composition is obvious, but to perform inheritance there's a distinctly different form. When you inherit, you are saying, "This new class is like that old class." You state this in code by giving the name of the class, as usual, but before the opening brace of the class body, you put the keyword **extends** followed by the name of the *base class*. When you do this, you automatically get all the data members and methods in the base class. Here's an example:

```
//: Detergent.java
// Inheritance syntax & properties

class Cleanser {
  private String s = new String("Cleanser");
  public void append(String a) { s += a; }
  public void dilute() { append(" dilute()"); }
  public void apply() { append(" apply()"); }
  public void scrub() { append(" scrub()"); }
  public void print() { System.out.println(s); }
  public static void main(String args[]) {
    Cleanser x = new Cleanser();
    x.dilute(); x.apply(); x.scrub();
    x.print();
  }
}

public class Detergent extends Cleanser {
  // Change a method:
  public void scrub() {
    append(" Detergent.scrub()");
    super.scrub(); // Call base-class version
  }
  // Add methods to the interface:
  public void foam() { append(" foam()"); }
  // Test the new class:
  public static void main(String args[]) {
    Detergent x = new Detergent();
    x.dilute();
    x.apply();
    x.scrub();
    x.foam();
    x.print();
    System.out.println("Testing base class:");
    Cleanser.main(args);
```

```
  }
} ///:~
```

This demonstrates a number of features. First, in the **Cleanser append( )** method, strings are concatenated to **s** using the **+=** operator, which is one of the operators (along with '**+**') that the Java designers "overloaded" to work with **String**s.

Second, both **Cleanser** and **Detergent** contain a **main( )** method. You can create a **main( )** for each one of your classes, and it's often recommended to code this way so that your test code is wrapped in with the class. Even if you have lots of classes in a program, only the **main( )** for the **public** class invoked on the command line will be called (and you can only have one **public** class per file). So in this case, when you say **java Detergent**, **Detergent.main( )** will be called. But you can also say **java Cleanser** to invoke **Cleanser.main( )**, even though **Cleanser** is not a **public** class. This technique of putting a **main( )** in each class allows easy unit testing for each class. In addition, you don't have to remove the **main( )** when you're finished testing; you can leave it in for later testing.

Here, you can see that **Detergent.main( )** calls **Cleanser.main( )** explicitly.

It's important that all the methods in **Cleanser** are **public**. Remember that if you leave off any access specifier the member defaults to "friendly," which only allows access to package members. Thus, within this package, anyone could use those methods if there were no access specifier. **Detergent** would have no trouble, for example. However if some other class were to inherit **Cleanser** it can only access **public** members. So to plan for inheritance, as a general rule make all fields **private** and all methods **public** (**protected** members also allow access by derived classes; you'll learn about this later). Of course, in particular cases you'll need to make adjustments, but this is a useful guideline.

Note that **Cleanser** has a set of methods in its interface: **append( )**, **dilute( )**, **apply( )**, **scrub( )** and **print( )**. Because **Detergent** is *derived from* **Cleanser** (via the **extends** keyword) it automatically gets all these methods in its interface, even though you don't see them all explicitly defined in **Detergent**. You can think of inheritance, then, as *reusing the interface*.

As seen in **scrub( )**, it's possible to take a method that's been defined in the base class and modify it. In this case, you may want to call the method from the base class inside the new version. But inside **scrub( )** you cannot simply call **scrub( )**, since that would produce a recursive call which isn't what you want. To solve this problem Java has a keyword **super** which refers to the "superclass" that the current class has been inherited from. Thus the expression **super.scrub( )** calls the base-class version of the method **scrub( )**.

When inheriting you're not restricted to the interface of the base class. You can also add new methods to the interface, exactly the way you put any method in a class: just define it. The **extends** keyword actually suggests that you are going to add new methods to the base-class interface, and the method **foam( )** is an example of this.

In **Detergent.main( )** you can see that for a **Detergent** object you can call all the methods that are available in **Cleanser** as well as in **foam( )**.

initializing the base class

When you inherit from a class, you get more than just the interface – you get a complete subobject of the base class inside the derived class. Of course, it's essential that this object be initialized correctly and there's only one way to guarantee it: perform the initialization in the constructor. Java automatically inserts calls to the base-class constructor in the derived-class constructor. The following example shows this working with three levels of inheritance:

```
//: Cartoon.java
// Constructor calls during inheritance

class Art {
  Art() {
    System.out.println("Art constructor");
  }
}
```

```
class Drawing extends Art {
  Drawing() {
    System.out.println("Drawing constructor");
  }
}

public class Cartoon extends Drawing {
  Cartoon() {
    System.out.println("Cartoon constructor");
  }
  public static void main(String args[]) {
    Cartoon x = new Cartoon();
  }
} ///:~
```

The output for this program shows the automatic calls:

```
Art constructor
Drawing constructor
Cartoon constructor
```

You can see that the construction happens from the base "outward," so the base class is initialized before the derived-class constructors can access it.

Even if you don't create a constructor for **Cartoon( )**, the compiler will synthesize a default constructor for you that calls the base class constructor.

constructors with arguments

The above example has default constructors – that is, they don't have any arguments. It's easy for the compiler to call these because there's no question about what arguments to pass. But what if your class doesn't have default arguments or you want to call a base-class constructor that has an argument? You must explicitly write the calls to the base-class constructor using the **super** keyword and the appropriate argument list:

```
//: Chess.java
// Inheritance, constructors and arguments

class Game {
  Game(int i) {
    System.out.println("Game constructor");
  }
}

class BoardGame extends Game {
  BoardGame(int i) {
    super(i);
    System.out.println("BoardGame constructor");
  }
}

public class Chess extends BoardGame {
  Chess() {
    super(11);
    System.out.println("Chess constructor");
  }
  public static void main(String args[]) {
    Chess x = new Chess();
  }
} ///:~
```

If you don't call the base-class constructor in **BoardGame( )**, the compiler will complain that it can't find a constructor of the form **Game( )**. In addition, the call to the base-class constructor *must* be the first thing you do in the derived-class constructor (the compiler will remind you if you get it wrong).

catching base constructor exceptions

As just noted, the compiler forces you to place the base-class constructor call first in the body of the derived-class constructor. This means nothing else can appear before it. As you'll see in Chapter 9, this also prevents a derived-class constructor from catching any exceptions that come from a base class. This can be inconvenient at times.

combining composition & inheritance

Of course, you can use the two together. The following example shows the creation of a more complex class, using both inheritance and composition, along with the necessary constructor initialization:

```
//: PlaceSetting.java
// Combining composition & inheritance

class Plate {
  Plate(int i) {
    System.out.println("Plate constructor");
  }
}

class DinnerPlate extends Plate {
  DinnerPlate(int i) {
    super(i);
    System.out.println(
      "DinnerPlate constructor");
  }
}

class Utensil {
  Utensil(int i) {
    System.out.println("Utensil constructor");
  }
}

class Spoon extends Utensil {
  Spoon(int i) {
    super(i);
    System.out.println("Spoon constructor");
  }
}

class Fork extends Utensil {
  Fork(int i) {
    super(i);
    System.out.println("Fork constructor");
  }
}

class Knife extends Utensil {
  Knife(int i) {
```

```
    super(i);
    System.out.println("Knife constructor");
  }
}

// A cultural way of doing something:
class Custom {
  Custom(int i) {
    System.out.println("Custom constructor");
  }
}

public class PlaceSetting extends Custom {
  Spoon sp;
  Fork frk;
  Knife kn;
  DinnerPlate pl;
  PlaceSetting(int i) {
    super(i + 1);
    sp = new Spoon(i + 2);
    frk = new Fork(i + 3);
    kn = new Knife(i + 4);
    pl = new DinnerPlate(i + 5);
    System.out.println(
      "PlaceSetting constructor");
  }
  public static void main(String args[]) {
    PlaceSetting x = new PlaceSetting(9);
  }
} ///:~
```

While the compiler forces you to initialize the base classes, and requires that you do it right at the beginning of the constructor, it doesn't watch over you to make sure you initialize the member objects, so you must remember to pay attention to that.

guaranteeing proper cleanup

Java doesn't have the C++ concept of a *destructor*, a method that is automatically called when an object is destroyed. The reason is probably that in Java the practice is simply to forget about objects rather than destroying them, allowing the garbage collector to reclaim the memory as necessary.

Often this is fine, but there are times when your class may perform some activities during its lifetime that require cleanup. As mentioned in Chapter 4, you can't rely on when the garbage collector will be called, or if it will ever be called. Thus, if you want something cleaned up for a class, you must write a special method to do it explicitly, and make sure that the client programmer knows they must call this method. On top of this, as described in Chapter 9 (exception handling), you must guard against an exception by putting such cleanup in a **finally** clause.

Consider an example of a computer-aided-design system that draws pictures on the screen:

```
//: CADSystem.java
// Ensuring proper cleanup
import java.util.*;

class Shape {
  Shape(int i) {
    System.out.println("Shape constructor");
  }
  void cleanup() {
    System.out.println("Shape cleanup");
  }
```

```
}

class Circle extends Shape {
  Circle(int i) {
    super(i);
    System.out.println("Drawing a Circle");
  }
  void cleanup() {
    System.out.println("Erasing a Circle");
    super.cleanup();
  }
}

class Triangle extends Shape {
  Triangle(int i) {
    super(i);
    System.out.println("Drawing a Triangle");
  }
  void cleanup() {
    System.out.println("Erasing a Triangle");
    super.cleanup();
  }
}

class Line extends Shape {
  private int start, end;
  Line(int start, int end) {
    super(start);
    this.start = start;
    this.end = end;
    System.out.println("Drawing a Line: " +
           start + ", " + end);
  }
  void cleanup() {
    System.out.println("Erasing a Line: " +
           start + ", " + end);
    super.cleanup();
  }
}

public class CADSystem extends Shape {
  private Circle c;
  private Triangle t;
  private Line[] lines = new Line[10];
  CADSystem(int i) {
    super(i + 1);
    for(int j = 0; j < 10; j++)
      lines[j] = new Line(j, j*j);
    c = new Circle(1);
    t = new Triangle(1);
    System.out.println("Combined constructor");
  }
  void cleanup() {
    System.out.println("CADSystem.cleanup()");
    t.cleanup();
    c.cleanup();
    for(int i = 0; i < lines.length; i++)
      lines[i].cleanup();
    super.cleanup();
  }
```

```
  public static void main(String args[]) {
    CADSystem x = new CADSystem(47);
    try {
      // Code and exception handling...
    } finally {
      x.cleanup();
    }
  }
} ///:~
```

Everything in this system is some kind of **Shape** (which is itself a kind of **Object** since it's implicitly inherited from the root class). Each class redefines **Shape**'s **cleanup( )** method in addition to calling the base-class version of that method using **super**. The specific **Shape** classes **Circle**, **Triangle** and **Line** all have constructors that "draw," although any method called during the lifetime of the object may be responsible for doing something that needs cleanup. Each class has its own **cleanup( )** method to restore non-memory things back to the way they were before the object existed.

In **main( )**, you can see two keywords that are new, and won't officially be introduced until Chapter 9: **try** and **finally**. The **try** keyword indicates that the block that follows (delimited by curly braces) is a *guarded region*, which means that it is given special treatment. One of these special treatments is that the code in the **finally** clause following this guarded region is *always* executed, no matter how the **try** block exits (with exception handling, it's possible to leave a **try** block in a number of non-ordinary ways). Here, the **finally** clause is saying "always call **cleanup( )** for **x**, no matter what happens." These keywords will be explained thoroughly in Chapter 9.

Note that in your cleanup method you must also pay attention to the order in which the base-class and member-object cleanup methods get called, in case one subobject may depend on another. In general you should follow the form imposed by the C++ compiler on its destructors: first perform all the work specific to your class (which may require that base-class elements still be viable) then call the base-class cleanup method, as demonstrated here.

There may be many cases where the cleanup issue is not a problem; you just let the garbage collector do the work. But when you must do it explicitly, diligence and attention is required.

order of garbage collection

There's not much you can rely on when it comes to garbage collection. The garbage collector may never be called. If it is, it can reclaim objects in any order it wants. In addition, implementations of the garbage collector in Java 1.0 often don't call the **finalize( )** methods. It's best not to rely on garbage collection for anything but memory reclamation, and if you want cleanup to take place, make your own cleanup methods and don't rely on **finalize( )** (as mentioned earlier, Java 1.1 can be forced to call all the finalizers).

name hiding

Only C++ programmers may be surprised by this, since it works differently in that language. If a Java base class has a method name that's overloaded several times, redefining that method name in the derived class will *not* hide any of the base-class versions. Thus overloading works regardless of whether the method was defined at this level or in a base class:

```
//: Hide.java
// Overloading a base-class method name
// in a derived class does not hide the
// base-class versions

class Homer {
  char doh(char c) {
    System.out.println("doh(char)");
    return 'd';
  }
  float doh(float f) {
    System.out.println("doh(float)");
```

```
    return 1.0f;
  }
}

class Milhouse {}

class Bart extends Homer {
  void doh(Milhouse m) {}
}

class Hide {
  public static void main(String args[]) {
    Bart b = new Bart();
    b.doh(1); // doh(float) used
    b.doh('x');
    b.doh(1.0f);
    b.doh(new Milhouse());
  }
} ///:~
```

As you'll see in the next chapter, it's far more common to redefine methods of the same name using exactly the same signature and return type as in the base class. It can be confusing otherwise (which is why C++ disallows it, to prevent you from making what is probably a mistake).

choosing composition vs. inheritance

Both composition and inheritance allow you to place subobjects inside your new class. You may now be wondering what the difference is between the two, and when to choose one over the other.

Composition is generally used when you want the features of an existing class inside your new class, but not its interface. That is, you embed an object that you're planning on using to implement features of your new class, but the user of your new class sees the interface you've defined rather than the interface from the original class. For this effect, you embed **private** objects of existing classes inside your new class.

Sometimes it makes sense to allow the class user to directly access the composition of your new class, that is, to make the member objects **public**. The member objects use implementation hiding themselves, so this is a safe thing to do and when the user knows you're assembling a bunch of parts, it makes the interface easier to understand. A **car** object is a good example:

```
//: Car.java
// Composition with public objects

class Engine {
  public void start() {}
  public void rev() {}
  public void stop() {}
}

class Wheel {
  public void inflate(int psi) {}
}

class Window {
  public void rollup() {}
  public void rolldown() {}
```

```
}

class Door {
  public Window window = new Window();
  public void open() {}
  public void close() {}
}

public class Car {
  public Engine engine = new Engine();
  public Wheel wheel[] = new Wheel[4];
  public Door left = new Door(),
       right = new Door(); // 2-door
  Car() {
    for(int i = 0; i < 4; i++)
      wheel[i] = new Wheel();
  }
  public static void main(String args[]) {
    Car car = new Car();
    car.left.window.rollup();
    car.wheel[0].inflate(72);
  }
} ///:~
```

Because the composition of a car is part of the analysis of the problem (and not simply part of the underlying design), making the members public assists the client programmer's understanding of how to use the class and requires less code complexity for the creator of the class.

When you inherit, you take an existing class and make a special version of it. Generally, this means you're taking a general-purpose class and specializing it for a particular need. With a little thought, you'll see that it would make no sense to compose a car using a vehicle object — a car doesn't contain a vehicle, it *is* a vehicle. The *is-a* relationship is expressed with inheritance, and the *has-a* relationship is expressed with composition.

protected

Now that you've been introduced to inheritance, the keyword **protected** finally has meaning. In an ideal world, **private** members would always be hard-and-fast **private**, but in real projects there are times when you want to make something hidden from the world at large and yet allow access for members of derived classes. The **protected** keyword is a nod to pragmatism; it says, "This is **private** as far as the class user is concerned, but available to anyone who inherits from this class or anyone else in the same **package**." That is, **protected** in Java is automatically "friendly."

The best tack to take is to leave the data members **private** — you should always preserve your right to change the underlying implementation. You can then allow controlled access to inheritors of your class through **protected** methods:

```
//: Orc.java
// The protected keyword
import java.util.*;

class Villain {
  private int i;
  protected int read() { return i; }
  protected void set(int ii) { i = ii; }
  public Villain(int ii) { i = ii; }
  public int value(int m) { return m*i; }
}
```

```
public class Orc extends Villain {
  private int j;
  public Orc(int jj) { super(jj); j = jj; }
  public void change(int x) { set(x); }
} ///:~
```

You can see that **change( )** has access to **set( )** because it's **protected**.

incremental development

One of the advantages of inheritance is that it supports *incremental development* by allowing you to introduce new code without causing bugs in existing code. This also isolates new bugs to the new code. By inheriting from an existing, functional class and adding data members and methods (and redefining existing methods) you leave the existing code — that someone else may still be using — untouched and unbugged. If a bug happens, you know it's in your new code, which is much shorter and easier to read than if you had modified the body of existing code.

It's rather amazing how cleanly the classes are separated. You don't even need the source code for the methods in order to reuse the code. At most, you just import a package. (This is true for both inheritance and composition.)

It's important to realize that program development is an incremental process, just like human learning. You can do as much analysis as you want, but you still won't know all the answers when you set out on a project. You'll have much more success — and more immediate feedback — if you start out to "grow" your project as an organic, evolutionary creature, rather than constructing it all at once like a glass-box skyscraper.

Although inheritance for experimentation is a useful technique, at some point after things stabilize you need to take a new look at your class hierarchy with an eye to collapsing it into a sensible structure. Remember that underneath it all, inheritance is meant to express a relationship that says, "This new class is a *type of* that old class." Your program should not be concerned with pushing bits around, but instead with creating and manipulating objects of various types to express a model in the terms given you by the problem's space.

upcasting

The most important aspect of inheritance is not that it provides methods for the new class. It's the relationship expressed between the new class and the base class. This relationship can be summarized by saying, "The new class *is a type of* the existing class."

This description is not just a fanciful way of explaining inheritance — it's supported directly by the language. As an example, consider a base class called **Instrument** that represents musical instruments and a derived class called **Wind**. Because inheritance means that all the methods in the base class are also available in the derived class, any message you can send to the base class can also be sent to the derived class. So if the **Instrument** class has a **play( )** method, so will **Wind** instruments. This means we can accurately say that a **Wind** object is also a type of **Instrument**. The following example shows how the compiler supports this notion:

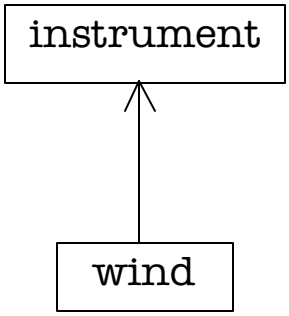
```
//: Wind.java
// Inheritance & upcasting
import java.util.*;

class Instrument {
  public void play() {}
  static void tune(Instrument i) {
    // ...
    i.play();
  }
```

```
}

// Wind objects are instruments
// because they have the same interface:
class Wind extends Instrument {
  public static void main(String args[]) {
    Wind flute = new Wind();
    Instrument.tune(flute); // Upcasting
  }
} ///:~
```

What's interesting in this example is the **tune( )** method, which accepts an **Instrument** reference. However, in **Wind**.**main( )** the **tune( )** method is called by giving it a **Wind** object. Given that Java is very particular about type checking, it seems strange that a method that accepts one type will readily accept another type, until you realize that a **Wind** object is also an **Instrument** object, and there's no method that **tune( )** could call for an **Instrument** that isn't also in **Wind**. Inside **tune( )**, the code works for **Instrument** and anything derived from **Instrument**, and the act of converting a **Wind** object, reference, or pointer into an **Instrument** object, reference, or pointer is called *upcasting*.

why "upcasting"?

The reason for the term is historical and is based on the way class inheritance diagrams have traditionally been drawn: with the root at the top of the page, growing downward. (Of course, you can draw your diagrams any way you find helpful.) The inheritance diagram for **Wind.java** is then:



Casting from derived to base moves *up* on the inheritance diagram, so it's commonly referred to as upcasting. Upcasting is always safe because you're going from a more specific type to a more general type — the only thing that can occur to the class interface is that it lose methods, not gain them. This is why the compiler allows upcasting without any explicit casts or other special notation.

You can also perform the reverse of upcasting, called *downcasting*, but this involves a dilemma that is the subject of Chapter 11.

composition vs. inheritance revisited

In object-oriented programming, the most likely way that you'll create and use code is by simply packaging data and methods together into a class, and using objects of that class. Less often, you'll use existing classes to build new classes with composition. Even less often than that you'll use inheritance. Thus, although inheritance gets a lot of emphasis while learning OOP it doesn't mean you should use it everywhere you possibly can; on the contrary you should use it sparingly, only when it's clear that inheritance is useful. One of the clearest ways to determine whether you should be using composition or inheritance is by asking whether you'll ever need to upcast from your new class to the base class. If you must upcast, then inheritance is necessary, but if you don't need to upcast, then you should look closely at whether you need inheritance. The next chapter (polymorphism) provides one of the most compelling reasons for upcasting, but if you remember to ask why you would want to upcast you'll have a good tool for deciding between composition and inheritance.

the final keyword

The **final** keyword has slightly different meanings depending on the context in which it is used, but in general it says "this cannot be changed." You may want to prevent changes for two reasons: design or efficiency. Because these two reasons are quite different, it's possible to misuse the **final** keyword.

The following sections discuss the three places where **final** can be used: for data, methods and for a class itself.

final data

Many programming languages have a way to tell the compiler that a piece of data is "constant." A constant is useful for two reasons:

1. It may be a *compile-time constant* that cannot change

2. It may be a value initialized at run-time that you don't want changed

In the case of a compile-time constant the compiler may "fold" the constant value into any calculations where it's used; that is, the calculation may be performed at compile time, thus eliminating some run-time overhead. In Java, these sorts of constants must be primitives and are expressed using the **final** keyword. A value must be given at the time of definition of such a constant.

A field that is both **static** and **final** has only one piece of storage that cannot be changed.

When using **final** with object handles rather than primitives, the meaning gets a bit confusing. With a primitive, **final** makes the *value* a constant, but with an object handle, **final** makes the handle itself a constant. The handle must be initialized to an object at the point of declaration, and the handle can never be changed to point to another object. However, the object itself may be modified; Java does not provide a way to make any arbitrary object a constant (you can, however, write your class so that objects have the effect of being constant). This restriction includes arrays, which are also objects.

Here's an example that demonstrates **final** fields:

```
//: FinalData.java
// The effect of final on fields

class Value {
  int i = 1;
}

public class FinalData {
  // Can be compile-time constants
  final int i1 = 9;
  static final int i2 = 99;
  // Typical public constant:
  public static final int i3 = 39;
  // Cannot be compile-time constants:
  final int i4 = (int)(Math.random()*20);
  static final int i5 = (int)(Math.random()*20);

  Value v1 = new Value();
  final Value v2 = new Value();
  static final Value v3 = new Value();
  //! final Value v4; // Error: no initializer

  // Arrays:
  final int a[] = { 1, 2, 3, 4, 5, 6 };

  public void print(String id) {
```

```
    System.out.println(
      id + ": " + "i4 = " + i4 +
      ", i5 = " + i5);
  }
  public static void main(String[] args) {
    FinalData fd1 = new FinalData();
    //! fd1.i1++; // Error: can't change value
    fd1.v2.i++; // Object isn't constant!
    fd1.v1 = new Value(); // OK -- not final
    for(int i = 0; i < fd1.a.length; i++)
      fd1.a[i]++; // Object isn't constant!
    //! fd1.v2 = new Value(); // Error: Can't
    //! fd1.v3 = new Value(); // change handle
    //! fd1.a = new int[3];

    fd1.print("fd1");
    System.out.println("Creating new FinalData");
    FinalData fd2 = new FinalData();
    fd1.print("fd1");
    fd2.print("fd2");
  }
} ///:~
```

Since **i1** and **i2** are **final** primitives with compile-time values, they can both be used as compile-time constants and are not different in any important way. However, **i3** is the more typical way you'll see such constants defined: **public** so they're usable outside the package, **static** to emphasize that there's only one, and **final** to say it's a constant.

Just because something is **final** doesn't mean its value is known at compile-time. This is demonstrated by initializing **i4** and **i5** at run-time using randomly generated numbers. This portion of the example also shows the difference between making a **final** value **static** or non-**static**. This difference only shows up when the values are initialized at run-time, since the compile-time values are treated the same by the compiler (and presumably optimized out of existence). The difference is shown in the output from one run:

```
fd1: i4 = 15, i5 = 9
Creating new FinalData
fd1: i4 = 15, i5 = 9
fd2: i4 = 10, i5 = 9
```

Note that the values of **i4** for **fd1** and **fd2** are unique, but the value for **i5** is not changed by creating the second **FinalData** object. That's because it's **static** and is initialized once upon loading and not each time a new object is created.

The variables **v1** through **v4** demonstrate the meaning of a **final** handle. As you can see in **main( )**, just because **v2** is **final** doesn't mean you can't change its value. However, you cannot re-bind **v2** to a new object, precisely because it's **final**. That's what **final** means for a handle. You can also see the same meaning holds true for an array, which is just another kind of handle. Making handles **final** seems much less useful than making primitives **final**.

blank finals

Java 1.1 allows the creation of *blank finals*, which are fields that are declared as **final** but which are not given an initialization value. In all cases, the blank final *must* be initialized before it is used, and the compiler ensures this. However, blank finals provide much more flexibility in the use of the **final** keyword since, for example, a **final** field inside a class can now be different for each object and yet still retains its immutable quality. Here's an example:

```
//: BlankFinal.java
// "Blank" final data members

class Poppet { }
```

```
class BlankFinal {
  final int i = 0; // Initialized final
  final int j; // Blank final
  final Poppet p; // Blank final handle
  // Blank finals MUST be initialized
  // in the constructor:
  BlankFinal() {
    j = 1; // Initialize blank final
    p = new Poppet();
  }
  BlankFinal(int x) {
    j = x; // Initialize blank final
    p = new Poppet();
  }
  public static void main(String args[]) {
    BlankFinal bf = new BlankFinal();
  }
} ///:~
```

You're forced to perform assignments to finals either with an expression at the point of definition of the field, or in every constructor. This way it's guaranteed that the final field is always initialized before use.

final arguments

Java 1.1 allows you to make arguments **final** by declaring them as such in the argument list. All this means is that inside the method you cannot change what the argument handle points to:

```
//: FinalArguments.java
// Using "final" with method arguments

class Gizmo {
  public void spin() {}
}

public class FinalArguments {
  void with(final Gizmo g) {
    //! g = new Gizmo(); // Illegal -- g is final
    g.spin();
  }
  void without(Gizmo g) {
    g = new Gizmo(); // OK -- g not final
    g.spin();
  }
  public static void main(String args[]) {
    FinalArguments bf = new FinalArguments();
    bf.without(null);
    bf.with(null);
  }
} ///:~
```

Note that you can still assign a **null** handle to an argument that's final without the compiler catching it, just like you can with a non-final argument.

final methods

There are two reasons for **final** methods. The first is simply to put a "lock" on the method to prevent any inheriting class from changing its meaning. This is done for design reasons, when you want to make sure that the behavior of a method is retained.

The second and more predominant reason for **final** is efficiency. If you make a method **final**, you are allowing the compiler to turn any calls to that method into *inline* calls. That is, when the compiler sees a **final** method call it may (at its discretion) skip the normal approach of inserting code to perform the method call mechanism (push arguments on the stack, hop over to the method code and execute it, hop back and clean off the stack arguments, deal with the return value) and instead replace the method call with a copy of the method code. This eliminates the overhead of the method call. Of course, if a method is big then your code begins to bloat, and if the method is big you probably won't see any performance gains from inlining since any improvements will be dwarfed by the amount of time spent inside the method. It is implied that the Java compiler is able to detect these situations and choose wisely whether to actually inline a **final** method. However, it's better not to trust that the compile is able to do this and only make a method **final** if it's quite small or if you want to explicitly prevent overriding.

Any **private** methods in a class are implicitly **final**. Because you can't access a **private** method, you can't override it. You can add the **final** specifier to a **private** method but it doesn't give that method any extra meaning.

final classes

When you say that an entire class is **final** (by preceding its definition with the **final** keyword) you state that you don't want to inherit from this class or allow anyone else to do so. For some reason the design of your class is such that there is never a need to make any changes. Alternatively you may be dealing with an efficiency issue and you want to make sure that any activity involved with objects of this class is as efficient as possible.

```
//: Jurassic.java
// Making an entire class final

class SmallBrain {}

final class Dinosaur {
  int i = 7;
  int j = 1;
  SmallBrain x = new SmallBrain();
  void f() {}
}

//! class Further extends Dinosaur {}
// error: Cannot extend final class 'Dinosaur'

public class Jurassic {
  public static void main(String args[]) {
    Dinosaur n = new Dinosaur();
    n.f();
    n.i = 40;
    n.j++;
  }
} ///:~
```

Note that the data members can be **final** or not, as you choose. The same rules apply to **final** for data members regardless of whether the class is defined as **final**. Defining the class as **final** simply prevents inheritance, no more. However, because it prevents inheritance this means that all methods in a **final** class are implicitly **final**, since there's no way to override them. Thus the compiler has the same efficiency options as it does if you explicitly declare a method **final**.

You can add the **final** specifier to a method in a **final** class, but it doesn't add any meaning.

final caution

It can seem to be very sensible to make a method **final** while you're designing a class. You may feel that efficiency is very important when using your class and that no one could possibly want to override your methods anyway. Sometimes this is true.

But be very careful with your assumptions here. In general, it's very hard to anticipate how a class can be reused, especially a general-purpose class. If you define a method as **final** you may prevent the possibility of reusing your class through inheritance in some other programmer's project simply because you couldn't imagine it being used that way.

The standard Java library is a good example of this. In particular, the **Vector** class is commonly used and might be even more useful if, in the name of efficiency, all the methods hadn't been made **final**. It's easily conceivable that you might want to inherit and override with such a fundamentally useful class, but the designers somehow decided this wasn't appropriate. This is ironic in two places. First, **Stack** is inherited from **Vector.** The **Vector** class itself was not made **final** so this is possible, but you can only add members to the interface, as **Stack** does. Second, many of the most important methods of **Vector**, such as **addElement( )** and **elementAt( )** are **synchronized**, which as you shall see in Chapter 14 incurs a significant performance overhead that probably wipes out any gains provided by **final**. This lends credence to the theory that programmers are consistently bad at guessing where optimizations should occur. It's just too bad that such a clumsy design made it into the standard library, where we all have to cope with it.

It's also interesting to note that **Hashtable**, another important standard library class, does *not* have any **final** methods. As mentioned elsewhere in this book, it's quite obvious that some classes were designed by completely different people than others (note the brevity of the method names in **Hashtable** compared to **Vector**). This is precisely the sort of thing that should *not* be obvious to consumers of a class library – when things are inconsistent it just makes more work for the user. Yet another paean to the value of design and code walkthroughs.

initialization & class loading

In many more traditional languages, programs are loaded all at once as part of the startup process. This is followed by initialization, and then the program begins. The process of initialization in these languages must be carefully controlled so that the order of initialization of **static**s doesn't cause trouble. C++, for example, has problems if one **static** is expecting another **static** to be valid before the second one has been initialized.

Java doesn't have this problem because it takes a different approach to loading. Because everything in Java is an object many activities become easier, and this is one of them. As you will learn in the next chapter, the code for each object exists in a separate file. That file isn't loaded until the code is needed. In general, you can say that until an object of that class is constructed, the class code doesn't get loaded. Since there can be some subtleties with **static** methods, you can also say "class code is loaded at the point of first use."

The point of first use is also where the **static** initialization takes place: all the **static** objects and the **static** code block will be initialized in textual order (that is, the order that you write them down in the class definition) at the point of loading. The **static**s, of course, are only initialized once.

initialization with inheritance

It's helpful to look at the whole initialization process including inheritance to get a full picture of what happens. Consider the following code:

```
//: Beetle.java
// The full process of initialization.

class Insect {
  int i = 9;
```

```
  int j;
  Insect() {
    prt("i = " + i + ", j = " + j);
    j = 39;
  }
  static int x =
    prt("static Insect.x initialized");
  static int prt(String s) {
    System.out.println(s);
    return 47;
  }
}

public class Beetle extends Insect {
  int k = prt("Beetle.k " + "initialized");
  Beetle() {
    prt("k = " + k);
    prt("j = " + j);
  }
  static int x =
    prt("static Beetle.x " + "initialized");
  static int prt(String s) {
    System.out.println(s);
    return 63;
  }
  public static void main(String args[]) {
    prt("Beetle constructor");
    Beetle x = new Beetle();
  }
} ///:~
```

The output for this program is:

```
static Insect.x initialized
static Beetle.x initialized
Beetle constructor
i = 9, j = 0
Beetle.k initialized
k = 63
j = 39
```

The first thing that happens when you run Java on **Beetle** is that the loader goes out and finds that class. In the process of loading it, the loader notices that it has a base class (that's what the **extends** keyword says), which it then loads. This will happen whether or not you're actually going to make an object of that base class (try commenting out the object creation to prove it to yourself).

If the base class itself has a base class, that second base class would then be loaded, and so on. Next, the **static** initialization in the root base class (in this case, **Insect**) is performed, then the next derived class and so on. This is important because the derived-class static initialization may depend on the base class member being initialized properly.

At this point, the necessary classes have all been loaded so the object can be created. First, all the primitives in this object are set to their default values and the object handles are set to **null**. Then the base-class constructor will be called; in this case the call is automatic but you may also specify the constructor call (as the first operation in the **Beetle( )** constructor) using **super**. The base class construction goes through the same process in the same order as the derived-class constructor. After the base-class constructor completes, the instance variables are initialized, in textual order. Finally, the rest of the body of the constructor is executed.

summary

Both inheritance and composition allow you to create a new type from existing types. Typically, however, you use composition to reuse existing types as part of the underlying implementation of the new type and inheritance when you want to reuse the interface. Since the derived class has the base-class interface, it can be *upcast* to the base, which is critical for polymorphism as you'll see in the next chapter.

Despite the strong emphasis on inheritance in object-oriented programming, when starting a design, you should generally prefer composition during the first cut, and only use inheritance when it is clearly necessary (as you'll see in the next chapter). Composition tends to be more generally flexible. In addition, by using the added artifice of inheritance with your member type, you can change the exact type, and thus the behavior, of those member objects at run-time, therefore you can change the behavior of the composed object at run-time.

Although code reuse through composition and inheritance is very helpful for rapid project development, you'll generally want to redesign your class hierarchy before allowing other programmers to become dependent on it. Your goal is a hierarchy where each class has a specific use and is neither too big (encompassing so much functionality that it's unwieldy to reuse) nor annoyingly small (you can't use it by itself or without adding functionality). Your finished classes should themselves be easily reused.

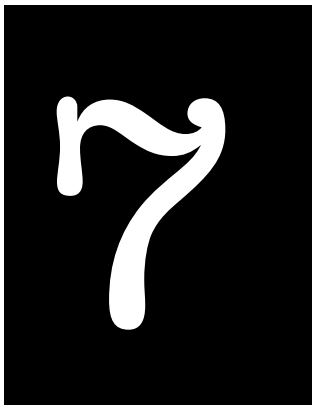
exercises

1. Create two classes, A and B, with default constructors (empty argument lists) that announce themselves. Inherit a new class called C from A, and create a member B inside C. Do not create a constructor for C. Create an object of class C and observe the results.

2. Modify exercise one so A and B have constructors with arguments instead of default constructors. Write a constructor for C and perform all initialization within C's constructor.

3. Take the file **Cartoon.java** and comment out the constructor for the **Cartoon** class. Explain what happens.

4. Take the file **Chess.java** and comment out the constructor for the **Chess** class. Explain what happens.



7: polymorphism

Polymorphism is the third essential feature of an object-oriented programming language, after data abstraction and inheritance.

It provides another dimension of separation of interface from implementation, to decouple *what* from *how*. Polymorphism allows improved code organization and readability as well as the creation of *extensible* programs that can be "grown," not only during the original creation of the project, but also when new features are desired.

Encapsulation creates new data types by combining characteristics and behaviors. Implementation hiding separates the interface from the implementation by making the details **private**. This sort of mechanical organization makes ready sense to someone with a procedural programming background. But polymorphism deals with decoupling in terms of *types*. In the last chapter, you saw how inheritance allows the treatment of an object as its own type *or* its base type. This ability is critical because it allows many types (derived from the same base type) to be treated as if they were one type, and a single piece of code to work on all those different types equally. The polymorphic method call allows one type to express its distinction from another, similar type, as long as they're both derived from the same base type. This distinction is expressed through differences in behavior of the methods you can call through the base class.

In this chapter, you'll learn about polymorphism (also called *dynamic binding*) starting from the very basics, with simple examples that strip away everything but the polymorphic behavior of the program.

upcasting

In the last chapter you saw how an object can be used as its own type or as an object of its base type. Taking an object handle and treating it as the handle of the base type is called *upcasting* because of the way inheritance trees are drawn with the base class at the top.

You also saw a problem arise, which is embodied in the following:

```
//: Music.java
// Inheritance & upcasting

class Note {
  private int value;
  private Note(int val) { value = val; }
  public static final Note
    middleC = new Note(0),
    cSharp = new Note(1),
    cFlat = new Note(2);
} // Etc.

class Instrument {
  public void play(Note n) {
    System.out.println("Instrument.play()");
  }
}

// Wind objects are instruments
// because they have the same interface:
class Wind extends Instrument {
  // Redefine interface method:
  public void play(Note n) {
    System.out.println("Wind.play()");
  }
}

public class Music {
  public static void tune(Instrument i) {
    // ...
    i.play(Note.middleC);
  }
  public static void main(String[] args) {
    Wind flute = new Wind();
    tune(flute); // Upcasting
  }
} ///:~
```

The method **Music.tune( )** accepts an **Instrument** handle, but also without complaint anything derived from **Instrument**. In **main( )**, you can see this happening as a **Wind** handle is passed to **tune( )**, with no cast necessary. This is acceptable; the interface in **Instrument** must exist in **Wind**, because **Wind** is inherited from **Instrument**. Upcasting from **Wind** to **Instrument** may "narrow" that interface, but it cannot make it any less than the full interface to **Instrument**.

why upcast?

This program may seem strange to you. Why should anyone intentionally *forget* the type of an object? This is what happens when you upcast, and it seems like it could be much more straightforward if **tune( )** simply takes a **Wind** handle as its argument. This brings up an essential point: if you did that, you'd have to write a new **tune( )** for every type of **Instrument** in your system. Suppose we follow this reasoning and add **string** and **brass** instruments:

```
//: Music2.java
// Overloading instead of upcasting

class Note2 {
  private int value;
  private Note2(int val) { value = val; }
  public static final Note2
    middleC = new Note2(0),
    cSharp = new Note2(1),
    cFlat = new Note2(2);
} // Etc.

class Instrument2 {
  public void play(Note2 n) {
    System.out.println("Instrument2.play()");
  }
}

class Wind2 extends Instrument2 {
  public void play(Note2 n) {
    System.out.println("Wind2.play()");
  }
}

class Stringed2 extends Instrument2 {
  public void play(Note2 n) {
    System.out.println("Stringed2.play()");
  }
}

class Brass2 extends Instrument2 {
  public void play(Note2 n) {
    System.out.println("Brass2.play()");
  }
}

public class Music2 {
  public static void tune(Wind2 i) {
    i.play(Note2.middleC);
  }
  public static void tune(Stringed2 i) {
    i.play(Note2.middleC);
  }
  public static void tune(Brass2 i) {
    i.play(Note2.middleC);
  }
  public static void main(String[] args) {
    Wind2 flute = new Wind2();
    Stringed2 violin = new Stringed2();
    Brass2 frenchHorn = new Brass2();
    tune(flute); // No upcasting
    tune(violin);
    tune(frenchHorn);
  }
} ///:~
```

This works, but there's a major drawback: you must write type-specific methods for each new **Instrument2** class you add. This means more programming in the first place, but it also means that if you want to add a new method like **tune( )** or a new type of Instrument you've got a lot of work to do.

Add to that the fact that the compiler won't give you any error messages if you forget to overload one of your methods and the whole process of working with types becomes unmanageable.

Wouldn't it be much nicer if you could just write a single method that takes the base class as its argument, and not any of the specific derived classes. That is, wouldn't it be nice if you could forget that there are derived classes, and only write your code to talk to the base class?

That's exactly what polymorphism allows you to do. However, most programmers (who come from a procedural programming background) have a bit of trouble with the way polymorphism works.

the twist

The difficulty with **Music.java** can be seen by running the program. The output is **Wind.play( )**. This is clearly the desired output, but it doesn't seem to make sense that it would work that way. Look at the **tune( )** method:

```
public static void tune(Instrument i) {
  // ...
  i.play(Note.middleC);
}
```

It receives an **Instrument** handle. So how can the compiler possibly know that this **Instrument** handle happens to be pointing to a **Wind** in this case and not a **Brass** or **Stringed**? Actually, the compiler can't. To get a deeper understanding of the issue, it's useful to examine the subject of *binding*.

method call binding

Connecting a method call to a method body is called *binding*. When binding is performed before the program is run (by the compiler and linker, if there is one), it's called *early binding*. You may not have heard the term before because it has never been an option with procedural languages: C compilers have only one kind of method call, and that's early binding.

The confusing part of the above program revolves around early binding because the compiler cannot know the correct method to call when it only has an **Instrument** handle.

The solution is called *late binding*, which means the binding occurs at run-time, based on the type of the object. Late binding is also called *dynamic binding* or *run-time binding*. When a language implements late binding, there must be some mechanism to determine the type of the object at run-time and call the appropriate method. That is, the compiler still doesn't know the actual object type, but the method-call mechanism finds out and calls the correct method body. The late-binding mechanism varies from language to language, but you can imagine that some sort of type information must be installed in the objects themselves.

All method binding in Java is late binding, unless a method has been declared **final** (which is the other reason for the existence of the **final** keyword). This means you ordinarily don't have to make any decisions about whether late binding will occur – it just happens automatically.

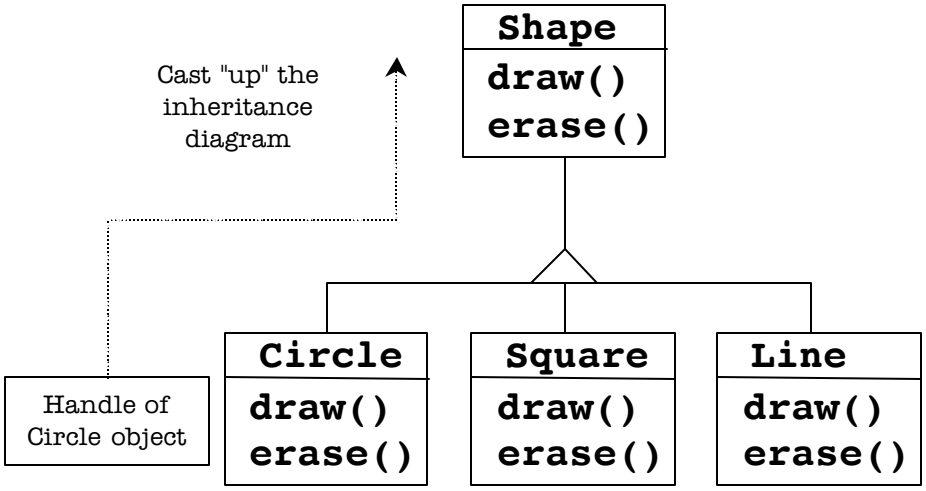
Why would you declare a method **final**? As noted in the last chapter, it prevents anyone from overriding that method. Perhaps more importantly, it effectively "turns off" dynamic binding, or rather it tells the compiler that dynamic binding isn't necessary. This allows the compiler to generate more efficient code for **final** method calls.

producing the right behavior

Once you know that all method binding in Java happens polymorphically via late binding, you can always write your code to talk to the base-class and know that all the derived-class cases will work correctly using the same code. Or to put it another way, you "send a message to an object and let the object figure out the right thing to do."

The classic example in OOP is the "shapes" example. This is very commonly used because it is easy to visualize, but unfortunately it can confuse novice programmers into thinking that OOP is just for graphics programming, which is of course not the case.

The shapes example has a base class called **Shape** and various derived types: **Circle**, **Square**, **Triangle**, etc. The reason the example works so well is that it's very easy to say "a circle is a type of shape" and be understood. The inheritance diagram shows the relationships:



The upcast could occur in a statement as simple as:

```
Shape s = new Circle();
```

Here, a **Circle** object is created and the resulting handle is immediately assigned to a **Shape**, which would seem to be an error (assigning one type to another) and yet it's fine because a **Circle** *is* a **Shape** by inheritance. So the compiler agrees with the statement and doesn't issue an error message.

When you call one of the base class methods (that have been overridden in the derived classes):

```
s.draw();
```

again, you might expect that **Shape**'s **draw( )** is called because this is, after all, a **Shape** handle so how could the compiler know to do anything else? And yet the proper **Circle.draw( )** is called because of late binding (polymorphism).

The following example puts it a slightly different way:

```
//: Shapes.java
// Polymorphism in Java

class Shape {
  void draw() {}
  void erase() {}
}

class Circle extends Shape {
  void draw() {
    System.out.println("Circle.draw()");
  }
  void erase() {
    System.out.println("Circle.erase()");
  }
}
```

```
class Square extends Shape {
  void draw() {
    System.out.println("Square.draw()");
  }
  void erase() {
    System.out.println("Square.erase()");
  }
}

class Triangle extends Shape {
  void draw() {
    System.out.println("Triangle.draw()");
  }
  void erase() {
    System.out.println("Triangle.erase()");
  }
}

public class Shapes {
  public static Shape randShape() {
    switch((int)(Math.random() * 3)) {
      default: // To quiet the compiler
      case 0: return new Circle();
      case 1: return new Square();
      case 2: return new Triangle();
    }
  }
  public static void main(String args[]) {
    Shape s[] = new Shape[9];
    // Fill up the array with shapes:
    for(int i = 0; i < s.length; i++)
      s[i] = randShape();
    // Make polymorphic method calls:
    for(int i = 0; i < s.length; i++)
      s[i].draw();
  }
} ///:~
```

The base class **Shape** establishes the common interface to anything inherited from **Shape** – that is, all shapes can be drawn and erased. The derived classes override these definitions to provide unique behavior for each specific type of shape.

The main class **Shapes** contains a **static** method **randShape( )** that produces a handle to a randomly-selected **Shape** object each time you call it. Notice that the upcasting is happening in each of the **return** statements, which take a handle to a **Circle**, **Square** or **Triangle** and send it out of the method as the return type, **Shape**. Thus when you call this method you never get a chance to see what specific type it is, since you always get back a plain **Shape** handle.

main( ) contains an array of **Shape** handles which is filled through calls to **randShape( )**. At this point you know you have **Shape**s, but you don't know anything more specific than that (and neither does the compiler). However, when you step through this array and call **draw( )** for each one, the correct type-specific behavior magically occurs, as you can see from one output example:

```
Circle.draw()
Triangle.draw()
Circle.draw()
Circle.draw()
Circle.draw()
Square.draw()
Triangle.draw()
Square.draw()
```

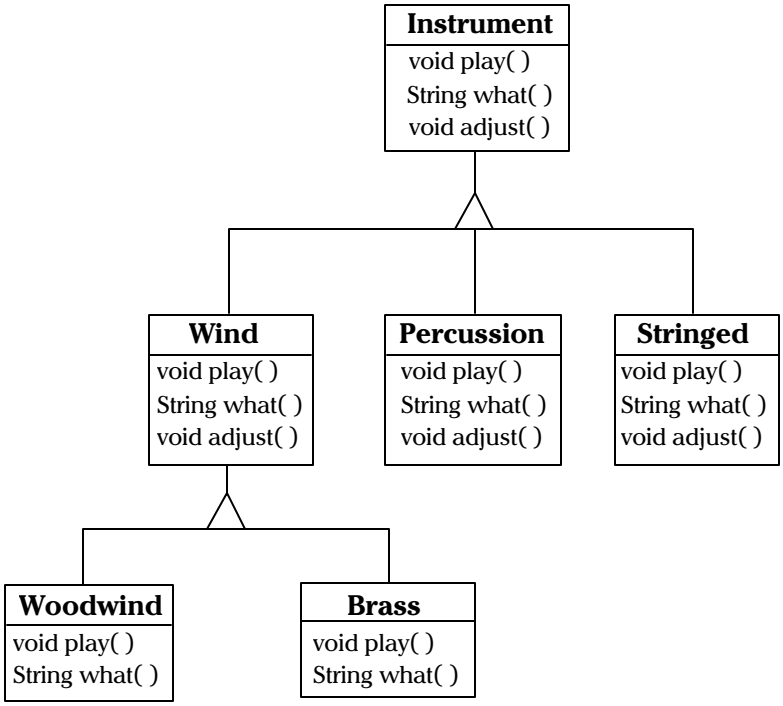
```
Square.draw()
```

Of course, since the shapes are all chosen randomly each time, your runs will have different results. The point of choosing the shapes randomly is to drive home the understanding that the compiler can have no special knowledge that allows it to make the correct calls at compile time. All the calls to **draw( )** are made through dynamic binding.

extensibility

Now return to the musical instrument example. Because of polymorphism you can add as many new types as you want to the system without changing the **tune( )** method. In a well-designed OOP program, most or all of your methods will follow the model of **tune( )** and communicate only with the base-class interface. Such a program is *extensible* because you can add new functionality by inheriting new data types from the common base class. The methods that manipulate the base-class interface will not need to be changed at all to accommodate the new classes.

Consider what happens if you take the instrument example and add more methods in the base class and a number of new classes. Here's the diagram:



All these brand new classes work correctly with the old, unchanged **tune( )** method. Even if **tune( )** is in a separate file and new methods are added to the interface of **Instrument**, **tune( )** works correctly without recompilation. Here is the implementation of the above diagram:

```
//: Music3.java
// An extensible program
import java.util.*;

class Instrument3 {
  public void play() {
    System.out.println("Instrument3.play()");
  }
  public String what() {
    return "Instrument3";
  }
  public void adjust() {}
```

```
}

class Wind3 extends Instrument3 {
  public void play() {
    System.out.println("Wind3.play()");
  }
  public String what() { return "Wind3"; }
  public void adjust() {}
}

class Percussion3 extends Instrument3 {
  public void play() {
    System.out.println("Percussion3.play()");
  }
  public String what() { return "Percussion3"; }
  public void adjust() {}
}

class Stringed3 extends Instrument3 {
  public void play() {
    System.out.println("Stringed3.play()");
  }
  public String what() { return "Stringed3"; }
  public void adjust() {}
}

class Brass3 extends Wind3 {
  public void play() {
    System.out.println("Brass3.play()");
  }
  public String what() { return "Brass3"; }
}

class Woodwind3 extends Wind3 {
  public void play() {
    System.out.println("Woodwind3.play()");
  }
  public String what() { return "Woodwind3"; }
}

public class Music3 {
  // Doesn't care about type, so new types
  // added to the system still work right:
  static void tune(Instrument3 i) {
    // ...
    i.play();
  }
  static void tuneAll(Instrument3[] e) {
    for(int i = 0; i < e.length; i++)
      tune(e[i]);
  }
  public static void main(String args[]) {
    Instrument3[] orchestra = new Instrument3[5];
    int i = 0;
    // Upcasting during addition to the array:
    orchestra[i++] = new Wind3();
    orchestra[i++] = new Percussion3();
    orchestra[i++] = new Stringed3();
    orchestra[i++] = new Brass3();
    orchestra[i++] = new Woodwind3();
```

```
    tuneAll(orchestra);
  }
} ///:~
```

The new methods are **what( )**, which returns a **String** handle with a description of the class, and **adjust( )**, which provides some way to adjust each instrument.

In **main( )**, when you place something inside the **Instrument3** array you automatically upcast to **Instrument3**.

You can see the **tune( )** method is blissfully ignorant of all the code changes that have happened around it, and yet it works correctly. This is exactly what polymorphism is supposed to provide: your code changes don't cause damage to parts of the program that should not be affected. Put another way, polymorphism is one of the most important techniques that allow the programmer to "separate the things that change from the things that stay the same."

overriding vs. overloading

Let's take a different look at the first example in this chapter. In the following program, the interface of the method **play( )** is changed in the process of overriding it, which means you haven't actually *overridden* the method, but instead *overloaded* it. The compiler allows you to overload methods so it gives no complaint. But the behavior is probably not what you want. Here's the example:

```
//: WindError.java
// Accidentally changing the interface

class NoteX {
  public static final int
    middleC = 0, cSharp = 1, cFlat = 2;
}

class InstrumentX {
  public void play(int NoteX) {
    System.out.println("InstrumentX.play()");
  }
}

class WindX extends InstrumentX {
  // OOPS! Changes the method interface:
  public void play(NoteX n) {
    System.out.println("WindX.play(NoteX n)");
  }
}

public class WindError {
  public static void tune(InstrumentX i) {
    // ...
    i.play(NoteX.middleC);
  }
  public static void main(String[] args) {
    WindX flute = new WindX();
    tune(flute); // Not the desired behavior!
  }
} ///:~
```

There's another confusing aspect thrown in here. In **InstrumentX**, the **play( )** method takes an **int** which has the identifier **NoteX**. That is, even though **NoteX** is a class name it can also be used as an identifier without complaint. But in **WindX**, **play( )** takes a **NoteX** handle that has an identifier **n** (although you could even say **play(NoteX NoteX)** without an error). Thus it appears the programmer

intended to override **play( )** but just mistyped the method a little bit. The compiler, however, assumed that an overload and not an override was intended.

In **tune**, the **InstrumentX i** is sent the **play( )** message, with one of **NoteX**'s members (**middleC**) as an argument. Since **NoteX** contains **int** definitions, this means that the **int** version of the now-overloaded **play( )** method is called, and since that has *not* been overridden the base-class version is used.

The output is:

```
InstrumentX.play()
```

Which certainly doesn't appear to be a polymorphic method call. Once you understand what's happening you can fix the problem fairly easily, but imagine how difficult it might be to find the bug if it's buried in a program of significant size.

abstract classes & methods

In all the instrument examples, the methods in the base class **Instrument** were always "dummy" methods. If these methods are ever called, you've done something wrong. That's because the intent of **Instrument** is to create a *common interface* for all the classes derived from it.

The only reason to establish this common interface is so it can be expressed differently for each different subtype. It establishes a basic form, so you can say what's in common with all the derived classes. Nothing else. Another way of saying this is to call **Instrument** an *abstract base class* (or simply an *abstract class*). You create an abstract class when you want to manipulate a set of classes through this common interface. All derived-class methods that match the signature of the base-class declaration will be called using the dynamic binding mechanism (as seen in the last section, if the method's name is the same as the base class but the arguments are different, you've got overloading which probably isn't what you want).

If you have an abstract class like **Instrument**, objects of that class almost always have no meaning. That is, **Instrument** is meant to express only the interface, and not a particular implementation, so creating an **Instrument** object makes no sense, and you'll probably want to prevent the user from doing it. This can be accomplished by making all the methods in **Instrument** print error messages, but this delays the information until run-time and requires reliable exhaustive testing on the part of the user. It's always better to catch problems at compile time.

Java provides a mechanism for doing this called the *abstract method*. This is a method that is incomplete; it has only a declaration and no method body. Here is the syntax for an abstract method declaration:

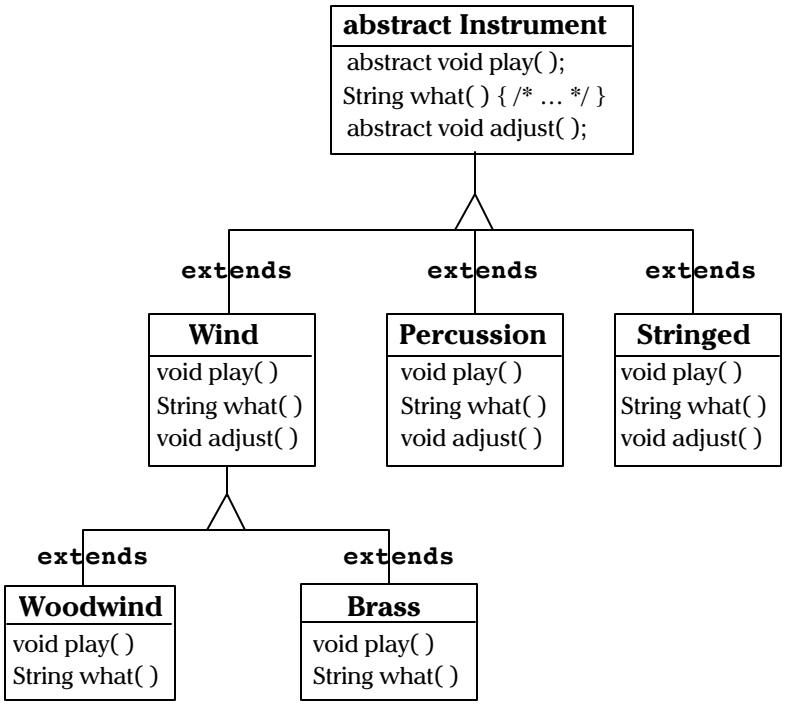
```
abstract void X();
```

A class containing abstract methods is called an *abstract class*. If a class contains one or more abstract methods, the class itself must be qualified as **abstract** (otherwise the compiler gives you an error message).

If an abstract class is incomplete, what is the compiler supposed to do when someone tries to make an object of that class? It cannot safely create an object of an abstract class, so you get an error message from the compiler. Thus, the compiler ensures the purity of the abstract class, and you don't have to worry about misusing it.

If you inherit from an abstract class and you want to make objects of the new type, you must provide method definitions for all the abstract methods in the base class. If you don't (and you may choose not to) then the derived class is also abstract and the compiler will force you to qualify *that* class with the **abstract** keyword.

The **Instrument** class can easily be turned into an abstract class. Only some of the methods will be abstract, since making a class abstract doesn't force you to make all the methods abstract. Here's what it looks like:



Here's the orchestra example modified to use **abstract** classes and methods:

```
//: Music4.java
// Abstract classes and methods
import java.util.*;

abstract class Instrument4 {
  int i; // storage allocated for each
  public abstract void play();
  public String what() {
    return "Instrument4";
  }
  public abstract void adjust();
}

class Wind4 extends Instrument4 {
  public void play() {
    System.out.println("Wind4.play()");
  }
  public String what() { return "Wind4"; }
  public void adjust() {}
}

class Percussion4 extends Instrument4 {
  public void play() {
    System.out.println("Percussion4.play()");
  }
  public String what() { return "Percussion4"; }
  public void adjust() {}
}
```

```
class Stringed4 extends Instrument4 {
  public void play() {
    System.out.println("Stringed4.play()");
  }
  public String what() { return "Stringed4"; }
  public void adjust() {}
}

class Brass4 extends Wind4 {
  public void play() {
    System.out.println("Brass4.play()");
  }
  public String what() { return "Brass4"; }
}

class Woodwind4 extends Wind4 {
  public void play() {
    System.out.println("Woodwind4.play()");
  }
  public String what() { return "Woodwind4"; }
}

public class Music4 {
  // Doesn't care about type, so new types
  // added to the system still work right:
  static void tune(Instrument4 i) {
    // ...
    i.play();
  }
  static void tuneAll(Instrument4[] e) {
    for(int i = 0; i < e.length; i++)
      tune(e[i]);
  }
  public static void main(String args[]) {
    Instrument4[] orchestra = new Instrument4[5];
    int i = 0;
    // Upcasting during addition to the array:
    orchestra[i++] = new Wind4();
    orchestra[i++] = new Percussion4();
    orchestra[i++] = new Stringed4();
    orchestra[i++] = new Brass4();
    orchestra[i++] = new Woodwind4();
    tuneAll(orchestra);
  }
} ///:~
```

You can see that there's really no change except in the base class.

It's helpful to create **abstract** classes and methods because they make the abstractness of a class explicit and tell both the user and the compiler how it was intended to be used.

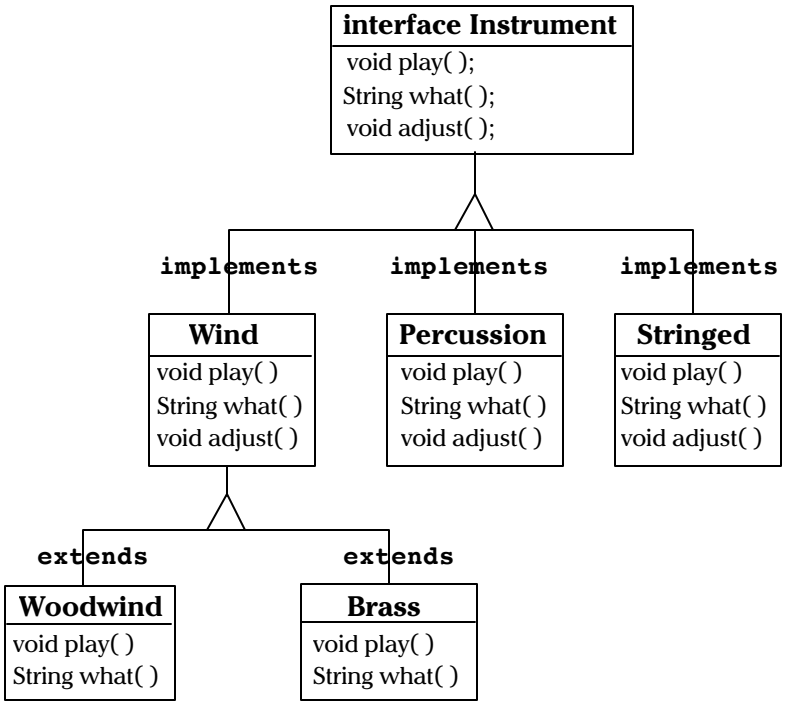
interfaces

The **interface** keyword takes the abstract concept one step further. You could think of it as a "pure" abstract class. It allows the creator to establish the form for a class: method names, argument lists and return types, but no method bodies. It's possible to add initialized definitions for primitives, but these become compile-time constants: they are **static** and **final** and thus cause no storage to be created for the actual object. An **interface** only provides a form, but no implementation.

An **interface** says: "this is what all classes that *implement* this particular interface shall look like." Thus, any code that uses a particular **interface** knows what methods may be called for that **interface**, and that's all. So the **interface** is used to establish a "protocol" between classes (some object-oriented programming languages have a keyword called *protocol* to do the same thing).

To create an **interface**, you use the **interface** keyword instead of the **class** keyword. Like a class, you can add the **public** keyword before the **interface** keyword (but only if that **interface** is defined in a file of the same name) or leave it off to give "friendly" status.

To make a class that conforms to a particular **interface** (or group of **interface**s) you use the **implements** keyword. You're saying "the **interface** is what it looks like, here's how it *works*." Other than that, it bears a strong resemblance to inheritance. The diagram for the instrument example shows this:



Once you've implemented an **interface**, that implementation becomes an ordinary class which may be extended in the regular way.

For methods, an **interface** *only* allows declarations. That is, you cannot have method bodies. An **interface** can also contain data members of primitive types, but these are implicitly **static** and **final.** They have the effect of automatically being compile-time constants[1] and do not require any storage to be created for the **interface** (this is the point of an **interface** – it's only a description and doesn't use any storage or even have a physical representation). You can see this in the modified version of the **Instrument** example. Notice that every method in the **interface** is strictly a declaration, which is the only thing the compiler will allow.

You can choose to explicitly declare the method declarations in an **interface** as **public**. However, they are **public** even if you don't say it. This means that when you **implement** an **interface**, the methods from the **interface** must be defined as **public.** Otherwise they would default to "friendly" and you'd be restricting the accessibility of a method during inheritance, which is not allowed by the Java

[1] This is used to create the equivalent of an **enum** when converting code from C or C++ to Java. You create an **interface** containing all your enumeration identifiers, giving each one a unique integer value. It's not ideal, but it's tolerable.

compiler. In the following example, none of the methods in **Instrument5** are declared as **public**, but they're automatically **public** anyway:

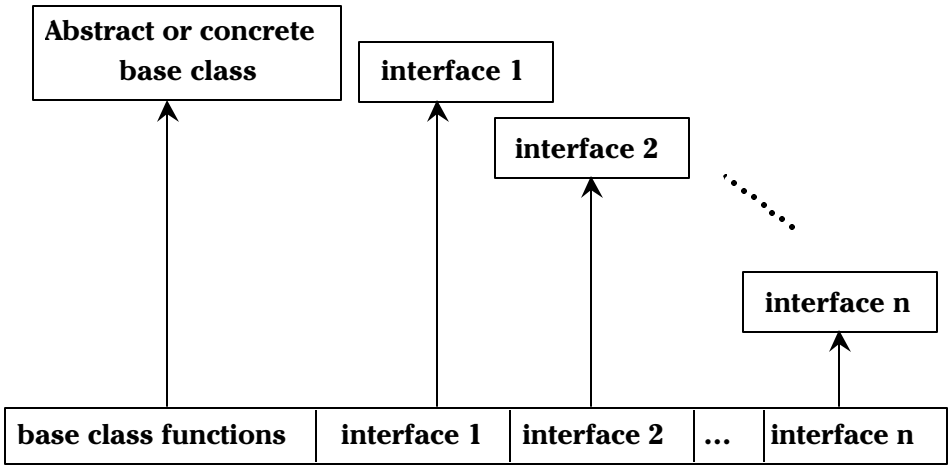
```
//: Music5.java
// Interfaces
import java.util.*;

interface Instrument5 {
  // Compile-time constant:
  int i = 5; // static & final
  // Cannot have method definitions:
  void play(); // Automatically public
  String what();
  void adjust();
}

class Wind5 implements Instrument5 {
  public void play() {
    System.out.println("Wind5.play()");
  }
  public String what() { return "Wind5"; }
  public void adjust() {}
}

class Percussion5 implements Instrument5 {
  public void play() {
    System.out.println("Percussion5.play()");
  }
  public String what() { return "Percussion5"; }
  public void adjust() {}
}

class Stringed5 implements Instrument5 {
  public void play() {
    System.out.println("Stringed5.play()");
  }
  public String what() { return "Stringed5"; }
  public void adjust() {}
}

class Brass5 extends Wind5 {
  public void play() {
    System.out.println("Brass5.play()");
  }
  public String what() { return "Brass5"; }
}

class Woodwind5 extends Wind5 {
  public void play() {
    System.out.println("Woodwind5.play()");
  }
  public String what() { return "Woodwind5"; }
}

public class Music5 {
  // Doesn't care about type, so new types
  // added to the system still work right:
  static void tune(Instrument5 i) {
    // ...
    i.play();
```

```
  }
  static void tuneAll(Instrument5[] e) {
    for(int i = 0; i < e.length; i++)
      tune(e[i]);
  }
  public static void main(String args[]) {
    Instrument5[] orchestra = new Instrument5[5];
    int i = 0;
    // Upcasting during addition to the array:
    orchestra[i++] = new Wind5();
    orchestra[i++] = new Percussion5();
    orchestra[i++] = new Stringed5();
    orchestra[i++] = new Brass5();
    orchestra[i++] = new Woodwind5();
    tuneAll(orchestra);
  }
} ///:~
```

The rest of the code works the same. That is, it doesn't matter if you are upcasting to a "regular" class called **Instrument5**, an **abstract** class called **Instrument5**, or an **interface** called **Instrument5**. The behavior is the same. In fact, you can see in the **tune( )** method that there isn't even any evidence about whether **Instrument5** is a "regular" class, an **abstract** class or an **interface**. This is the intent: each approach gives the programmer different control over the way objects are created and used.

"multiple inheritance" in Java

The **interface** isn't simply a "more pure" form of **abstract** class. It has a higher purpose than that. Because an **interface** has no implementation at all – That is, there is no storage associated with an **interface** – there's nothing to prevent many **interface**s from being combined. This is valuable because there are times when you need to say: "an **x** is an **a** *and* a **b** *and* a **c**." In C++, this act of combining multiple class interfaces is called *multiple inheritance*, and it carries with it some rather sticky baggage because each class can have an implementation. In Java, you can perform the same act but only one of the classes can have an implementation, so the problems seen in C++ do not occur with Java when combining multiple interfaces:



Although you aren't forced to have an **abstract** or "concrete" (one with no **abstract** methods) base class, if you do you can only have a maximum of one. All the rest of the base elements must be **interface**s. You can have as many **interface**s as you want, and each one becomes an independent type that you can upcast to. The following example shows a concrete class combined with several **interface**s to produce a new class:

```
//: Adventure.java
// Multiple interfaces
```

```
import java.util.*;

interface CanFight {
  void fight();
}

interface CanSwim {
  void swim();
}

interface CanFly {
  void fly();
}

class ActionCharacter {
  public void fight() {}
}

class Hero extends ActionCharacter
    implements CanFight, CanSwim, CanFly {
  public void swim() {}
  public void fly() {}
}

public class Adventure {
  static void t(CanFight x) { x.fight(); }
  static void u(CanSwim x) { x.swim(); }
  static void v(CanFly x) { x.fly(); }
  static void w(ActionCharacter x) { x.fight(); }
  public static void main(String args[]) {
    Hero i = new Hero();
    t(i); // Treat it as a CanFight
    u(i); // Treat it as a CanSwim
    v(i); // Treat it as a CanFly
    w(i); // Treat it as an ActionCharacter
  }
} ///:~
```

Notice that the signature for **fight( )** is the same in the **interface CanFight** and the class **ActionCharacter**, and that **fight( )** is *not* provided with a definition in **Hero**. The rule for an **interface** is that you can inherit from it (as you shall see shortly) but then you've got another **interface**. If you want to create an object of the new type, it must be a class with all definitions provided. But even though **Hero** does not explicitly provide a definition for **fight( )**, the definition comes along with **ActionCharacter** so it is automatically provided and it's possible to create objects of **Hero**.

In class **Adventure**, you can see there are four methods which take as arguments the various interfaces and the concrete class. When a **Hero** object is created, it can be passed to any of these methods, which means it is being upcast to each **interface** in turn. Because of the way interfaces are designed in Java, this works without a hitch and without any particular effort on the part of the programmer.

Keep in mind that the core reason for interfaces is shown in the above example: to be able to upcast to more than one base type. However, a second reason for using interfaces is the same as using an **abstract** base class: to prevent the client programmer from making an object of this class and to establish that it is only an interface. This brings up a question: should you use an **interface** or an **abstract** class? Well, an **interface** gives you the benefits of an **abstract** class *and* the benefits of an **interface**, so if it's possible to create your base class without any method definitions or member variables you should always prefer **interface**s to **abstract** classes. In fact, if you know something is going to be a base class, your first choice should be to make it an **interface**, and only if you're forced to have method definitions or member variables should you change to an **abstract** class.

extending an interface with inheritance

You can easily add new method declarations to an **interface** using inheritance, and you can also combine several interfaces into a new interface with inheritance. In both cases you get a new **interface**, as seen in this example:

```
//: HorrorShow.java
// Extending an interface with inheritance

interface Monster {
  void menace();
}

interface DangerousMonster extends Monster {
  void destroy();
}

interface Lethal {
  void kill();
}

class DragonZilla implements DangerousMonster {
  public void menace() {}
  public void destroy() {}
}

interface Vampire
    extends DangerousMonster, Lethal {
  void drinkBlood();
}

class HorrorShow {
  static void u(Monster b) { b.menace(); }
  static void v(DangerousMonster d) {
    d.menace();
    d.destroy();
  }
  public static void main(String[] args) {
    DragonZilla if2 = new DragonZilla();
    u(if2);
    v(if2);
  }
} ///:~
```

DangerousMonster is a simple extension to **Monster** which produces a new interface. This is implemented in **DragonZilla**.

The syntax used in **Vampire** *only* works when inheriting interfaces. Normally, you can only use **extends** with a single class, but since an **interface** can be made from multiple other interfaces, **extends** can refer to multiple base interfaces when building a new **interface**. As you can see, the **interface** names are simply separated with commas.

inner classes

In Java 1.1 it's possible to place a class definition within another class definition. This is called an *inner class*. This is a useful feature because it allows you to group classes that logically belong together and to control the visibility of one within the other. However, it's important to understand that inner classes are distinctly different from composition.

You create an inner class just as you'd expect: by placing the class definition inside a surrounding class:

```
//: Parcel1.java
// Creating inner classes
package c07.parcel1;

public class Parcel1 {
  class Contents {
    private int i = 11;
    public int value() { return i; }
  }
  class Destination {
    private String label;
    Destination(String whereTo) {
      label = whereTo;
    }
    String readLabel() { return label; }
  }
  public static void main(String args[]) {
    Parcel1 p = new Parcel1();
    // What can you do with it? Create some
    // objects. Must use instance of outer class
    // to create instances of inner classes:
    Parcel1.Contents c = p.new Contents();
    Parcel1.Destination d =
      p.new Destination("Tanzania");
  }
} ///:~
```

But all that's happened so far is the class definitions for **Contents** and **Destination** have been hidden away inside **Parcel1**. This doesn't make handles or objects of those classes, however, so you must create them explicitly. This is done in **main( )**, and it reveals a somewhat odd syntax. Creating an object of the outer class follows the typical form:

```
Parcel1 p = new Parcel1();
```

If you want to make an object of the inner class, the type of that object is *OuterClassName.InnerClassName*, as you might guess. However, to create the object itself you don't follow the same form and refer to the outer class name **Parcel1** as you might expect, but instead you must use an *object* of the outer class to make an object of the inner class:

```
Parcel1.Contents c = p.new Contents();
```

Thus it's not possible to create an object of the inner class unless you already have an object of the outer class. This, as you shall see later, is because the object of the inner class is quietly connected to the object of the outer class that it was made from.

More typically, an outer class will have a method that returns a handle to an inner class, like this:

```
//: Parcel2.java
// Returning a handle to an inner class
package c07.parcel2;

public class Parcel2 {
  class Contents {
    private int i = 11;
    public int value() { return i; }
  }
  class Destination {
    private String label;
    Destination(String whereTo) {
      label = whereTo;
```

```
    }
    String readLabel() { return label; }
  }
  public Destination dest(String s) {
    return new Destination(s);
  }
  public Contents cont() { return new Contents(); }
  public static void main(String args[]) {
    Parcel2 p = new Parcel2();
    Parcel2.Contents c = p.cont();
    Parcel2.Destination d = p.dest("Tanzania");
  }
} ///:~
```

You still have a somewhat strange syntax for the declaration of a handle to an inner class, but the creation of the inner class object itself seems much more natural.

inner classes and upcasting

So far, the use of inner classes seems nothing more than awkward. After all, if it's hiding you're after, Java already has a perfectly good hiding mechanism – just allow the class to be "friendly" rather than creating it as an inner class.

However, inner classes really come into their own when you start upcasting to a base class, and in particular an **interface**. That's because the inner class can then be completely unseen and unavailable to anyone; all you get back is a handle to the base class or the **interface** and it's possible that you can't even find out the exact type, as shown here:

```
//: Parcel3.java
// Returning a handle to an inner class
package c07.parcel3;

abstract class Contents {
  abstract public int value();
}

interface Destination {
  String readLabel();
}

public class Parcel3 {
  private class PContents extends Contents {
    private int i = 11;
    public int value() { return i; }
  }
  protected class PDestination
      implements Destination {
    private String label;
    private PDestination(String whereTo) {
      label = whereTo;
    }
    public String readLabel() { return label; }
  }
  public Destination dest(String s) {
    return new PDestination(s);
  }
  public Contents cont() {
    return new PContents();
  }
  public static void main(String args[]) {
```

```
    Parcel3 p = new Parcel3();
    Contents c = p.cont();
    Destination d = p.dest("Tanzania");
  }
}

class Test {
  public static void main(String args[]) {
    Parcel3 p = new Parcel3();
    // Illegal -- can't access private class:
    //! Parcel3.PContents c = p.new PContents();
  }
} ///:~
```

Now **Contents** and **Destination** represent interfaces available to the client programmer (the **interface**, remember, automatically makes all its members **public**). For convenience, these are placed inside a single file, but ordinarily **Contents** and **Destination** would each be **public** in their own files.

In **Parcel3**, something new has been added: the inner class **PContents** is **private** so no one but **Parcel3** can access it. **PDestination** is **protected**, so no one but **Parcel3**, classes in the **Parcel3** package (since **protected** also gives package access), and the inheritors of **Parcel3** can access **PDestination**. This means that the client programmer has restricted knowledge and access to these members. In fact, you can't even downcast to a **private** inner class (or a **protected** inner class unless you're an inheritor), because you can't access the name, as you can see in **class Test**. Thus, the **private** inner class provides a way for the class designer to completely prevent any type-coding dependencies and to completely hide details about implementation. In addition, extension of an interface is useless from the client programmer's perspective since the client programmer cannot access any additional methods that aren't part of the public interface class. This also provides an opportunity for the Java compiler to generate more efficient code.

Remember that the ability to make a class **private** or **protected** does not apply to normal (non-inner) classes, which can only be **public** or "friendly."

Note that **Contents** is an **abstract** class but it doesn't have to be. You could use an ordinary class here as well, but the most typical starting point for such a design is an **interface**.

static inner classes

If you don't need to create an object of the outer class in order to create an object of the inner class, you can make everything **static**. But for this to work, you must also make the inner classes themselves **static**:

```
//: Parcel4.java
// Static inner classes
package c07.parcel4;

abstract class Contents {
  abstract public int value();
}

interface Destination {
  String readLabel();
}

public class Parcel4 {
  private static class PContents extends Contents {
    private int i = 11;
    public int value() { return i; }
  }
  protected static class PDestination
      implements Destination {
```

```
    private String label;
    private PDestination(String whereTo) {
      label = whereTo;
    }
    public String readLabel() { return label; }
  }
  public static Destination dest(String s) {
    return new PDestination(s);
  }
  public static Contents cont() {
    return new PContents();
  }
  public static void main(String args[]) {
    Contents c = Parcel4.cont();
    Destination d = Parcel4.dest("Tanzania");
  }
} ///:~
```

This is the same as the previous example except that the inner classes and the methods that return handles to objects of the inner classes are **static**. In **main( )**, no object of **Parcel4** is necessary; instead you use the normal syntax for selecting a **static** member to call the methods that return handles to **Contents** and **Destination**.

inner classes in methods & scopes

An inner class may also be created within a method or even an arbitrary scope. There are two reasons for doing this:

1. As shown previously, you're implementing an interface of some kind so you can create and return a handle.
2. You're solving a complicated problem and you want to create a class to aid in your solution, but you don't want it to be publicly used.

Here is the previous example modified to use:

1. A class defined within a method
2. A class defined within a scope inside a method
3. An anonymous class implementing an **interface**
4. An anonymous class extending a class that has a non-default constructor.

```
//: Parcel5.java
// Inner classes inside methods and scopes
package c07.parcel5;

abstract class Contents {
  abstract public int value();
}

interface Destination {
  String readLabel();
}

class Wrapping {
  private int i;
  public Wrapping(int x) { i = x; }
  public int value() { return i; }
}
```

```
public class Parcel5 {
  public Destination dest(String s) {
    class PDestination
        implements Destination {
      private String label;
      private PDestination(String whereTo) {
        label = whereTo;
      }
      public String readLabel() { return label; }
    }
    return new PDestination(s);
  }
  private void internalTracking(boolean b) {
    if(b) {
      class TrackingSlip {
        private String id;
        TrackingSlip(String s) {
          id = s;
        }
        String getSlip() { return id; }
      }
      TrackingSlip ts = new TrackingSlip("slip");
      String s = ts.getSlip();
    }
  }
  public Contents cont() {
    return new Contents() {
      private int i = 11;
      public int value() { return i; }
    }; // Semicolon required in this case
  }
  public Wrapping wrap(int x) {
    return new Wrapping(x) {
      public int value() {
        return super.value() * 47;
      }
    }; // Semicolon required
  }
  public static void main(String args[]) {
    Parcel5 p = new Parcel5();
    Contents c = p.cont();
    Destination d = p.dest("Tanzania");
  }
} ///:~
```

You'll note the addition of a new class, **Wrapping**, with a constructor that requires an argument.

Inside **Parcel5**, the class **PDestination** is now part of **dest( )** rather than being part of **Parcel5** (also notice that the identifier **PDestination** has been used over again several times in this chapter – and thus in the same subdirectory – without a name clash). Other than that, it looks the same. Notice the upcasting that occurs during the return statement – nothing comes out of **dest( )** except a handle to the base class **Destination**. Of course, the fact that the name of the class **PDestination** is placed inside **dest( )** doesn't mean that **PDestination** is not a valid type of object once **dest( )** returns.

The method **internalTracking** demonstrates the creation of an inner class inside an arbitrary scope (in this case, inside the **if** clause). Again, it looks just like an ordinary class.

In the next two parts of the example things look a little strange. The **cont( )** and **wrap( )** methods combine the creation of the return value with the definition of the class that represents that return value! In addition, the class is anonymous – it has no name. To make matters a bit worse, it looks like you're starting out to create a **Contents** object:

```
return new Contents()
```

but then, before you get to the semicolon you say "but wait, I think I'll slip in a class definition:

```
return new Contents() {
  private int i = 11;
  public int value() { return i; }
};
```

What this strange syntax means is "create an object of an anonymous class that's inherited from **Contents**." The result of the **new** expression is automatically upcast to a **Contents**.

In the above example, **Contents** is created using a default constructor. If your base class needs a constructor with an argument, you say:

```
public Wrapping wrap(int x) {
  return new Wrapping(x) {
    public int value() {
      return super.value() * 47;
    }
  }; // Semicolon required
}
```

That is, you just pass the appropriate argument to the base-class constructor. An anonymous class cannot have a constructor (although individual members may be initialized at their point of definition).

In both cases, the semicolon doesn't mark the end of the class body (as it does in C++) but the end of the expression that happens to contain the anonymous class. Thus it's identical to the use of the semicolon everywhere else.

the link to the outer class

So far, it appears that inner classes are just a name-hiding and code-organization scheme, which is helpful but not totally compelling. However, there's another twist. When you create an inner class, objects of that inner class have a link to the object that made them, and so can access the members of the enclosing object. In addition, inner classes have access rights to all the elements in the enclosing class[2]. The following example demonstrates this:

```
//: Sequence.java
// Holds a sequence of Objects

interface Selector {
  boolean end();
  Object current();
  void next();
}

public class Sequence {
  private Object[] o;
  private int next = 0;
  public Sequence(int size) {
    o = new Object[size];
  }
  public void add(Object x) {
    if(next < o.length) {
      o[next] = x;
```

[2] This is very different from the design of *nested classes* in C++, which is simply a name-hiding mechanism. There is no link to an enclosing object and no implied permissions in C++.

```
      next++;
    }
  }
  private class SSelector implements Selector {
    int i = 0;
    public boolean end() {
      return i == o.length;
    }
    public Object current() {
      return o[i];
    }
    public void next() {
      if(i < o.length) i++;
    }
  }
  public Selector getSelector() {
    return new SSelector();
  }
  public static void main(String args[]) {
    Sequence s = new Sequence(10);
    for(int i = 0; i < 10; i++)
      s.add(Integer.toString(i));
    Selector sl = s.getSelector();
    int counter = 0;
    while(!sl.end()) {
      System.out.println((String)sl.current());
      sl.next();
    }
  }
} ///:~
```

The **Sequence** is simply a fixed-sized array of **Object** with a class wrapped around it. You call **add( )** to add a new **Object** to the end of the sequence (if there's room left). To fetch each of the objects in a **Sequence**, there's an interface called **Selector**, which allows you to see if you're at the **end( )**, to look at the **current( ) Object**, and to move to the **next( ) Object** in the **Sequence**. Because **Selector** is an **interface**, many other classes may implement the **interface** in their own ways, and many methods may take the **interface** as an argument, in order to create generic code.

Here, the **SSelector** is a private class that provides **Selector** functionality. In **main( )**, you can see the creation of a **Sequence**, followed by the addition of a number of **String** objects. Then, a **Selector** is produced with a call to **getSelector( )** and this is used to move through the **Sequence** and select each item.

At first, the creation of **SSelector** looks like just another inner class. But examine it closely. Notice that each of the methods **end( )**, **current( )** and **next( )** refer to **o**, which is a handle that isn't part of **SSelector**, but is instead a **private** field in the enclosing class. However, the inner class can access methods and fields from the enclosing class as if they owned them. This turns out to be very convenient, as you can see in the above example.

So an inner class has access to the members of the enclosing class. But how can this happen? The inner class must keep a reference to the particular object of the enclosing class that was responsible for creating it. Then when you refer to a member of the enclosing class, that (hidden) reference is used to select that member. Fortunately, the compiler takes care of all these details for you, but you can also understand now that an object of an inner class can only be created in association with an object of the enclosing class. The process of construction requires the initialization of the handle to the object of the enclosing class, and the compiler will complain if it cannot access the handle. Most of the time this occurs without any intervention on the part of the programmer.

referring to the outer class object

If you need to produce the handle to the outer class object, you name the outer class followed by a dot and **this**. For example, in the above class **SSelector**, any of its methods can produce the stored

handle to the outer class **Sequence** by saying **Sequence.this**. The resulting handle is automatically the correct type (this is known and checked at compile time, so there is no run-time overhead).

To understand the meaning of **static** when applied to inner classes, you must keep in mind the idea that the object of the inner class implicitly keeps a handle to the object of the enclosing class that created it. When you say an inner class is **static**, this is not true, which means:

1. You don't need an outer-class object in order to create an object of a **static** inner class.
2. You can't access an outer-class object from an object of a **static** inner class.

inheriting from inner classes

Because the inner class constructor must attach to a handle of the enclosing class object, things are slightly complicated when you inherit from an inner class. The problem is that the "secret" handle to the enclosing class object *must* be initialized, and yet in the derived class there's no longer a default object to attach to. The answer is to use a syntax provided to make the association explicit:

```
//: InheritInner.java
// Inheriting an inner class

class WithInner {
  class Inner {}
}

public class InheritInner
    extends WithInner.Inner {
  //! InheritInner() {} // Won't compile
  InheritInner(WithInner wi) {
    wi.super();
  }
  public static void main(String args[]) {
    WithInner wi = new WithInner();
    InheritInner ii = new InheritInner(wi);
  }
} ///:~
```

You can see that **InheritInner** is only extending the inner class, not the outer one. But when it comes time to create a constructor, the default one is no good and you can't just pass a handle to an enclosing object. In addition, you must use the syntax

```
enclosingClassHandle.super();
```

inside the constructor. This provides the necessary handle and the program will then compile.

can inner classes be overridden?

What happens when you create an inner class, then inherit from the enclosing class and redefine the inner class? That is, is it possible to override an inner class? This seems like it would be a powerful concept, but "overriding" an inner class has no effect:

```
//: BigEgg.java
// "Overriding" an inner class has no effect

class Egg {
  class Yolk {
    Yolk() {
      System.out.println("Egg.Yolk()");
    }
  }
  Yolk y;
```

```
  Egg() {
    System.out.println("New Egg()");
    y = new Yolk();
  }
}

public class BigEgg extends Egg {
  class Yolk {
    Yolk() {
      System.out.println("BigEgg.Yolk()");
    }
  }
  public static void main(String args[]) {
    new BigEgg();
  }
} ///:~
```

The default constructor is synthesized automatically by the compiler, and this calls the base-class default constructor. You might think that since a **BigEgg** is being created the "overridden" version of **Yolk** would be used, but this is not the case. The output is:

```
New Egg()
Egg.Yolk()
```

class identifiers

As you might imagine, the creation of inner classes must produce **.class** files to contain the class objects. The names of these files/classes have a strict formula: the name of the enclosing class, followed by a '**$**' followed by the name of the inner class. For example, the **.class** files created by **InheritInner.java** include:

```
InheritInner.class
WithInner$Inner.class
WithInner.class
```

Although this scheme of generating internal names is simple and straightforward, it's also robust and handles most situations. Since it is the standard naming scheme for Java, the generated files are automatically platform-independent.

constructors & polymorphism

As usual, constructors are different from other kinds of methods. This is also true when polymorphism is involved. Even though constructors themselves are not polymorphic (although you can have a kind of "virtual constructor" as you will see in Chapter 11), it's important to understand the way constructors work in complex hierarchies and with polymorphism. This understanding will help you avoid unpleasant entanglements.

order of constructor calls

The order in which constructors are called was briefly discussed in Chapter 4, but that was before inheritance and polymorphism were introduced.

A constructor for the base class is always called in the constructor for a derived class, chaining upward so that a constructor for every base class is called. This makes sense because the constructor has a special job: to see that the object is built properly. A derived class has access only to its own members, and not to those of the base class (whose members are typically **private**). Only the base-class constructor has the proper knowledge and access to initialize its own elements. Therefore it's essential that all constructors get called, otherwise the entire object wouldn't be constructed properly. That's why the compiler enforces a constructor call for every portion of a derived class. It will silently call the

default constructor if you don't explicitly call a base-class constructor in the derived-class constructor body. If there is no default constructor, the compiler will complain. (In the case where a class has no constructors the compiler will automatically synthesize a default constructor.)

Let's take a look at an example that shows the effects of composition, inheritance and polymorphism on the order of construction:

```
//: Sandwich.java
// Order of constructor calls

class Meal {
  Meal() { System.out.println("Meal()"); }
}

class Bread {
  Bread() { System.out.println("Bread()"); }
}

class Cheese {
  Cheese() { System.out.println("Cheese()"); }
}

class Lettuce {
  Lettuce() { System.out.println("Lettuce()"); }
}

class Lunch extends Meal {
  Lunch() { System.out.println("Lunch()");}
}

class PortableLunch extends Lunch {
  PortableLunch() {
    System.out.println("PortableLunch()");
  }
}

class Sandwich extends PortableLunch {
  Bread b = new Bread();
  Cheese c = new Cheese();
  Lettuce l = new Lettuce();
  Sandwich() {
    System.out.println("Sandwich()");
  }
  public static void main(String[] args) {
    new Sandwich();
  }
} ///:~
```

This example creates a complex class out of other classes, and each class has a constructor that announces itself. The important class is **Sandwich**, which reflects three levels of inheritance (four, if you count the implicit inheritance from **Object**) and three member objects. When a **Sandwich** object is created in **main( )**, the output is:

```
Meal()
Lunch()
PortableLunch()
Bread()
Cheese()
Lettuce()
Sandwich()
```

This means that the order of constructor calls for a complex object is as follows:

1. The base-class constructor is called. This step is repeated recursively such that the very root of the hierarchy is constructed first, followed by the next-derived class, etc., until the most-derived class is reached.
2. Member initializers are called in the order of declaration.
3. The body of the derived-class constructor is called.

The order of the constructor calls is important. When you inherit, you know all about the base class and can access any **public** and **protected** members of the base class. This means you must be able to assume that all the members of the base class are valid when you're in the derived class. In a normal method, construction has already taken place, so all the members of all parts of the object have been built. Inside the constructor, however, you must be able to assume that all members that you use have been built. The only way to guarantee this is for the base-class constructor to be called first. Then when you're in the derived-class constructor, all the members you can access in the base class have been initialized. "Knowing all members are valid" inside the constructor is also the reason that, whenever possible, you should initialize all member objects (that is, objects placed in the class using composition) at their point of definition in the class (e.g.: **b**, **c** and **l** in the example above). If you follow this practice, you will help ensure that all base class members *and* member objects of the current object have been initialized. Unfortunately, this doesn't handle every case, as you will see in the next section.

inheritance and finalize( )

When you use composition to create a new class, you never worry about finalizing the member objects of that class. Each member is an independent object and thus is garbage collected and finalized regardless of whether it happens to be a member of your class. With inheritance, however, you must override **finalize( )** in the derived class if you have any special cleanup that must happen as part of garbage collection. When you override **finalize( )** in an inherited class, it's important to remember to call the base-class version of **finalize( )**, since otherwise the base-class finalization will not happen. The following example proves this:

```
//: Frog.java
// Testing finalize with inheritance

class DoBaseFinalization {
  public static boolean flag = false;
}

class Characteristic {
  String s;
  Characteristic(String c) {
    s = c;
    System.out.println(
      "Creating Characteristic " + s);
  }
  protected void finalize() {
    System.out.println(
      "finalizing Characteristic " + s);
  }
}

class LivingCreature {
  Characteristic p =
    new Characteristic("is alive");
  LivingCreature() {
    System.out.println("LivingCreature()");
  }
  protected void finalize() {
```

```
    System.out.println(
      "LivingCreature finalize");
    // Call base-class version LAST!
    if(DoBaseFinalization.flag)
      try {
        super.finalize();
      } catch(Throwable t) {}
  }
}

class Animal extends LivingCreature {
  Characteristic p =
    new Characteristic("has heart");
  Animal() {
    System.out.println("Animal()");
  }
  protected void finalize() {
    System.out.println("Animal finalize");
    if(DoBaseFinalization.flag)
      try {
        super.finalize();
      } catch(Throwable t) {}
  }
}

class Amphibian extends Animal {
  Characteristic p =
    new Characteristic("can live in water");
  Amphibian() {
    System.out.println("Amphibian()");
  }
  protected void finalize() {
    System.out.println("Amphibian finalize");
    if(DoBaseFinalization.flag)
      try {
        super.finalize();
      } catch(Throwable t) {}
  }
}

public class Frog extends Amphibian {
  Frog() {
    System.out.println("Frog()");
  }
  protected void finalize() {
    System.out.println("Frog finalize");
    if(DoBaseFinalization.flag)
      try {
        super.finalize();
      } catch(Throwable t) {}
  }
  public static void main(String args[]) {
    if(args.length != 0 &&
       args[0].equals("finalize"))
       DoBaseFinalization.flag = true;
    else
      System.out.println("not finalizing bases");
    new Frog(); // Instantly becomes garbage
    System.out.println("bye!");
    // Must do this to guarantee that all
```

```
    // finalizers will be called:
    System.runFinalizersOnExit(true);
  }
} ///:~
```

The class **DoBaseFinalization** simply holds a flag that indicates to each class in the hierarchy whether to call **super.finalize( )**. This flag is set based on a command-line argument, so you can view the behavior with and without base-class finalization.

Each class in the hierarchy also contains a member object of class **Characteristic**. You will see that regardless of whether the base class finalizers are called, the **Characteristic** member objects are always finalized.

Each overridden **finalize( )** must have access of at least **protected** since the **finalize( )** method in class **Object** is **protected** and the compiler will not allow you to reduce the access during inheritance ("friendly" is less accessible than **protected**).

In **Frog.main( )** the **DoBaseFinalization** flag is configured, and a single **Frog** object is created. Remember that garbage collection and in particular finalization may not happen for any particular object so to enforce this, **System.runFinalizersOnExit(true)** adds the extra overhead to guarantee that finalization takes place. Without base-class finalization, the output is:

```
not finalizing bases
Creating Characteristic is alive
LivingCreature()
Creating Characteristic has heart
Animal()
Creating Characteristic can live in water
Amphibian()
Frog()
bye!
Frog finalize
finalizing Characteristic is alive
finalizing Characteristic has heart
finalizing Characteristic can live in water
```

You can see that indeed, no finalizers are called for the base classes of **Frog**. But if you add the "finalize" argument on the command line, you get:

```
Creating Characteristic is alive
LivingCreature()
Creating Characteristic has heart
Animal()
Creating Characteristic can live in water
Amphibian()
Frog()
bye!
Frog finalize
Amphibian finalize
Animal finalize
LivingCreature finalize
finalizing Characteristic is alive
finalizing Characteristic has heart
finalizing Characteristic can live in water
```

Although the order in which the member objects are finalized is the same order in which they are created, technically the order of finalization of objects is unspecified. With base classes, however, you have control over the order of finalization. The best order to use is the one that's shown here, which is the reverse of the order of initialization. Following the form that's used in C++ for destructors, you want to perform the derived-class finalization first, then the base-class finalization. That's because the derived-class finalization could call some methods in the base class that require that the base-class components are still alive, so you must not destroy them prematurely.

behavior of polymorphic methods inside constructors

The hierarchy of constructor calls brings up an interesting dilemma. What happens if you're inside a constructor and you call a dynamically-bound method? Inside an ordinary method you can imagine what will happen — the dynamically-bound call is resolved at run-time because the object cannot know whether it belongs to the class the method is in, or some class derived from it. For consistency, you might think this is what should happen inside constructors.

This is not exactly the case. If you call a dynamically-bound method inside a constructor, the overridden definition for that method is in fact used. However, the *effect* can be rather unexpected, and can conceal some very difficult-to-find bugs.

Conceptually, the constructor's job is to bring the object into existence (which is hardly an ordinary feat). Inside any constructor, the entire object may only be partially formed — you can only know that the base-class objects have been initialized, but you cannot know which classes are inherited from you. A dynamically-bound method call, however, reaches "forward" or "outward" into the inheritance hierarchy. It calls a method in a derived class. If you do this inside a constructor, you call a method that might manipulate members that haven't been initialized yet: a sure recipe for disaster.

You can see the problem in the following example:

```
//: PolyConstructors.java
// Constructors and polymorphism
// don't produce what you might expect.

abstract class Glyph {
  abstract void draw();
  Glyph() {
    System.out.println("Glyph() before draw()");
    draw();
    System.out.println("Glyph() after draw()");
  }
}

class RoundGlyph extends Glyph {
  int radius = 1;
  RoundGlyph(int r) {
    radius = r;
    System.out.println(
      "RoundGlyph.RoundGlyph(), radius = "
      + radius);
  }
  void draw() {
    System.out.println(
      "RoundGlyph.draw(), radius = " + radius);
  }
}

public class PolyConstructors {
  public static void main(String[] args) {
    new RoundGlyph(5);
  }
} ///:~
```

In **Glyph**, the **draw( )** method is **abstract**, so it is designed to be overridden. Indeed, you are forced to override it in **RoundGlyph**. But the **Glyph** constructor calls this method, and the call ends up in **RoundGlyph.draw( )**, which would seem to be the intent. But look at the output:

```
Glyph() before draw()
```

```
RoundGlyph.draw(), radius = 0
Glyph() after draw()
RoundGlyph.RoundGlyph(), radius = 5
```

When **Glyph**'s constructor calls **draw( )**, the value of **radius** isn't even the default initial value 1. It's zero. This would probably result in either a dot or nothing at all being drawn on the screen, and you'd be staring, trying to figure out why the program won't work.

The order of initialization described in the previous section isn't quite complete, and that's the key to solving the mystery. The actual process of initialization is:

1. The storage allocated for the object is initialized to binary zero before anything else happens.
2. The base-class constructors are called as described previously. At this point, the overridden **draw( )** method is called, (yes, *before* the **RoundGlyph.draw( )** constructor is called) which discovers a **radius** value of zero, due to step one.
3. Member initializers are called in the order of declaration.
4. The body of the derived-class constructor is called.

There's an upside to this, which is that everything is at least initialized to zero (or whatever zero means for that particular data type) and not just left as garbage. This includes object handles that are embedded inside a class via composition, and so if you forget to initialize that handle you'll get an exception at run time. Everything else gets zero, which is usually a telltale value when looking at output.

On the other hand, you should be pretty horrified at the outcome of this program. You've done a perfectly logical thing and yet the behavior is mysteriously wrong, with no complaints from the compiler. Bugs like this could easily be buried and take a long time to discover.

As a result, a good guideline for constructors is "do as little as possible to set the object into a good state, and don't call any methods." The only safe methods to call inside a constructor are those that are **final** in the base class. These cannot be overridden and thus cannot produce this kind of surprise.

designing with inheritance

Once you learn about polymorphism, it can seem that everything ought to be inherited because polymorphism is such a clever tool. This can burden your designs; in fact if you choose inheritance first when you're using an existing class to make a new class things can become needlessly complicated.

A better approach is to choose composition first, when it's not obvious which one you should use. Composition does not force a design into an inheritance hierarchy. But composition is also more flexible since it's possible to dynamically choose a type (and thus behavior) when using composition, whereas inheritance requires an exact type to be known at compile time. The following example illustrates this:

```
//: Transmogrify.java
// Dynamically changing the behavior of
// an object via composition.

interface Actor {
  void act();
}

class HappyActor implements Actor {
  public void act() {
    System.out.println("HappyActor");
  }
}
```

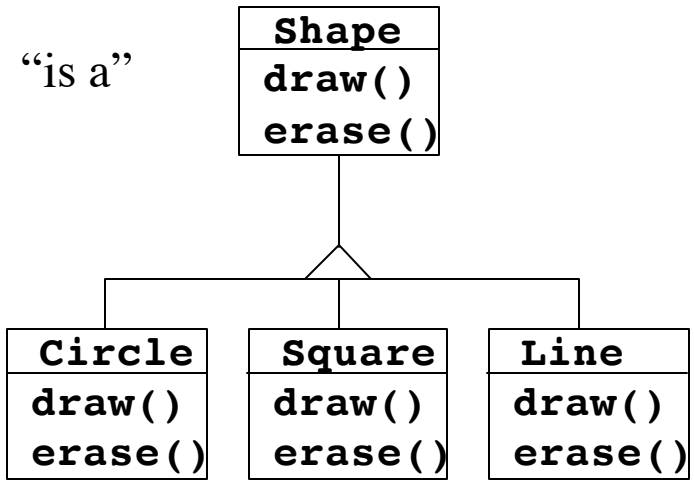
```
class SadActor implements Actor {
  public void act() {
    System.out.println("SadActor");
  }
}

class Stage {
  Actor a = new HappyActor();
  void change() { a = new SadActor(); }
  void go() { a.act(); }
}

public class Transmogrify {
  public static void main(String[] args) {
    Stage s = new Stage();
    s.go(); // Prints "HappyActor"
    s.change();
    s.go(); // Prints "SadActor"
  }
} ///:~
```

A **Stage** object contains a handle to an **Actor**, which is initialized to a **HappyActor** object. This means **go( )** produces a particular behavior. But since a handle can be re-bound to a different object at run time, a handle for a **SadActor** object can be substituted in **a** and then the behavior produced by **go( )** changes. Thus you gain dynamic flexibility at run time. In contrast, you can't decide to inherit differently at run time; that must be completely determined at compile time.

A general guideline is "use inheritance to express differences in behavior, and member variables to express variations in state." In the above example, both are used: two different classes are inherited to express the difference in the **act( )** method, and **Stage** uses composition to allow its state to be changed. In this case, that change in state happens to produce a change in behavior.

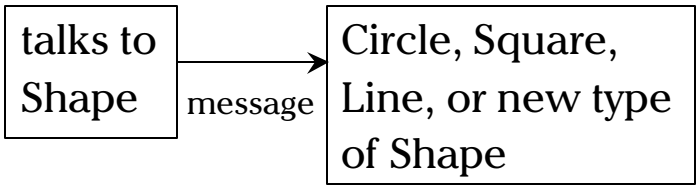
pure inheritance vs. extension

When studying inheritance, it would seem that the cleanest way to create an inheritance hierarchy is to take the "pure" approach. That is, only methods that have been established in the base class or **interface** are to be overridden in the derived class, as seen in this diagram:



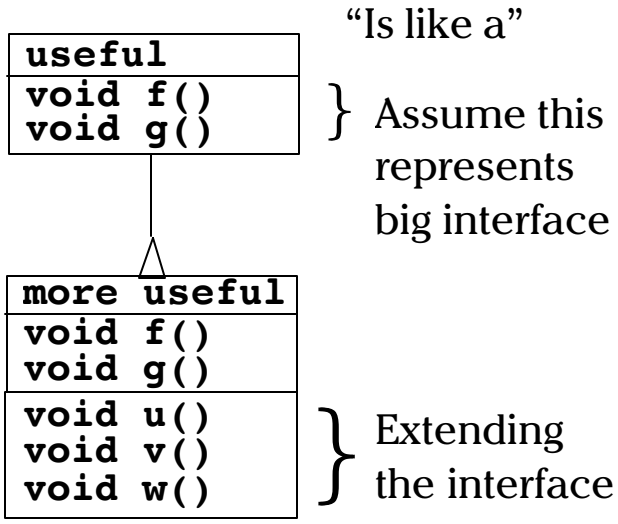
This can be termed a pure "is-a" relationship, because the interface of a class establishes what it is. Inheritance guarantees that any derived class will have the interface of the base class and nothing less. If you follow the above diagram, derived classes will also have *no more* than the base class interface.

This can be thought of as *pure substitution*, because it means that derived class objects can be perfectly substituted for the base class, and you never need to know any extra information about the subclasses when you're using them:

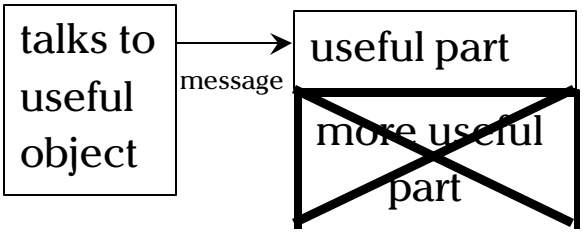


That is, the base class can receive any message you can send to the derived class because the two have exactly the same interface. This means that all you have to do is upcast from the derived class and never look back to see what exact type of object you're dealing with. Everything is handled through polymorphism.

When you see it this way, it seems like a pure "is-a" relationship is the only sensible way to do things, and any other design indicates muddled thinking and is by definition broken. This too is a trap. As soon as you start thinking this way, you'll turn around and discover that extending the interface (which, unfortunately, the keyword **extend** seems to promote) is the perfect solution to a particular problem. This could be termed an "is-like-a" relationship because the derived class is *like* the base class – it has the same fundamental interface – but it has other features that require additional methods to implement:



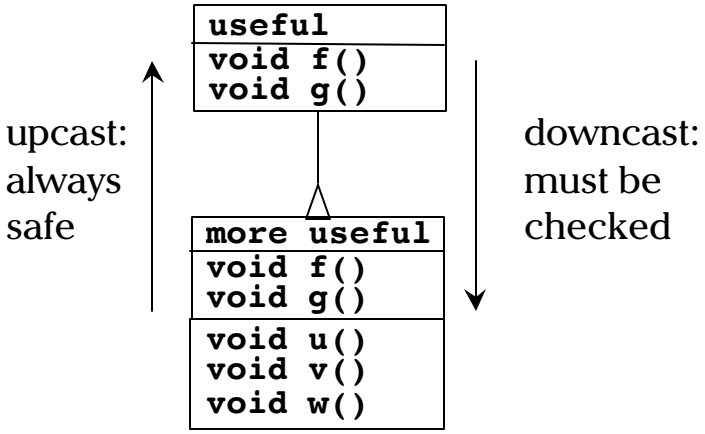
While this is also a useful and sensible approach (depending on the situation) it has a drawback. The extended part of the interface in the derived class is not available from the base class, so once you upcast you can't call the new methods:



You may not be upcasting in this case, so it may not bother you, but very often you'll get into a situation where you need to rediscover the exact type of the object so you can access the extended methods of that type.

downcasting & run-time type identification

Since you lose the specific type information via an *upcast* (moving up the inheritance hierarchy), it makes sense that to retrieve the type information – that is, to move back down the inheritance hierarchy – you use a *downcast*. However, you know an upcast is always safe: the base class cannot have a bigger interface than the derived class, therefore every message you send through the base class interface is guaranteed to be accepted. But with a downcast, you don't really know that a shape (for example) is actually a circle. It could instead be a triangle or square or some other type.



To solve this problem there must be some way to guarantee that a downcast is correct, so you won't accidentally cast to the wrong type and then send a message that the object can't accept. This would be quite unsafe.

In some languages (like C++) you must perform a special operation in order to get a type-safe downcast, but in Java *every cast* is checked! So even though it looks like you're just performing an ordinary parenthesized cast, at run time this cast is checked to ensure that it is in fact the type you think it is. If it isn't, you get a **ClassCastException**. This act of checking types at run time is called *run-time type identification* (RTTI). The following example demonstrates the behavior of RTTI:

```
//: RTTI.java
// Downcasting & Run-Time Type
// Identification (RTTI)
import java.util.*;

class Useful {
  public void f() {}
  public void g() {}
}

class MoreUseful extends Useful {
  public void f() {}
```

```
  public void g() {}
  public void u() {}
  public void v() {}
  public void w() {}
}

public class RTTI {
  public static void main(String args[]) {
    Useful x[] = {
      new Useful(),
      new MoreUseful()
    };
    x[0].f();
    x[1].g();
    // Compile-time: method not found in Useful:
    //! x[1].u();
    ((MoreUseful)x[1]).u(); // Downcast/RTTI
    ((MoreUseful)x[0]).u(); // Exception thrown
  }
} ///:~
```

As in the diagram, **MoreUseful** extends the interface of **Useful**. But since it's inherited, it can also be upcast to a **Useful**. You can see this happening in the initialization of the array **x** in **main( )**. Since both objects in the array are of class **Useful**, you can send the **f( )** and **g( )** methods to both, and if you try to call **u( )** (which only exists in **MoreUseful**) you'll get a compile-time error message.

If you want to access the extended interface of a **MoreUseful** object, you can try to downcast. If it's the right type, it will be successful. Otherwise, you'll get a **ClassCastException**. You don't have to write any special code for this exception, since it indicates a programmer error that could happen anywhere in a program.

There's more to RTTI than a simple cast. For example, there's a way to see what type you're dealing with *before* you try to downcast it. All of Chapter 11 is devoted to the study of different aspects of Java run-time type identification.

summary

Polymorphism means "different forms." In object-oriented programming, you have the same face (the common interface in the base class) and different forms using that face: the different versions of the dynamically-bound methods.

You've seen in this chapter that it's impossible to understand, or even create, an example of polymorphism without using data abstraction and inheritance. Polymorphism is a feature that cannot be viewed in isolation (like a **switch** statement, for example), but instead works only in concert, as part of a "big picture" of class relationships. People are often confused by other, non-object-oriented features of Java, like method overloading, which are sometimes presented as object-oriented. Don't be fooled: If it isn't late binding, it isn't polymorphism.

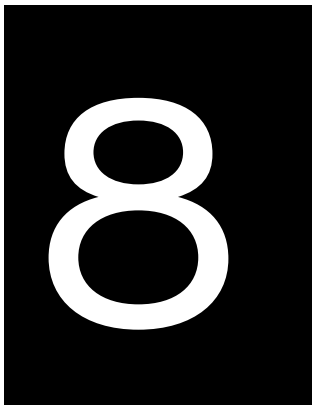
To use polymorphism, and thus object-oriented techniques, effectively in your programs you must expand your view of programming to include not just members and messages of an individual class, but also the commonality among classes and their relationships with each other. Although this requires significant effort, it's a worthy struggle, because the results are faster program development, better code organization, extensible programs, and easier code maintenance.

exercises

1. Create an inheritance hierarchy of **Rodent**: **Mouse**, **Gerbil**, **Hamster**, etc. In the base class, provide methods that are common to all **Rodent**s, and override these in the derived classes to perform different behaviors depending on the specific type of **Rodent**. Create an array of **Rodent**, fill it with different specific types of **Rodent**s, and call your base-class methods to see what happens.

2. Change exercise one so that **Rodent** is an **interface**.

3. Repair the problem in **WindError.java**.



8: holding your objects

[[[Chapter 8 directory:c08]]]

It's a fairly simple program that only ever has a fixed quantity of objects with known lifetimes.

In general your programs will always be creating new objects based on some criteria that will only be known at the time the program is running. In addition, you won't know until run-time the quantity or even the exact type of the objects you need. To solve the general programming problem, you need to create any number of objects, anytime, anywhere. So you can't rely on creating a named handle to hold each one of your objects:

```
MyObject myHandle;
```

since you'll never know how many of these things you'll actually need.

To solve this rather essential problem, Java has several ways to hold objects (or rather, handles to objects). The built-in type is the array, which has been discussed before and will get additional coverage in this chapter. Also, the Java utilities library has some *container classes* which provide more sophisticated ways to hold and even manipulate your objects; this will comprise the remainder of this chapter.

arrays

Most of the necessary introduction to arrays is in the last section of Chapter 4, which shows how you define and initialize an array. Holding objects is the focus of this chapter, and an array is just a way to hold objects. But there are a number of other ways to hold objects, so what makes an array special?

There are two issues that distinguish arrays from other types of containers: efficiency and type. The array is the most efficient way that Java provides to store and access objects (actually, object handles). The array is a simple linear sequence, which makes it very fast, but you pay for this speed: when you create an array, the size is fixed and cannot be changed for the lifetime of that array object. Remember that you can find out what the length of the array is using the read-only **length** member that's available in any array. Although this brings up a drawback: you can't find out how many elements are actually *in* the array, since **length** only tells you how many elements *can* be placed in the array.

You might suggest creating an array of a particular size and then, if you run out of space, creating a new one and moving all the handles from the old one to the new one. This is exactly what the **Vector** class does, which will be studied in the next section. However, because of the overhead of this size flexibility, a **Vector** is measurably less efficient than an array.

In C++, the **vector** class has another drawback over arrays in Java: the C++ **vector** doesn't do bounds checking, so you can run past the end (it's possible to *ask* how big the **vector** is, which is an improvement). In Java, you get bounds checking regardless of whether you're using an array or a container – you'll get a **RuntimeException** if you exceed the bounds. As you'll learn in Chapter 9, this type of exception indicates a programmer error and thus you don't need to check for it in your code. As an aside, the reason the C++ **vector** doesn't check bounds with every access is speed – in Java you have the constant overhead of bounds checking all the time for both arrays and containers.

The other generic container classes that will be studied in this chapter, **Vector**, **Stack** and **Hashtable**, all deal with objects as if they had no specific type. That is, they treat them as type **Object**, the root class of all classes in Java. This works fine from one standpoint: you only need to build one container, and any Java object will go into that container (except for primitives – a major reason that wrapper classes exist for primitives is so they can be placed in containers). This is the second place where an array is superior to the generic containers: when you create an array, you create it to hold a specific type. This means you get compile-time type checking to prevent you from putting the wrong type in, or mistaking the type that you're extracting. Of course, Java will prevent you from sending an inappropriate message to an object one way or another, either at compile-time or at run-time, so it's not as if it's riskier one way or another, it's just nicer if the compiler points it out to you, faster at run-time, and there's less likelihood that the end user will get surprised by an exception.

For both of the aforementioned reasons – efficiency and type checking – it's always worth trying to use an array if you can. However, when you're trying to solve a more general problem arrays can be too restrictive. After looking at an array example, the rest of this chapter will be devoted to the container classes provided by Java.

returning an array

Suppose you're writing a method and you don't just want to return one thing, but instead a whole bunch of things. Languages like C and C++ make this difficult because you can't just return an array, but only a pointer to an array. This introduces problems because it becomes messy to control the lifetime of the array, which easily leads to memory leaks.

Java takes a similar approach, but you just "return an array." Actually, of course, you're returning a handle to an array but with Java you never worry about responsibility for that array – it will be around as long as you need it, and the garbage collector will clean it up when you're done.

As an example, consider returning an array of **String**:

```
//: IceCream.java
// Returning arrays from functions
```

```
public class IceCream {
  static String flav[] = {
    "Chocolate", "Strawberry",
    "Vanilla Fudge Swirl", "Mint Chip",
    "Mocha Almond Fudge", "Rum Raisin",
    "Praline Cream", "Mud Pie"
  };
  static String[] flavorSet(int n) {
    // Force it to be positive & within bounds:
    n = Math.abs(n) % (flav.length + 1);
    String results[] = new String[n];
    int picks[] = new int[n];
    for(int i = 0; i < picks.length; i++)
      picks[i] = -1;
    for(int i = 0; i < picks.length; i++) {
      retry:
      while(true) {
        int t =
          (int)(Math.random() * flav.length);
        for(int j = 0; j < i; j++)
          if(picks[j] == t) continue retry;
        picks[i] = t;
        results[i] = flav[t];
        break;
      }
    }
    return results;
  }
  public static void main(String args[]) {
    for(int i = 0; i < 20; i++) {
      System.out.println(
        "flavorSet(" + i + ") = ");
      String fl[] = flavorSet(flav.length);
      for(int j = 0; j < fl.length; j++)
        System.out.println("\t" + fl[j]);
    }
  }
} ///:~
```

The method **flavorSet( )** creates an array of **String** called **results**. The size of this array is **n**, determined by the argument you pass into the method. Then it proceeds to randomly choose flavors from the array **flav** and place them into **results**, which it finally returns. Returning an array is just like returning any other object: it's a handle. It's not important that the array was created within **flavorSet( )**, or that the array was created anyplace else, for that matter. The garbage collector takes care of cleaning up the array when you're done with it, and the array will persist for as long as you need it.

As an aside, notice that when **flavorSet( )** randomly chooses flavors, it ensures that a random choice hasn't been picked before. This is performed in a seemingly infinite **while** loop that keeps making random choices until it finds one that's not already in the **picks** array (of course a **String** comparison could also have been performed to see if the random choice was already in the **results** array but **String** comparisons are very inefficient). If it's successful it adds the entry and **break**s out to go find the next one (**i** gets incremented). But if **t** is a number that's already in **picks**, then a labeled **continue** is used to jump back two levels which forces a new **t** to be selected. It's particularly convincing to watch this happen with a debugger.

main( ) prints out 20 full sets of flavors, so you can see that **flavorSet( )** chooses the flavors in a random order each time. It's easiest to see this if you redirect the output into a file. And while you're looking at the file, remember, you're not really hungry (You just *want* the ice cream, you don't *need* it).

arrays of primitives

There's one other thing you can accomplish with an array that you can't with a container class: container classes can only hold handles to objects. An array, however, can be created to hold primitives directly, as well as handles to objects. It *is* possible to use the "wrapper" classes such as **Integer**, **Double** etc. to place primitive values inside a container, but as you'll see later in this chapter in the **WordCount.java** example, the wrapper classes for primitives are only somewhat useful anyway. Whether you put primitives in arrays or wrap them in a class that's placed in a container is a question of efficiency: it's much more efficient to create and access an array of primitives than a container of wrapped primitives.

Of course, if you're using a primitive type and you need the flexibility of a container that automatically expands itself when more space is needed, the array won't work and you're forced to use a container of wrapped primitives. You might think that there should be a specialized type of **Vector** for each of the primitive data types, but Java doesn't provide this for you. Some sort of templatizing mechanism may someday provide a better way for Java to handle this problem[1].

containers

To summarize what we've seen so far: your first, most efficient choice to hold a group of objects should be an array, and you're forced into this choice if you want to hold a group of primitives. In the remainder of the chapter we'll look at the more general case, when you don't know at the time you're writing the program how many objects you're going to need, or if you need a more sophisticated way to store your objects. Java provides four types of *container classes* to solve this problem: **Vector**, **BitSet**, **Stack** and **Hashtable**. Although compared to other languages that provide containers this is a fairly meager supply, you can nonetheless solve a surprising number of problems using these tools.

Among their other characteristics – **Stack**, for example, implements a LIFO (last-in, first-out) sequence, and **Hashtable** is an *associative array* that lets you associate any object with any other object – the Java container classes will automatically resize themselves. Thus, you can put in any number of objects and you don't have to worry about how big to make the container while you're writing the program.

disadvantage: unknown type

The "disadvantage" to using the Java containers is that you lose type information when you put an object into a container. This happens because, when the container was written the programmer of that container had no idea what specific type you wanted to put in the container, and making the container hold only your type would prevent it from being a general-purpose tool. So instead, the container holds handles to objects of type **Object**, which is of course every object in Java since it's the root of all the classes (of course, this doesn't include primitive types, since they aren't inherited from anything). This is a great solution, except for a couple of things:

1. Since the type information is thrown away when you put an object handle into a container, *any* type of object can be put into your container, even if you only mean it to hold, say, cats. Someone could just as easily put a dog into the container.

2. Since the type information is lost and the only thing the container knows it is holding is an **Object** handle, you must remember and perform a cast to the correct type before you use it.

On the up side, Java won't let you *misuse* the objects that you put into a container. If you throw a dog into a container of cats, then go through and try to treat everything in the container as a cat, you'll get

---

[1] This is one of the places where C++ is distinctly superior to Java, since C++ supports *parameterized types* with the **template** keyword.

an exception when you get to the dog. In the same vein, if you try to cast the dog handle that you pull out of the cat container into a cat, you'll get an exception at run-time.

Here's an example:

```
//: CatsAndDogs.java
// Simple container example (Vector)
import java.util.*;

class Cat {
  private int catNumber;
  Cat(int i) {
    catNumber = i;
  }
  void print() {
    System.out.println("Cat #" + catNumber);
  }
}

class Dog {
  private int dogNumber;
  Dog(int i) {
    dogNumber = i;
  }
  void print() {
    System.out.println("Dog #" + dogNumber);
  }
}

public class CatsAndDogs {
  public static void main(String args[]) {
    Vector cats = new Vector();
    for(int i = 0; i < 7; i++)
      cats.addElement(new Cat(i));
    // Not a problem to add a dog to cats:
    cats.addElement(new Dog(7));
    for(int i = 0; i < cats.size(); i++)
      ((Cat)cats.elementAt(i)).print();
    // Dog is only detected at run-time
  }
} ///:~
```

You can see that using a **Vector** is straightforward: create one, put objects in using **addElement( )** and later get them out with **elementAt( )** (notice that **Vector** has a method **size( )** to let you know how many elements have been added so you don't inadvertently run off the end and cause an exception).

The classes **Cat** and **Dog** are distinct – they have nothing in common other than they are objects (if you don't explicitly say what class you're inheriting from, you automatically inherit from **Object**). The **Vector** class, which comes from **java.util**, holds **Object**s, so not only can I put **Cat** objects into this container using the **Vector** method **addElement( )**, but I can also add **Dog** objects without complaint at either compile-time or run-time. When I go to fetch out what I think are **Cat** objects using the **Vector** method **elementAt( )**, I get back a handle to an **Object** that I must cast to a **Cat**. Then I have to surround the entire expression with parentheses to force the evaluation of the cast before calling the **print( )** method for **Cat**, otherwise I'll get a syntax error. Then, at run-time, when I try to cast the **Dog** object to a **Cat**, I'll get an exception.

This is more than just an annoyance. It's something that can create some difficult-to-find bugs. If one part (or several parts) of a program inserts objects into a container, and you only discover in a separate part of the program, through an exception, that a bad object was placed in the container, then you must find out where the bad insert occurred by code inspection, which is about the worst debugging

tool we have. On the upside, it's very convenient to start with some standardized container classes for programming, despite the scarcity and awkwardness.

sometimes it works right anyway

It turns out that in some cases things seem to work correctly without casting back to your original type. The first case is quite special: the **String** class has some extra help from the compiler to make it work smoothly. Whenever the compiler expects a **String** object and it hasn't got one, it will automatically call the **toString( )** method that's defined in **Object** and may be redefined by any Java class. This method produces the desired **String** object, which is then used wherever it was wanted.

Thus, all you need to do to make objects of your class magically print out is to redefine the **toString( )** method, as shown in the following example:

```
//: WorksAnyway.java
// In special cases, things just seem
// to work correctly.
import java.util.*;

class Mouse {
  private int mouseNumber;
  Mouse(int i) {
    mouseNumber = i;
  }
  // Magic method:
  public String toString() {
    return "This is Mouse #" + mouseNumber;
  }
  void print(String msg) {
    if(msg != null) System.out.println(msg);
    System.out.println(
      "Mouse number " + mouseNumber);
  }
}

class MouseTrap {
  static void caughtYa(Object m) {
    Mouse mouse = (Mouse)m; // cast from Object
    mouse.print("Caught one!");
  }
}

public class WorksAnyway {
  public static void main(String args[]) {
    Vector mice = new Vector();
    for(int i = 0; i < 3; i++)
      mice.addElement(new Mouse(i));
    for(int i = 0; i < mice.size(); i++) {
      System.out.println(
        "Free mouse: " + mice.elementAt(i));
      MouseTrap.caughtYa(mice.elementAt(i));
    }
  }
} ///:~
```

You can see the redefinition of **toString( )** in **Mouse**. In the second **for** loop in **main( )** you find the statement:

```
System.out.println("Free mouse: " + mice.elementAt(i));
```

After the '**+**' sign the compiler is expecting to see a **String** object. **elementAt( )** produces an **Object**, so to get the desired **String** the compiler implicitly calls **toString( )**. Unfortunately, you can only work this kind of magic with **String**; it isn't available for any other type.

A second approach to hiding the cast has been placed inside **Mousetrap**: the **caughtYa( )** method accepts, not a **Mouse**, but an **Object** which it then casts to a **Mouse**. This is quite presumptuous, of course, since by accepting an **Object** anything could be passed to the method. However, if the cast is incorrect – if you passed the wrong type – you'll get an exception at run-time. Still not as good as compile-time checking but still robust. Notice that in the use of this method:

```
MouseTrap.caughtYa(mice.elementAt(i));
```

no cast is necessary.

Making a type-conscious Vector

You may not want to give up on this issue just yet. A more ironclad solution is to create a new class using the **Vector**, such that it will only accept your type and only produce your type:

```
//: GopherVector.java
// A type-conscious Vector
import java.util.*;

class Gopher {
  private int gopherNumber;
  Gopher(int i) {
    gopherNumber = i;
  }
  void print(String msg) {
    if(msg != null) System.out.println(msg);
    System.out.println(
      "Gopher number " + gopherNumber);
  }
}

class GopherTrap {
  static void caughtYa(Gopher m) {
    m.print("Caught one!");
  }
}

class GopherVector {
  private Vector v = new Vector();
  public void addElement(Gopher m) {
    v.addElement(m);
  }
  public Gopher elementAt(int index) {
    return (Gopher)v.elementAt(index);
  }
  public int size() { return v.size(); }
  public static void main(String args[]) {
    GopherVector gophers = new GopherVector();
    for(int i = 0; i < 3; i++)
      gophers.addElement(new Gopher(i));
    for(int i = 0; i < gophers.size(); i++)
      GopherTrap.caughtYa(gophers.elementAt(i));
  }
} ///:~
```

This is similar to the previous example, except that the new **GopherVector** class has a **private** member of type **Vector** (inheriting from **Vector** tends to be frustrating, for reasons you'll see later), and

methods just like **Vector**. However, it doesn't accept and produce generic **Object**s, only **Gopher** objects.

Because a **GopherVector** will only accept a **Gopher**, if you were to say:

```
gophers.addElement(new Pigeon());
```

you would get an error message *at compile time*. So this approach, while more tedious from a coding standpoint, will tell you immediately if you're using a type improperly.

Notice that no cast is necessary when using **elementAt( )** – it's always a **Gopher**.

parameterized types

This kind of problem isn't isolated – there are numerous cases where you need to create new types based on other types, and where it is very useful to have specific type information at compile-time. This is the concept of a *parameterized type*. In C++ this is directly supported by the language in the form of *templates*. At one point, Java had reserved the keyword **generic** to someday support parameterized types, but it's uncertain if this will ever occur.

enumerators (iterators)

In any container class, you must have a way to put things in and a way to get things out. After all, that's the primary job of a container – to hold things. In the **Vector**, **addElement( )** is the way you insert objects, and **elementAt( )** is *one* way to get things out. **Vector** is quite flexible – you can select anything at any time, and select multiple elements at once using different indexes.

But if you want to start thinking at a higher level, there's a drawback: you have to know the exact type to use it. This may not seem bad at first, but what if you start out using a **Vector**, and later on in your program you decide, for reasons of efficiency, that you want to change to a **List** (not yet part of the standard Java library). Or you'd like to write a piece of code that doesn't know or care what type of container it's working with.

The concept of an *iterator* can be used to achieve this next level of abstraction. This is an object whose job is to move through a sequence of objects and select each object in that sequence, without knowing or caring about the underlying structure of that sequence. In addition, an iterator is usually what's called a "light-weight" object; that is, one that's cheap to create. For that reason, you'll often find seemingly strange constraints for iterators, such as they can only move in one direction.

The Java **Enumeration**[2] is an example of an iterator with these kinds of constraints – there's not much you can do with one except:

1. Ask a container to hand you an **Enumeration** using a method called **elements( )**. This **Enumeration** will be selecting the first element in the sequence)
2. Get the next object in the sequence with **nextElement( )**
3. See if there *are* any more objects in the sequence with **hasMoreElements( )**

That's all. It's a very simple implementation of an iterator, but still powerful. To see how it works, let's revisit the **CatsAndDogs.java** program from earlier in the chapter. In the original version, the method **elementAt( )** was used to select each element, but in the following modified version an enumeration is used:

```
//: CatsAndDogs2.java
// Simple container with Enumeration
```

[2] I suspect that since the term *iterator* is so common in C++ and elsewhere in OOP, the Java team used a strange name just to be different, which will no doubt simply cause confusion and nothing else.

```
import java.util.*;

class Cat2 {
  private int catNumber;
  Cat2(int i) {
    catNumber = i;
  }
  void print() {
    System.out.println("Cat number " +catNumber);
  }
}

class Dog2 {
  private int dogNumber;
  Dog2(int i) {
    dogNumber = i;
  }
  void print() {
    System.out.println("Dog number " +dogNumber);
  }
}

public class CatsAndDogs2 {
  public static void main(String args[]) {
    Vector cats = new Vector();
    for(int i = 0; i < 7; i++)
      cats.addElement(new Cat2(i));
    // Not a problem to add a dog to cats:
    cats.addElement(new Dog2(7));
    Enumeration e = cats.elements();
    while(e.hasMoreElements())
      ((Cat2)e.nextElement()).print();
    // Dog is only detected at run-time
  }
} ///:~
```

You can see that the only change is in the last few lines. Instead of:

```
    for(int i = 0; i < cats.size(); i++)
      ((Cat)cats.elementAt(i)).print();
```

an **Enumeration** is used to step through the sequence:

```
while(e.hasMoreElements())
      ((Cat2)e.nextElement()).print();
```

With the **Enumeration**, you don't have to worry about the number of elements in the container. That's taken care of for you by **hasMoreElements( )** and **nextElement( )**.

As another example, consider the creation of a general-purpose printing method:

```
//: HamsterMaze.java
// Using an Enumeration
import java.util.*;

class Hamster {
  private int hamsterNumber;
  Hamster(int i) {
    hamsterNumber = i;
  }
  public String toString() {
```

```
    return new String(
      "This is Hamster #" + hamsterNumber);
  }
}

class Printer {
  static void printAll(Enumeration e) {
    while(e.hasMoreElements())
      System.out.println(
        e.nextElement().toString());
  }
}

public class HamsterMaze {
  public static void main(String args[]) {
    Vector v = new Vector();
    for(int i = 0; i < 3; i++)
      v.addElement(new Hamster(i));
    Printer.printAll(v.elements());
  }
} ///:~
```

Look closely at the printing method:

```
static void printAll(Enumeration e) {
  while(e.hasMoreElements())
    System.out.println(
      e.nextElement().toString());
}
```

Notice that there's no information about the sequence that's being dealt with. All you have is an **Enumeration**, and that's all you need to know about the sequence: that you can get the next object, and that you can know when you're at the end. This idea of taking a collection of objects, and passing through and performing an operation on each one is very powerful and will be seen again and again throughout this book.

This particular example is even more generic, since it uses the ubiquitous **toString( )** method (ubiquitous only because it's part of the **Object** class). An alternative way to call print (although probably slightly less efficient, if you could even notice the difference) is:

```
System.out.println("" + e.nextElement());
```

which uses the "automatic conversion to **String**" that's wired into Java. When the compiler sees a **String**, followed by a '**+**', it expects another **String** to follow and calls **toString( )** automatically. You can also perform a cast, which has the effect of calling **toString( )**:

```
System.out.println((String)e.nextElement());
```

In general, however, you'll want to do something more than call **Object** methods, so you'll run up against the type-casting issue again. That is, you'll have to assume that you've gotten an **Enumeration** to a sequence of the particular type you're interested in, and cast the resulting objects to that (getting a run-time exception if you're wrong).

types of containers

The standard Java library comes with the bare minimum set of container classes, but they're probably enough to get by with for most of your programming projects.

Vector

The **Vector** is quite simple to use, as you've seen so far. Although most of the time you'll just use **addElement( )** to insert objects, **elementAt( )** to get them out one at a time and **elements( )** to get an **Enumeration** to the sequence, there's also a set of other methods that can be useful. As usual with the Java libraries, we won't use or talk about them all here, but be sure to look them up in the electronic documentation to get a feel for what they can do.

BitSet

A **BitSet** is really a **Vector** of bits, and is used if you want to efficiently store a whole lot of on-off information. It's efficient only from the standpoint of size; if you're looking for efficient access it is slightly slower than using an array of some native type.

In addition, the minimum size of the **BitSet** is that of a long: 64 bits. This implies that if you're storing anything smaller, like 8 bits, a **BitSet** will be wasteful so you're better off creating your own class to hold your flags.

In a normal **Vector**, the container will expand as you add more elements. The **BitSet** does this as well – sort of. That is, sometimes it works and sometimes it doesn't, which makes it appear that the Java version 1 implementation of **BitSet** is just badly done (it is fixed in Java 1.1). The following example shows how the **BitSet** works and demonstrates the version 1 bug:

```
//: Bits.java
// Demonstration of BitSet
import java.util.*;

public class Bits {
  public static void main(String args[]) {
    Random rand = new Random();
    // Take the LSB of nextInt():
    byte bt = (byte)rand.nextInt();
    BitSet bb = new BitSet();
    for(int i = 7; i >=0; i--)
      if(((1 << i) &  bt) != 0)
        bb.set(i);
      else
        bb.clear(i);
    System.out.println("byte value: " + bt);
    printBitSet(bb);

    short st = (short)rand.nextInt();
    BitSet bs = new BitSet();
    for(int i = 15; i >=0; i--)
      if(((1 << i) &  st) != 0)
        bs.set(i);
      else
        bs.clear(i);
    System.out.println("short value: " + st);
    printBitSet(bs);

    int it = rand.nextInt();
    BitSet bi = new BitSet();
    for(int i = 31; i >=0; i--)
      if(((1 << i) &  it) != 0)
        bi.set(i);
      else
        bi.clear(i);
    System.out.println("int value: " + it);
    printBitSet(bi);
```

```
    // Test bitsets >= 64 bits:
    BitSet b127 = new BitSet();
    b127.set(127);
    System.out.println("set bit 127: " + b127);
    BitSet b255 = new BitSet(65);
    b255.set(255);
    System.out.println("set bit 255: " + b255);
    BitSet b1023 = new BitSet(512);
// Without the following, an exception is thrown
// in the Java 1 implementation of BitSet:
//    b1023.set(1023);
    b1023.set(1024);
    System.out.println("set bit 1023: " + b1023);
  }
  static void printBitSet(BitSet b) {
    System.out.println("bits: " + b);
    String bbits = new String();
    for(int j = 0; j < b.size() ; j++)
      bbits += (b.get(j) ? "1" : "0");
    System.out.println("bit pattern: " + bbits);
  }
} ///:~
```

The random number generator is used to create a random **byte**, **short** and **int**, and each one is transformed into a corresponding bit pattern in a **BitSet**. All this works fine because a **BitSet** is 64 bits, so none of these cause it to increase in size. But in Java 1, when the **BitSet** is greater than 64 bits, though, some strange behavior occurs. If you set a bit that's just one greater than the **BitSet**'s currently-allocated storage, it will expand nicely. But if you try to set bits at higher locations than that without first just touching the boundary, you'll get an exception, since the **BitSet** won't expand properly in Java 1. The example shows a **BitSet** of 512 bits being created. The constructor allocates storage for twice that number of bits. Then if you try to set bit 1024 or greater without first setting bit 1023, you'll throw an exception in Java 1.0. Fortunately, this is fixed in Java 1.1 but you'll need to avoid using the BitSet if you must write code for Java 1.

Stack

A **Stack** is sometimes referred to as a "last-in, first-out" (LIFO) container. That is, whatever you "push" on the **Stack** last is the first item you can "pop" out. Like all the other containers in Java, what you push and pop are **Object**s, so you must cast what you pop.

What's rather odd is that rather than using a **Vector** as a building block to create a **Stack**, **Stack** is inherited from **Vector**. This means it has all the characteristics and behaviors of a **Vector** *plus* some extra **Stack** behaviors. It's difficult to know whether the designers explicitly decided that this was an especially useful way to do things, or whether it was just a naïve design.

Here's a simple demonstration of **Stack** that reads each line from a file and pushes it as a **String**:

```
//: Stacks.java
// Demonstration of Stack Class
import java.util.*;
import java.io.*;

public class Stacks {
  public static void main(String args[])
      throws IOException {
    BufferedReader in =
      new BufferedReader(
        new FileReader(args[0]));
    Stack stk = new Stack();
    String s;
    while((s = in.readLine()) != null)
```

```
      stk.push("\n" + s);
    System.out.println("stk = " + stk);
    // Treating a stack as a Vector:
    Vector v = stk; // No cast!
    System.out.println(
      "element 5 = " + v.elementAt(5));
    System.out.println("popping elements:");
    while(!stk.empty())
      System.out.print(stk.pop());
  }
} ///:~
```

This is the first time in the book that a file is read, and although it will be described thoroughly in Chapter 10 a brief introduction is in order here. To use the input/output facilities, you must **import java.io**. To read a line at a time from a stream, you create a **BufferedReader** object, and to open a file you must create a **FileReader** object (note these are Java 1.1 classes). The line that creates the **in** object does this all at once. Once that occurs, you can read each line by calling **readLine( )** which returns the line as a **String** object, with the newline character(s) stripped off. When **readLine( )** returns **null**, you're at the end of the file.

You'll notice that **main( )** has something new between the argument list and the opening curly brace: **throws IOException**. Although all the aspects of exception handling won't be revealed fully until Chapter 9, exceptions are so essential to Java programming that in order to call many of the methods in a Java library you must be aware of what exceptions are "thrown" by a particular method. In fact, the compiler keeps track of this and if you don't deal with the exception it will give you an error message. The laziest way to deal with an exception is to simply ignore it, but then you must officially state that your method is not going to do anything about the exception and let it pass on through. That's what the **throws** keyword means: these are the exceptions that may emerge when you call this method. By allowing the exceptions to emerge from **main( )**, you're saying "I'll let the system handle these."

As you'll see in Chapter 9, there are many different types of exceptions that can come out of the standard Java library methods (and you can create your own). In the case of input/output, the exception that can occur if an operation has any trouble is **IOException**. In this case the exception is thrown if the program cannot open the file that you've handed it on the command line.

Each line in the file is read and comes back as a **String** which is inserted into the **Stack** with **push( )**. Just to make a point, the **Stack** object is assigned directly to a **Vector** object, without a cast. This is possible because, by virtue of inheritance, a **Stack** *is* a **Vector**. Thus all operations that can be performed on a **Vector** can also be performed on a **Stack**, such as **elementAt( )**.

Hashtable

A **Vector** allows you to select from a sequence of objects using a number, so in a sense it associates numbers to objects. But what if you'd like to select from a sequence of objects using some other criterion? A **Stack** is an example: its selection criterion is "the last thing pushed on the stack." A very powerful twist on this idea of "selecting from a sequence" is alternately termed a *map*, a *dictionary* or an *associative array*. Conceptually, it seems like a vector, but instead of looking up objects using a number, you look them up using *another object*! This is very often a key process in a program.

The concept shows up in Java as the **abstract** class **Dictionary**. The interface for this class is straightforward: **size( )** tells you how many elements are within, **isEmpty( )** is **true** if there are no elements, **put(Object key, Object value)** adds a value (the thing you'll be wanting) and associates it with a key (the thing you'll be looking it up with). **get(Object key)** produces the value given the corresponding key, and **remove(Object key)** removes the key-value pair from the list. There are enumerations: **keys( )** produces an **Enumeration** of the keys, and **elements( )** produces an **Enumeration** of all the values. That's all there is to a **Dictionary**.

A **Dictionary** isn't terribly difficult to implement. Here's a simple approach, which uses two **Vector**s, one for keys and one for values:

```
//: AssocArray.java
// Simple version of a Dictionary
import java.util.*;

public class AssocArray extends Dictionary {
  private Vector keys = new Vector();
  private Vector values = new Vector();
  public int size() { return keys.size(); }
  public boolean isEmpty() {
    return keys.isEmpty();
  }
  public Object put(Object key, Object value) {
    keys.addElement(key);
    values.addElement(value);
    return key;
  }
  public Object get(Object key) {
    int index = keys.indexOf(key);
    // indexOf() Returns -1 if key not found:
    if(index == -1) return null;
    return values.elementAt(index);
  }
  public Object remove(Object key) {
    int index = keys.indexOf(key);
    if(index == -1) return null;
    keys.removeElementAt(index);
    Object returnval = values.elementAt(index);
    values.removeElementAt(index);
    return returnval;
  }
  public Enumeration keys() {
    return keys.elements();
  }
  public Enumeration elements() {
    return values.elements();
  }
  // Test it:
  public static void main(String args[]) {
    AssocArray aa = new AssocArray();
    for(char c = 'a'; c <= 'z'; c++)
      aa.put(String.valueOf(c),
             String.valueOf(c)
             .toUpperCase());
    char[] ca = { 'a', 'e', 'i', 'o', 'u' };
    for(int i = 0; i < ca.length; i++)
      System.out.println("Uppercase: " +
             aa.get(String.valueOf(ca[i])));
  }
} ///:~
```

The first thing you see in the definition of **AssocArray** is that it **extends Dictionary**. This means that **AssocArray** *is a type of* **Dictionary**, so you can make the same requests of it that you can a **Dictionary**. If you make your own **Dictionary**, as is done here, all you have to do is fill in all the methods that are in **Dictionary** (and you *must* override all the methods because all of them – with the exception of the constructor – are abstract).

The **Vector**s **keys** and **values** are linked by a common index number. That is, if I call **put( )** with a key of "roof" and a value of "blue" (assuming I'm associating the various parts of a house with the colors they are to be painted) and there are already 100 elements in the **AssocArray**, then "roof" will be the 101 element of **keys** and "blue" will be the 101 element of **values**. And if you look at **get( )**, when you

pass “roof” in as the key, it produces the index number with **keys.indexOf( )**, and then uses that index number to produce the value in the associated **values** vector.

The test in **main( )** is very simple; it’s just a map of lowercase characters to uppercase characters, which could obviously be done in a number of more efficient ways. But it shows that **AssocArray** is functional.

The standard Java library contains only one embodiment of a **Dictionary**, which is a **Hashtable**. Java’s **Hashtable** has the same basic interface as **AssocArray** (since they both inherit **Dictionary**), but it differs in one distinct way: efficiency. If you look at what must be done for a **get( )**, it seems pretty slow to search through a **Vector** for the key. This is where **Hashtable** speeds things up: instead of the tedious linear search for the key, it uses a special value called a *hash code*. The hash code is a way to take some information in the object in question and turn it into a “relatively unique” **int** for that object. All objects have a hash code, and **hashCode( )** is a method in the root class **Object**. A **Hashtable** takes the **hashCode( )** of the object and uses it to quickly hunt for the key. This results in a dramatic performance improvement. The *way* that a **Hashtable** works is beyond the scope of this book[3] – all you need to know is that **Hashtable** is a fast **Dictionary**, and that a **Dictionary** is a very useful tool.

As an example of the use of a **Hashtable**, consider a program to check the randomness of Java’s **Math.random( )** method. Ideally, it would produce a perfect distribution of random numbers, but to test this we need to generate a bunch of random numbers and count the ones that fall in the various ranges. A **Hashtable** is perfect for this, since it associates objects with objects (in this case, the values produced by **Math.random( )** with the number of times those values appear:

```
//: Statistics.java
// Simple demonstration of Hashtable
import java.util.*;

class Counter {
  int i = 1;
  public String toString() {
    return Integer.toString(i) + "\n";
  }
}

class Statistics {
  public static void main(String args[]) {
    Hashtable ht = new Hashtable();
    for(int i = 0; i < 10000; i++) {
      // Produce a number between 0 and 20:
      Integer r =
        new Integer((int)(Math.random() * 20));
      if(ht.containsKey(r))
        ((Counter)ht.get(r)).i++;
      else
        ht.put(r, new Counter());
    }
    System.out.println(ht);
  }
} ///:~
```

In **main( )**, each time a random number is generated it is wrapped inside an **Integer** object so that handle can be used with the **Hashtable** (you can’t use a primitive with a container, only an object handle). The **containsKey( )** method checks to see if this key is already in the container (that is, has the number been found already?). If so, the **get( )** methods gets the associated value for the key,

[3] The best reference I know of is *Practical Algorithms for Programmers*, by Andrew Binstock and John Rex, Addison-Wesley 1995.

which in this case is a **Counter** object. The value **i** inside the counter is then incremented to indicate one more of this particular random number has been found.

If the key has not been found yet, the method **put( )** will place a new key-value pair into the **Hashtable**. Since **Counter** automatically initializes its variable **i** to one when it's created, it indicates the first occurrence of this particular random number.

To display the **Hashtable**, it is simply printed out. The **Hashtable toString( )** method moves through all the key-value pairs and calls the **toString( )** for each one. The **Integer toString( )** is pre-defined, and you can see the **toString( )** for **Counter**. The output from one run is:

```
{19=526
, 18=533
, 17=460
, 16=513
, 15=521
, 14=495
, 13=512
, 12=483
, 11=488
, 10=487
, 9=514
, 8=523
, 7=497
, 6=487
, 5=480
, 4=489
, 3=509
, 2=503
, 1=475
, 0=505
}
```

So even considering noise from rounding off, the numbers shown here suggest that you should be reasonably suspicious of the randomness of **Math.random( )**.

You may wonder at the necessity of the class **Counter** which seems like it doesn't even have the functionality of the wrapper class **Integer**. Why not use **int** or **Integer**? Well, you can't use an **int** because all the containers can only hold **Object** handles. After seeing containers the wrapper classes might begin to make a little more sense to you, since you can't put any of the primitive types in containers. However, the only thing you *can* do with the Java wrappers is to (1) initialize them to a particular value and (2) read that value. That is, there's no way to change a value once a wrapper object has been created. This makes the **Integer** wrapper immediately useless to solve our problem, and so we're forced to create a new class that does satisfy the need.

Properties: a type of Hashtable

In the very first example in the book, a type of **Hashtable** was used called **Properties**. In that example, the line:

```
System.getProperties().list(System.out);
```

called the **static** method **getProperties( )** to get a special **Properties** object that described the system characteristics. The method **list( )** is a method of **Properties** that sends the contents to any stream output that you choose. In addition there's a **save( )** method to allow you to write your property list to a file in a way that it can be retrieved later with the **load( )** method.

Although the **Properties** class is inherited from **Hashtable**, it also *contains* a second **Hashtable** that acts to hold the list of "default" properties. So if a property isn't found in the primary list, the defaults will be searched.

The **Properties** class is also available for use in your programs.

enumerators revisited

Despite the shockingly small number of containers provided by the standard Java library, we can now demonstrate the true power of the **Enumeration**: the ability to separate the operation of traversing a sequence from the underlying structure of that sequence. In the following example, the class **PrintData** uses an **Enumeration** to move through a sequence and call two methods that exist in every **Object**: **getClass( )** and **toString( )**. The two different types of containers are created, a **Vector** and a **Hashtable**, and they are filled with various types of objects (as you'll see, it's not important that you understand what these objects are for in order to understand the example). Then the genericity of **PrintData** is demonstrated by extracting an **Enumeration** from each container and passing it in to **PrintData.print( )**:

```
//: Enumerators2.java
// Revisiting Enumerations
import java.util.*;
import java.io.*;
import java.net.*;

class PrintData {
  static void print(String s, Enumeration e) {
    System.out.println(s);
    while(e.hasMoreElements()) {
      Object o = e.nextElement();
      System.out.println(o.getClass().getName() +
        ": " + o.toString());
    }
  }
}

class Enumerators2 {
  public static void main(String args[]) {
    Vector v = new Vector();
    v.addElement(new File("."));
    v.addElement(new Vector());
    v.addElement(new PrintData());
    v.addElement(new Enumerators2());
    v.addElement(new Date());

    Hashtable h = new Hashtable();
    h.put(new Integer(1), new Thread());
    h.put(new Integer(2),
          new ProtocolException()
          .getClass().getSuperclass());
    h.put(new Integer(3), v.clone());

    PrintData.print("Vector", v.elements());
    PrintData.print("Hashtable", h.elements());
  }
} ///:~
```

I've tossed just about everything into these containers, including some things you haven't seen yet, like a **Class** object that represents a class itself. You can see this in the creation of the **ProtocolException**: **getClass( )** produces the **Class** object that describes the **ProtocolException**, and **getSuperclass( )** produces the **Class** object that describes its base class, so you can tell at run time who it was inherited from.

The last **put( )** into the **Hashtable** actually inserts a **clone( )** of the **Vector** created at the beginning of **main( )**. Only the **Vector**, and not the objects it contains, is duplicated by the **clone( )** method. Since **clone( )** is a method of **Object**, many classes are cloneable (although what that means may be different for each class).

Notice that the only thing the objects in these containers have in common is that they are of class **Object**, so that's what **PrintData.print( )** takes advantage of. It's more likely that in your problem, you'll have to make an assumption that some specific type is being held in the container that your **Enumeration** is walking through. For example, you might assume that everything in the container is a **Shape** with a **draw( )** method – then you'll have to downcast from the **Object** that **Enumeration.nextElement**() returns to produce a **Shape**.

sorting

One of the things that's missing in the Java libraries is algorithmic operations, even simple sorting. So it makes sense to create a **Vector** that sorts itself using the classic Quicksort (the algorithm will not be explained here – for details, see *Practical Algorithms for Programmers*, by Binstock & Rex, Addison-Wesley 1995). Here's one that works with **String** objects:

```
//: StrSortVector.java
// You can tell this Vector of Strings
// to sort itself.
package c08;
import java.util.*;

public class StrSortVector extends Vector {
  public void sort() {
    quickSort(0, size() - 1);
  }
  private void quickSort(int left, int right) {
    if(right > left) {
      String s1 = (String)elementAt(right);
      int i = left - 1;
      int j = right;
      while(true) {
        while(((String)elementAt(++i))
              .toLowerCase()
              .compareTo(
                s1.toLowerCase()) < 0)
          ;
        while(j > 0)
          if(((String)elementAt(--j))
             .toLowerCase()
             .compareTo(
               s1.toLowerCase()) <= 0)
            break; // out of while
        if(i >= j) break;
        swap(i, j);
      }
      swap(i , right);
      quickSort(left, i-1);
      quickSort(i+1, right);
    }
  }
  private void swap(int loc1, int loc2) {
    Object tmp = elementAt(loc1);
    setElementAt(elementAt(loc2), loc1);
    setElementAt(tmp, loc2);
  }
  // Test it:
  public static void main(String args[]) {
    StrSortVector sv = new StrSortVector();
    sv.addElement("d");
```

```
    sv.addElement("A");
    sv.addElement("C");
    sv.addElement("c");
    sv.addElement("b");
    sv.addElement("B");
    sv.addElement("D");
    sv.addElement("a");
    sv.sort();
    Enumeration e = sv.elements();
    while(e.hasMoreElements())
      System.out.println((String)e.nextElement());
  }
} ///:~
```

The sorting algorithm forces the strings to lower case, so that the capital **A**'s end up next to the small **a**'s, and not in some entirely different place. This example shows, however, a slight deficiency in this approach, since the test code above put the uppercase and lowercase single letters of the same letter in the order that they appear: A a b B c C d D. This is not usually much of a problem because the entire string is used to perform the sort so you generally don't notice it.

Inheritance (**extends**) is used here to create a new type of **Vector** – that is, **StrSortVector** *is a* **Vector** with some added functionality. The use of inheritance here is powerful but it presents problems. It turns out that some methods are **final** (described in Chapter 7) so you cannot redefine them. These include **addElement( )** and **elementAt( )**, which are precisely the ones we'd like to change so they only accept and produce **String** objects. No luck there.

On the other hand, consider composition: the placing of an object *inside* a new class. Rather than rewrite the above code to accomplish this, we can simply use a **StrSortVector** inside the new class:

```
//: StrSortVector2.java
// Automatically sorted Vector that only
// accepts and produces Strings
package c08;
import java.util.*;

public class StrSortVector2 {
  private StrSortVector v = new StrSortVector();
  private boolean sorted = false;
  public void addElement(String s) {
    v.addElement(s);
    sorted = false;
  }
  public String elementAt(int index) {
    if(!sorted) {
      v.sort();
      sorted = true;
    }
    return (String)v.elementAt(index);
  }
  public Enumeration elements() {
    if(!sorted) {
      v.sort();
      sorted = true;
    }
    return v.elements();
  }
  // Test it:
  public static void main(String args[]) {
    StrSortVector2 sv = new StrSortVector2();
    sv.addElement("d");
    sv.addElement("A");
```

```
    sv.addElement("C");
    sv.addElement("c");
    sv.addElement("b");
    sv.addElement("B");
    sv.addElement("D");
    sv.addElement("a");
    Enumeration e = sv.elements();
    while(e.hasMoreElements())
      System.out.println((String)e.nextElement());
  }
} ///:~
```

This quickly reuses the code from **StrSortVector** to create the desired functionality. However, all the **public** methods from **StrSortVector** and **Vector** do *not* appear in **StrSortVector2** – its methods are only the ones that are explicitly defined. So you can either make a definition for each one, or periodically go back and adjust it when you need new ones until the class design settles down.

The advantage to this approach is that it will only take **String** objects and produce **String** objects, and the checking happens at compile time instead of run time. Of course, that's only true for **addElement( )** and **elementAt( )**; **elements( )** still produces an **Enumeration** which is untyped at compile time. Type checking for the **Enumeration** and in **StrSortVector** still happens, of course, it just happens at run-time, by throwing exceptions if you do something wrong. It's a trade-off: do you find out about something *for sure* at compile time, or instead *probably* at run-time? (That is, probably while you're testing the code and probably not when the program user tries something you didn't test for). Given the choices and the hassle, it's easier to use inheritance and just grit your teeth while casting – again, if parameterized types are ever add to Java they solve this problem.

You can see there's a flag called **sorted** in this class. You could sort the vector every time **addElement( )** is called, and constantly keep it in a sorted state. But usually people add a lot of elements to a **Vector** before beginning to read it. So sorting after every **addElement( )** would be less efficient than waiting until someone wants to read the vector, and then sorting it, which is what is done here. The technique of delaying a process until it is absolutely necessary is called *lazy evaluation*.

the Generic Collection Library

You've seen in this chapter that the standard Java library has some fairly useful containers, but far from a complete set. In addition, algorithms like sorting are not supported at all. One of the strengths of C++ is its libraries, in particular the *Standard Template Library* (STL) which provides a fairly full set of containers as well as many algorithms like sorting and searching that work with those containers. Based on this model, the ObjectSpace company was inspired to create the *Generic Collection Library for Java* (formerly called the *Java Generic Library*, but the abbreviation JGL is still used – the old name infringed on Sun's copyright), which follows the design of the STL as much as possible (given the differences between the two languages) and seems to fulfill many if not all of the needs for a container library, or as far as one could go in this direction without C++'s template mechanism. The JGL includes linked lists, sets, queues, maps, stacks, sequences, and iterators that are far more functional than **Enumeration**, along with a full set of algorithms like searching and sorting. ObjectSpace has also made, in many cases, more intelligent design decisions than the Javasoft library designers. For example, the methods in the JGL containers are *not* final so it's easy to inherit and override those methods.

The JGL has been included in many vendors' Java distributions and it seems likely that it will eventually become part of the Java standard library. ObjectSpace has made the JGL freely available for all uses, including commercial use, at www.ObjectSpace.com. The online documentation that comes in the JGL package is quite good and should be adequate to get you started.

summary

To review the containers provided in the standard Java library:

1. An array associates numerical indices to objects. It holds objects of a known type, so you don't have to cast the result when you're looking up an object. It can be multi-dimensional, and it can hold primitives. However, its size is fixed once you create it.

2. A **Vector** also associates numerical indices to objects – you can think of arrays and **Vector**s as random-access containers. The **Vector** automatically resizes itself as you add more elements. But a **Vector** can only hold **Object** handles, so it won't hold primitives and you must always cast the result when you pull an **Object** handle out of a container.

3. A **Hashtable** is a type of **Dictionary**, which is a way to associate, not numbers, but *objects* with other objects. A **Hashtable** also supports random access to objects, in fact, its whole design is focused around rapid access.

4. A **Stack** is a last-in, first-out (LIFO) queue.

If you're familiar with data structures, you may wonder why there's not a larger set of containers. From a functionality standpoint, do you really *need* a larger set of containers? With a **Hashtable** you can put things in, find them quickly and with an **Enumeration**, iterate through the sequence and perform an operation on every element in the sequence. That's a very powerful tool, and maybe it should be enough.

But a **Hashtable** has no concept of order. **Vector**s and arrays give you a linear order, but it's expensive to insert an element into the middle of either one. In addition, queues and dequeues and priority queues and trees are about *ordering* the elements, not just putting them in and later finding them or moving through them linearly. These data structures are also very useful, and that's why they were included in Standard C++. For this reason, you should only consider the containers in the standard Java library as a starting point, and use the JGL when your needs go beyond that.

exercises

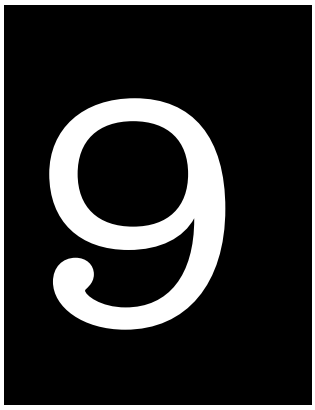
1. Create a new class called **gerbil** with an **int gerbilNumber** that's initialized in the constructor (similar to the **mouse** example in this chapter). Give it a method called **hop( )** that prints out which gerbil number this is and that it's hopping. Create a **Vector** and add a bunch of **gerbil** objects to the **Vector**. Now use the **elementAt( )** method to move through the **Vector** and call **hop( )** for each **gerbil**.

2. Modify the exercise one so you use an **Enumeration** to move through the **Vector** while calling **hop( )**.

3. Take the **gerbil** class in exercise one and put it into a **Hashtable** instead, associating the name of the **gerbil** as a **String** (the key) for each **gerbil** (the value) you put in the table. Get an **Enumeration** for the **keys( )** and use it to move through the **Hashtable**, looking up the **gerbil** for each key and printing out the key and telling the **gerbil** to **hop( )**.

4. Change exercise one in Chapter 7 to use a **Vector** to hold the **rodent**s and an **Enumeration** to move through the sequence of **rodent**s. Remember that a **Vector** only holds **Object**s so you'll have to use a cast (i.e.: RTTI) when accessing individual **rodent**s.

5. (Challenging). Find the source code for **Vector** in the Java source code library that comes with all Java distributions. Copy this code and make a special version called **intVector** that only holds **int**s. Consider what it would take to make a special version of **Vector** for all the primitive types. Now consider what happens if you want to make a linked list class that works with all the primitive types. If parameterized types are ever implemented in Java, they will provide a way to do this work for you, automatically (as well as many other benefits).



9: error handling with exceptions

The basic philosophy of Java is that "badly-formed code will not be run."

And, as with C++, the ideal time to catch the error is at compile time, before you even try to run the program. However, not all errors can be detected at compile time. The rest of the problems must be handled at run-time through some formality that allows the originator of the error to pass appropriate information to a recipient who will know how to properly handle the difficulty.

In C and other early languages, there could be several of these formalities, and they were generally established by convention and not part of the programming language. Typically you returned a special value or set a flag, and the recipient was supposed to look at the value or the flag and determine that something was amiss. However, as the years passed it was discovered that programmers who use a library tend to think of themselves as invincible, as in "yes, errors may happen to others but not in *my* code." So, not too surprisingly, they wouldn't check for the error conditions (and sometimes the error conditions were too silly to check for[1]). If you *were* thorough enough to check for an error every time

[1] The C programmer can look up the return value of **printf( )** for an example of this.

you called a method, your code could turn into an unreadable nightmare. Because programmers could still coax systems out of these languages they were resistant to admitting the truth: this approach to handling errors was a major limitation to creating large, robust, maintainable programs.

The solution is to take the casual nature out of error handling, to enforce the formality. This actually has a long history, since implementations of *exception handling* go back to operating systems in the 60's and even to BASIC's **on error goto**. But C++ exception handling was based on ADA, and Java's is based primarily on C++ (although Object Pascal programmers will see a greater similarity).

The word "exception" is meant in the sense of "I take exception to that." At the point where the problem occurs you may not know what to do with it, but you do know that you can't just continue merrily on, that you must stop and somebody somewhere must figure out what to do. But you don't have enough information in the current context to fix the problem. So you hand the problem out to a higher context where someone is qualified to make the proper decision (very much like a chain of command).

The other rather significant benefit of exceptions is that they clean up error handling code. Instead of checking for a particular error and dealing with it at multiple places in your program, you no longer need to check at the point of the method call (since the exception will guarantee that someone catches it) and you only need to handle the problem in one place, the so-called *exception handler*. This saves you code, and it separates the code that describes what you want to do from the code that is executed when things go awry. In general, reading, writing and debugging code becomes much clearer than when using the old way.

Because exception handling is enforced by the Java compiler, there are only so many examples that can be written without learning about exception handling. This chapter introduces you to the code you need to write to properly handle the exceptions, and the way you can generate your own exceptions if one of your methods gets into trouble.

basic exceptions

An *exceptional condition* is a problem that prevents the continuation of the method or scope that you're in. It's important to distinguish an exceptional condition from a normal problem, where you have enough information in the current context to somehow cope with the difficulty. With an exceptional condition, you cannot continue processing because you don't have the information necessary to deal with the problem *in the current context*. The only thing you can do is jump out of the current context and relegate that problem to a higher context. This is what happens when you throw an exception.

A simple example is a divide. If you're about to divide by zero, it's worth checking to make sure you don't go ahead and perform the divide. But what does it mean that the denominator is zero? Maybe you know, in the context of the problem you're trying to solve in that particular method, how to deal with a zero denominator. But if it's an unexpected value, you can't deal with it and so must throw an exception rather than continuing along that path.

When you throw an exception, several things happen. First, the exception object itself is created, in the same way that any Java object is created: on the heap, with **new**. Then the current path of execution (the one you couldn't continue, remember) is stopped and the handle for the exception object is ejected from the current context. At this point the exception-handling mechanism takes over and begins to look for an appropriate place to continue executing the program. This appropriate place is the *exception handler*, whose job is to recover from the problem so the program may either try another tack or simply continue.

As a simple example of throwing an exception, consider an object handle called **t**. It's possible you might be passed a handle that hasn't been initialized, and so you may want to check before trying to call a method using that object handle. You can send information about the error into a larger context by creating an object representing your information and "throwing" it out of your current context. This is called *throwing an exception*. Here's what it looks like:

```
if(t == null)
  throw new NullPointerException();
```

This throws the exception, which allows you – in the current context – to abdicate responsibility for thinking about the issue further. It's just magically handled somewhere else. Precisely *where* will be shown shortly.

exception arguments

Like any object in Java, you always create exceptions on the heap using **new**, and a constructor gets called. There are two constructors in all the standard exceptions; the first is the default constructor, and the second takes a string argument so you can place pertinent information in the exception:

```
if(t == null)
  throw new NullPointerException("t = null");
```

This string can later be extracted using various methods, as will be shown later.

The keyword **throw** causes a number of relatively magical things to happen. First it executes the **new**-expression to create an object that isn't there under normal program execution, and of course the constructor is called for that object. Then the object is, in effect, "returned" from the method, even though that object type isn't normally what the method is designed to return. A simplistic way to think about exception handling is as an alternate return mechanism, although you get into trouble if you take the analogy too far. You can also exit from ordinary scopes by throwing an exception. But a value is returned, and the method or scope exits.

Any similarity to method returns ends there because *where* you return to is someplace completely different than for a normal method call. (You end up in an appropriate exception handler that may be miles away from where the exception was thrown.)

In addition, you can throw as many different types of objects as you want. Typically, you'll throw a different class of exception for each different type of error. The idea is to store the information in the exception object and the *type* of exception object, so someone in the bigger context can figure out what to do with your exception.

catching an exception

If a method throws an exception, it must assume that exception is caught and dealt with. One of the advantages of Java exception handling is that it allows you to concentrate on the problem you're actually trying to solve in one place, and then deal with the errors from that code in another place.

To see how an exception is caught, you must first understand the concept of a *guarded region*, which is a section of code that may produce exceptions, and which is followed by the code to handle those exceptions.

the try block

If you're inside a method and you throw an exception (or another method you call within this method throws an exception), that method will exit in the process of throwing. If you don't want a **throw** to leave a method, you can set up a special block within that method to capture the exception. This is called the *try block* because you "try" your various method calls there. The try block is an ordinary scope, preceded by the keyword **try**:

```
try {
  // code that may generate exceptions
}
```

If you were carefully checking for errors in a programming language that didn't support exception handling, you'd have to surround every method call with setup and error testing code, even if you call the same method several times. With exception handling, you put everything in a try block and

capture all the exceptions in one place. This means your code is a lot easier to write and easier to read because the goal of the code is not confused with the error checking.

exception handlers

Of course, the thrown exception must end up someplace. This is the *exception handler*, and there's one for every exception type you want to catch. Exception handlers immediately follow the try block and are denoted by the keyword **catch**:

```
try {
  // code that may generate exceptions
} catch(type1 id1) {
  // handle exceptions of type1
} catch(type2 id2) {
  // handle exceptions of type2
} catch(type3 id3) {
  // handle exceptions of type3
}

// etc...
```

Each catch clause (exception handler) is like a little method that takes one and only one argument of one particular type. The identifier (**id1**, **id2**, and so on) may be used inside the handler, just like a method argument. Sometimes you never use the identifier because the type of the exception gives you enough information to deal with the exception, but the identifier must still be there.

The handlers must appear directly after the try block. If an exception is thrown, the exception-handling mechanism goes hunting for the first handler with an argument that matches the type of the exception. Then it enters that catch clause, and the exception is considered handled. (The search for handlers stops once the catch clause is finished.) Only the matching catch clause executes; it's not like a **switch** statement where you need a **break** after each **case** to prevent the remaining ones from executing.

Notice that, within the try block, a number of different method calls might generate the same exception, but you only need one handler.

termination vs. resumption

There are two basic models in exception-handling theory. In *termination* (which is what Java and C++ support) you assume the error is so critical there's no way to get back to where the exception occurred. Whoever threw the exception decided there was no way to salvage the situation, and they don't *want* to come back.

The alternative is called *resumption*. It means the exception handler is expected to do something to rectify the situation, and then the faulting method is retried, presuming success the second time. If you want resumption, it means you still hope to continue execution after the exception is handled. In this case, your exception is more like a method call — which is how you should set up situations in Java where you want resumption-like behavior (that is, don't throw an exception; call a method that fixes the problem). Alternatively, place your **try** block inside a **while** loop that keeps reentering the **try** block until the result is satisfactory.

Historically, programmers using operating systems that supported resumptive exception handling eventually ended up using termination-like code and skipping resumption. So although resumption sounds attractive at first, it seems it isn't quite so useful in practice. The dominant reason is probably the *coupling* that results: your handler must often be aware of where the exception is thrown from, which makes the code difficult to write and maintain, especially for large systems where the exception can be generated from many points.

the exception specification

In Java, you're required to inform the person calling your method of the exceptions that might be thrown out of that method. This is very civilized because it means the caller can know exactly what

code to write to catch all potential exceptions. Of course, if source code is available, the client programmer could hunt through and look for **throw** statements, but very often a library doesn't come with sources. To prevent this from being a problem, Java provides syntax (and *forces* you to use that syntax) to allow you to politely tell the client programmer what exceptions this method throws, so the client programmer may handle them. This is the *exception specification* and it's part of the method declaration, appearing after the argument list.

The exception specification uses an additional keyword, **throws**, followed by a list of all the potential exception types. So your method definition may look like this:

```
void f() throws tooBig, tooSmall, divZero { //...
```

If you say

```
void f() { // ...
```

it means that no exceptions are thrown from the method (*except* for the exceptions of type **RuntimeException**, which can reasonably be thrown anywhere – this is described later).

You can't lie about an exception specification – if your method causes exceptions and doesn't handle them, the compiler will detect this and tell you that you must either handle the exception or indicate with an exception specification that it may be thrown from your method. By enforcing exception specifications from top to bottom, Java guarantees that exception-correctness can be ensured *at compile time*[2].

There is one place you can lie: you can claim to throw an exception that you don't. The compiler takes your word for it, and forces the users of your method to treat it as if it really does throw that exception. This has the beneficial effect of being a placeholder for that exception, so you can actually start throwing the exception later without requiring changes to existing code.

catching any exception

It is possible to create a handler that catches any type of exception. You do this by catching the base-class exception type **Exception** (there are other types of base exceptions, but **Exception** is the base that's pertinent to virtually all programming activities):

```
catch(Exception e) {
  System.out.println("caught an exception");
}
```

This will catch any exception, so if you use it you'll want to put it at the *end* of your list of handlers to avoid pre-empting any exception handlers that follow it.

Since the **Exception** class is the base of all the exception classes that are important to the programmer, you don't get much specific information about the exception, but you can call the methods that come from *its* base type **Throwable**:

String getMessage( )
Gets the detail message .

String toString( )
Returns a short description of the Throwable, including the detail message if there is one.

void printStackTrace( )
void printStackTrace(PrintStream)
Prints the Throwable and the Throwable's call stack trace. The first version prints to the standard

---

[2] This is a significant improvement over C++ exception handling, which doesn't catch violations of exception specifications until run time, when it's not very useful.

output, the second prints to a stream of your choice. The call stack shows the sequence of method calls that brought you to the point where the exception was thrown.

In addition, you get some other methods from **Throwable**'s base type **Object** (everybody's base type). The one that may come in handy for exceptions is **getClass( )**, which returns an object representing the class of this object. You can in turn query this "class object" for its name with **getName( )** or **toString( )**. You can also do more sophisticated things with class objects that aren't necessary in exception handling. Class objects will be studied later in the book.

Here's an example that shows the use of the **Exception** methods:

```
//: ExceptionMethods.java
// Demonstrating the Exception Methods

public class ExceptionMethods {
  public static void main(String args[]) {
    try {
      throw new Exception("Here's my Exception");
    } catch(Exception e) {
      System.out.println("Caught Exception");
      System.out.println(
        "e.getMessage(): " + e.getMessage());
      System.out.println(
        "e.toString(): " + e.toString());
      System.out.println("e.printStackTrace():");
      e.printStackTrace();
    }
  }
} ///:~
```

The output for this program is:

```
Caught Exception
e.getMessage(): Here's my Exception
e.toString(): java.lang.Exception: Here's my Exception
e.printStackTrace():
java.lang.Exception: Here's my Exception
        at ExceptionMethods.main
```

You can see that the methods provide successively more information – each is effectively a superset of the previous one.

rethrowing an exception

Sometimes you'll want to rethrow the exception that you just caught, particularly when you use **Exception** to catch any exception. Since you already have the handle to the current exception, you can simply re-throw that handle:

```
catch(Exception e) {
  System.out.println("an exception was thrown");
  throw e;
}
```

Any further **catch** clauses for the same **try** block are still ignored — the **throw** causes the exception to go to the exception handlers in the next-higher context. In addition, everything about the exception object is preserved, so the handler at the higher context that catches the specific exception type can extract all the information from that object.

If you just re-throw the current exception, the information that you print about that exception in **printStackTrace( )** will pertain to the exception's origin, not the place where you re-throw it. If you want to install new stack trace information, you can do so by calling **fillInStackTrace( )**, which returns

an exception object that it creates by stuffing the current stack information into the old exception object. Here's what it looks like:

```
//: Rethrowing.java
// Demonstrating fillInStackTrace()

public class Rethrowing {
  public static void f() throws Exception {
    System.out.println(
      "originating the exception in f()");
    throw new Exception("thrown from f()");
  }
  public static void g() throws Throwable {
    try {
      f();
    } catch(Exception e) {
      System.out.println(
        "Inside g(), e.printStackTrace()");
      e.printStackTrace();
      throw e; // 17
      // throw e.fillInStackTrace(); // 18
    }
  }
  public static void
  main(String args[]) throws Throwable {
    try {
      g();
    } catch(Exception e) {
      System.out.println(
        "Caught in main, e.printStackTrace()");
      e.printStackTrace();
    }
  }
} ///:~
```

The important line numbers are marked inside of comments. With line 17 uncommented (as shown), the output is:

```
originating the exception in f()
Inside g(), e.printStackTrace()
java.lang.Exception: thrown from f()
        at Rethrowing.f(Rethrowing.java:8)
        at Rethrowing.g(Rethrowing.java:12)
        at Rethrowing.main(Rethrowing.java:24)
Caught in main, e.printStackTrace()
java.lang.Exception: thrown from f()
        at Rethrowing.f(Rethrowing.java:8)
        at Rethrowing.g(Rethrowing.java:12)
        at Rethrowing.main(Rethrowing.java:24)
```

So the exception stack trace always remembers its true point of origin, no matter how many times it gets rethrown.

With line 17 commented and line 18 uncommented, **fillInStackTrace( )** is used instead, and the result is:

```
originating the exception in f()
Inside g(), e.printStackTrace()
java.lang.Exception: thrown from f()
        at Rethrowing.f(Rethrowing.java:8)
        at Rethrowing.g(Rethrowing.java:12)
```

```
        at Rethrowing.main(Rethrowing.java:24)
Caught in main, e.printStackTrace()
java.lang.Exception: thrown from f()
        at Rethrowing.g(Rethrowing.java:18)
        at Rethrowing.main(Rethrowing.java:24)
```

Because of **fillInStackTrace( )**, line 18 becomes the new point of origin of the exception.

The class **Throwable** must appear in the exception specification for **g( )** and **main( )** because **fillInStackTrace( )** produces a handle to a **Throwable** object. Since **Throwable** is a base class of **Exception**, it's possible to get an object that's a **Throwable** but *not* an **Exception**, so the handler for **Exception** in **main( )** might miss it. To make sure everything is in order, the compiler forces an exception specification for **Throwable**.

It's also possible to rethrow a different exception than the one you caught. If you do this, you get a similar effect as when using **fillInStackTrace( )**: the information about the original site of the exception is lost, and what you're left with is the information pertaining to the new **throw**:

```
//: RethrowNew.java
// Rethrow a different object than you catch

public class RethrowNew {
  public static void f() throws Exception {
    System.out.println(
      "originating the exception in f()");
    throw new Exception("thrown from f()");
  }
  public static void
  main(String args[]) {
    try {
      f();
    } catch(Exception e) {
      System.out.println(
        "Caught in main, e.printStackTrace()");
      e.printStackTrace();
      throw new NullPointerException("from main");
    }
  }
} ///:~
```

The output is:

```
originating the exception in f()
Caught in main, e.printStackTrace()
java.lang.Exception: thrown from f()
        at RethrowNew.f(RethrowNew.java:8)
        at RethrowNew.main(RethrowNew.java:13)
java.lang.NullPointerException: from main
        at RethrowNew.main(RethrowNew.java:18)
```

The final exception only knows that it came from **main( )**, and not from **f( )**. Notice that **Throwable** isn't necessary in any of the exception specifications.

You never have to worry about cleaning up the previous exception, or any exceptions for that matter: they're all heap-based objects created with **new**, so the garbage collector automatically cleans them all up.

standard Java exceptions

Java contains a class called **Throwable** that describes anything that can be thrown as an exception. There are two general types of **Throwable** objects ("types of" = "inherited from"): **Error** represents compile-time and system errors that you don't worry about catching (except in very special cases). **Exception** is the basic type that can be thrown from any of the standard Java library class methods and from your methods and run-time accidents.

What follows is a brief description of the exception classes that come with the basic Java toolkit (from Sun Microsystems, at this writing). If you get an enhanced Java development environment from another vendor, you may have additional exception classes. You may also need to derive your own exceptions from one of the subtypes of **Exception** (this will be covered later in the chapter).

```
java.lang.Exception
```

This is the basic exception class your program can catch. Other exceptions are derived from this. What follows is all the different types of exceptions in Java 1.0. Java 1.1 adds a number of new exceptions, but this will give you the basic idea: the name of the exception represents the problem that occurred. The exceptions are organized by class, and the exception subtypes are indented. The exceptions are not all defined in **java.lang**; some are created to support other libraries like **util**, **net** and **io**, which you can see from their full class names.

```
java.awt.AWTException
java.lang.ClassNotFoundException
java.lang.CloneNotSupportedException
java.io.IOException
    java.io.EOFException
    java.io.FileNotFoundException
    java.io.InterruptedIOException
    java.net.MalformedURLException
    java.net.ProtocolException
    java.net.SocketException
    java.io.UTFDataFormatException
    java.net.UnknownHostException
    java.net.UnknownServiceException
java.lang.IllegalAccessException
java.lang.InstantiationException
java.lang.InterruptedException
java.lang.NoSuchMethodException
java.lang.RuntimeException
    java.lang.ArithmeticException
    java.lang.ArrayStoreException
    java.lang.ClassCastException
    java.util.EmptyStackException
    java.lang.IllegalArgumentException
        java.lang.IllegalThreadStateException
        java.lang.NumberFormatException
    java.lang.IllegalMonitorStateException
    java.lang.IndexOutOfBoundsException
        java.lang.ArrayIndexOutOfBoundsException
        java.lang.StringIndexOutOfBoundsException
    java.lang.NegativeArraySizeException
    java.util.NoSuchElementException
    java.lang.NullPointerException
    java.lang.SecurityException
```

exception descriptions

Although the exception names are intended to be relatively self-explanatory, descriptions can be helpful.

```
java.awt.AWTException
```

An exception that comes from the Java Abstract Window Toolkit (AWT).

```
java.lang.ClassNotFoundException
```

The first time you try to create an object of a particular class, that class is loaded into the program space. This exception indicates that the class couldn't be found.

```
java.lang.CloneNotSupportedException
```

Most objects allow themselves to be duplicated ("cloned") by using the **clone( )** method. If a class disallows cloning and you try to do it anyway you'll get this exception.

```
java.io.IOException
```

Describes the general class of IO (input/output) exceptions. All IO exceptions are inherited from this.

```
java.io.EOFException
```

You tried to read past the end of a file (EOF).

```
java.io.FileNotFoundException
```

You tried to open a file that didn't exist.

```
java.io.InterruptedIOException
```

An IO operation got interrupted in the middle before it was successfully completed. Inside this exception object there's a **public int bytesTransferred** variable containing the number of bytes read or written before the interruption.

```
java.net.MalformedURLException
```

A URL (Universal Resource Locator, e.g. an Internet address) was not built correctly and so couldn't be parsed.

```
java.net.ProtocolException
```

The **Socket** class has a protocol error. The **Socket** class catches this exception itself.

```
java.net.SocketException
```

You tried to use a **Socket** and something went wrong.

```
java.io.UTFDataFormatException
```

UTF-8 is the way that Unicode characters can be transformed into ASCII, and can be used to store and transmit Unicode text. If a UTF-8 string is malformed, you get this exception.

```
java.net.UnknownHostException
```

The host name didn't match with any known hosts.

```
java.net.UnknownServiceException
```

The service you requested of a particular network connection didn't exist.

```
java.lang.IllegalAccessException
```

You tried to call a method that has restricted access.

```
java.lang.InstantiationException
```

You tried to create an object of a class that is **abstract** or an **interface**; you can't create objects from either of these. This will make sense to you later in the book.

```
java.lang.InterruptedException
```

This concerns threading, another not-yet-introduced concept. This exception means that some other thread interrupted this one, when the current thread called a **wait( )** or **sleep( )** method. In effect, it's a way to break out of those methods before they complete their normal cycle.

```
java.lang.NoSuchMethodException
```

This exception is rarely thrown, since the existence of methods is primarily determined at compile time. You can get this exception if you remove a method from a class, recompile that class without dependency tracking, and then run a program containing classes which called the removed method.

```
java.lang.RuntimeException
```

The base class for all the errors that indicate some kind of programming bug, either yours or the client programmer using your library. Because they indicate bugs, you virtually never catch a **RuntimeException**; that all happens automatically. In your own packages you may choose to throw some of the **RuntimeException**s, discussed more completely later.

```
java.lang.ArithmeticException
```

Thrown if you try to divide by zero or perform a modulus by zero.

```
java.lang.ArrayStoreException
```

Thrown because you tried to put objects of one type into an array that holds objects of some other, incompatible type.

```
java.lang.ClassCastException
```

Casting will be discussed in future chapters. This exception is thrown if you try to cast to an incorrect type. It's possible to avoid this exception by checking the type first using the keyword **instanceof**.

```
java.util.EmptyStackException
```

An object of the **Stack** class (described in Chapter 8) is empty when you tried to get more objects out of it.

```
java.lang.IllegalArgumentException
```

A general-purpose exception that can be thrown by a library that receives an illegal argument. You may also choose to throw it from one of your methods.

```
java.lang.IllegalThreadStateException
```

You've done something silly with a thread, like trying to start it when it is already running. You can also throw this from your own threading code. Threads are the subject of Chapter 14.

```
java.lang.NumberFormatException
```

Indicates that the called method received an illegal number format.

```
java.lang.IllegalMonitorStateException
```

A *monitor* is used for synchronizing threads by indicating the current state of the thread that owns it. This exception is thrown if that state becomes illegal. Threads are the subject of Chapter 14.

```
java.lang.IndexOutOfBoundsException
```

The base class for the following two.

```
java.lang.ArrayIndexOutOfBoundsException
```

When using an array, you've tried to select an element with an index of less than zero or greater than or equal to the size of the array.

```
java.lang.StringIndexOutOfBoundsException
```

In a **String**, you tried to select a character using an index that's less than zero or greater than or equal to the length of the **String**.

```
java.lang.NegativeArraySizeException
```

You tried to allocate an array with a negative size.

```
java.util.NoSuchElementException
```

Means that an **Enumeration** (described in Chapter 8) is empty.

```
java.lang.NullPointerException
```

You tried to select a member of an object using a handle that hadn't been initialized to a new object (thus it's null). This is obviously a bug on the part of the programmer, so you virtually never check for it, and instead allow it to be part of the debugging process.

```
java.lang.SecurityException
```

Some situations, in particular when an applet is running inside a browser, prevent certain operations from occurring because they are unsafe. A primary example is writing to disk – by preventing this in an applet, it's very difficult (some claim impossible) for a hostile applet programmer to create a virus. If you try to do something that would violate security inside an applet, this exception is thrown.

the special case of RuntimeException

The first example in this chapter was

```
if(t == null)
  throw new NullPointerException();
```

It can be a bit horrifying to think that you must check for null on every handle that is passed into a method (since you can't know if the caller has passed you a valid handle). Fortunately, you don't – this is part of the standard run-time checking that Java performs for you, and if any call is made to a null handle, Java will automatically throw a **NullPointerException**. So the above bit of code is always superfluous. In addition, you never need to write an exception specification saying that a method may throw a **NullPointerException**, since that's just assumed.

There's a whole group of exception types that are in this category: they're always thrown automatically by Java and you don't need to include them in your exception specifications. Conveniently enough, they're all grouped by putting them under a single base class called **RuntimeException**, which is a perfect example of inheritance: it establishes a family of types that have some characteristics and behaviors in common. Here's the subpart of the whole list, again:

```
java.lang.RuntimeException
    java.lang.ArithmeticException
    java.lang.ArrayStoreException
    java.lang.ClassCastException
    java.util.EmptyStackException
    java.lang.IllegalArgumentException
        java.lang.IllegalThreadStateException
        java.lang.NumberFormatException
    java.lang.IllegalMonitorStateException
    java.lang.IndexOutOfBoundsException
        java.lang.ArrayIndexOutOfBoundsException
        java.lang.StringIndexOutOfBoundsException
```

```
    java.lang.NegativeArraySizeException
    java.util.NoSuchElementException
    java.lang.NullPointerException
    java.lang.SecurityException
```

If you were forced to check for any of these your code could get pretty messy so it's just as well that Java takes care of them.

What if you never catch such exceptions? Since the compiler doesn't enforce exception specifications for these, it's quite plausible that a **RuntimeException** could percolate all the way out to your **main( )** method without being caught. To see what happens in this case, try the following example:

```
//: NeverCaught.java
// Ignoring RuntimeExceptions

public class NeverCaught {
  static void f() {
    throw new RuntimeException("From f()");
  }
  static void g() {
    f();
  }
  public static void main(String args[]) {
    g();
  }
} ///:~
```

You can already see that a **RuntimeException** (or anything inherited from it) is a special case, since the compiler doesn't require an exception specification for these types.

The output is:

```
java.lang.RuntimeException: From f()
        at NeverCaught.f(NeverCaught.java:9)
        at NeverCaught.g(NeverCaught.java:12)
        at NeverCaught.main(NeverCaught.java:15)
```

So the answer is: if a RuntimeException gets all the way out to **main( )** without being caught, **printStackTrace( )** is called for that exception as the program exits.

Keep in mind that it's only possible to not catch an exception like this for **RuntimeException**s, since all other handling is carefully enforced by the compiler. The reasoning is that a **RuntimeException** represents either a programming error you cannot catch (receiving a null handle handed to your method by a client programmer, for example) or one that you, as a programmer, should have checked for in your code (such as **ArrayIndexOutOfBoundsException** where you should have paid attention to the size of the array). One way or another, a **RuntimeException** represents a programming error. You can see what a tremendous benefit it is to have exceptions in this case, since they assist in the debugging process.

It's interesting to note that you cannot classify Java exception handling as a single-purpose tool. Yes, it is designed to handle those pesky run-time errors that will occur because of forces outside the control of your code, but it's also essential for certain types of programming bugs that the compiler cannot detect.

creating your own exceptions

You're not stuck using the Java exceptions. This is important because you'll often need to create your own exceptions to denote a special error that your library is capable of creating, but which was not foreseen when the Java hierarchy was created.

To create your own exception class, you're forced to inherit from an existing type of exception, preferably one that is very close in meaning to your new exception. Inheriting an exception is quite simple:

```
//: Inheriting.java
// Inheriting your own exceptions

class MyException extends Exception {
  public MyException() {}
  public MyException(String msg) {
    super(msg);
  }
}

public class Inheriting {
  public static void f() throws MyException {
    System.out.println(
      "Throwing MyException from f()");
    throw new MyException();
  }
  public static void g() throws MyException {
    System.out.println(
      "Throwing MyException from g()");
    throw new MyException("Originated in g()");
  }
  public static void main(String args[]) {
    try {
      f();
    } catch(MyException e) {
      e.printStackTrace();
    }
    try {
      g();
    } catch(MyException e) {
      e.printStackTrace();
    }
  }
} ///:~
```

The inheritance occurs in the creation of the new class:

```
class MyException extends Exception {
  public MyException() {}
  public MyException(String msg) {
    super(msg);
  }
}
```

The key phrase here is **extends Exception**, which says "it's everything an **Exception** is, and more." The added code is very small – the addition of two constructors that define the way **MyException** is created.

The first constructor is the default constructor, since it takes no arguments. It appears to do nothing, but constructors are special because they control the creation of an object. All parts of the object must be properly created, including the base-class parts – that is, the parts in **Exception**. This takes place by calling the **Exception** constructor. In the **MyException** default constructor, you don't see the **Exception** default constructor being called, but it is anyway. Because it's so important, the compiler does it for you, automatically calling **Exception**'s default constructor (if there isn't one, the compiler complains). Note that this special automatic calling of base-class functionality only exists for constructors.

What if the base-class constructor has arguments? Then you must use a special keyword, **super**, to explicitly call that constructor, passing the appropriate arguments:

```
  public MyException(String msg) {
    super(msg);
  }
```

Although **super** can be used in other situations, inside a constructor it means "call the base-class constructor (**Exception** in this case) passing it these arguments." The result is that **Exception(msg)** gets called to build the base-class part of **MyException**. That is the constructor that stores the **msg String** away as the detail message, to be produced with calls to **getMessage( )** etc.

The output of the program is:

```
Throwing MyException from f()
MyException
        at Inheriting.f(Inheriting.java:16)
        at Inheriting.main(Inheriting.java:24)
Throwing MyException from g()
MyException: Originated in g()
        at Inheriting.g(Inheriting.java:20)
        at Inheriting.main(Inheriting.java:29)
```

You can see the absence of the detail message in the **MyException** thrown from **f( )**.

The process of creating your own exceptions can be taken further: you can add extra constructors and members:

```
//: Inheriting2.java
// Inheriting your own exceptions

class MyException2 extends Exception {
  public MyException2() {}
  public MyException2(String msg) {
    super(msg);
  }
  public MyException2(String msg, int x) {
    super(msg);
    i = x;
  }
  public int val() { return i; }
  private int i;
}

public class Inheriting2 {
  public static void f() throws MyException2 {
    System.out.println(
      "Throwing MyException2 from f()");
    throw new MyException2();
  }
  public static void g() throws MyException2 {
    System.out.println(
      "Throwing MyException2 from g()");
    throw new MyException2("Originated in g()");
  }
  public static void h() throws MyException2 {
    System.out.println(
      "Throwing MyException2 from h()");
    throw new MyException2("Originated in h()", 47);
  }
  public static void main(String args[]) {
```

```
    try {
      f();
    } catch(MyException2 e) {
      e.printStackTrace();
    }
    try {
      g();
    } catch(MyException2 e) {
      e.printStackTrace();
    }
    try {
      h();
    } catch(MyException2 e) {
      e.printStackTrace();
      System.out.println("e.val() = " + e.val());
    }
  }
} ///:~
```

A data member **i** has been added, along with a method that reads that value, and an additional constructor that sets it. The output is:

```
Throwing MyException2 from f()
MyException2
        at Inheriting2.f(Inheriting2.java:22)
        at Inheriting2.main(Inheriting2.java:34)
Throwing MyException2 from g()
MyException2: Originated in g()
        at Inheriting2.g(Inheriting2.java:26)
        at Inheriting2.main(Inheriting2.java:39)
Throwing MyException2 from h()
MyException2: Originated in h()
        at Inheriting2.h(Inheriting2.java:30)
        at Inheriting2.main(Inheriting2.java:44)
e.val() = 47
```

Since an exception is just another kind of object, you can continue this process of embellishing the power of your exception classes.

exception restrictions

When you override a method, you can only throw exceptions that have been specified in the base-class version of the method. This is a very useful restriction, since it means that code that works with the base class will automatically work with any object derived from the base class (a fundamental OOP concept, of course), including exceptions.

This example demonstrates the kinds of restrictions imposed (at compile time) for exceptions:

```
//: StormyInning.java
// Overridden methods may only throw exceptions
// specified in their base-class versions, or
// exceptions derived from the base-class
// exceptions.

class BaseballException extends Exception {}
class Foul extends BaseballException {}
class Strike extends BaseballException {}

abstract class Inning {
```

```
  Inning() throws BaseballException {}
  void event () throws BaseballException {
   // Doesn't actually have to throw anything
  }
  abstract void atBat() throws Strike, Foul;
  void walk() {} // Throws nothing
}

class StormException extends Exception {}
class RainedOut extends StormException {}
class PopFoul extends Foul {}

interface Storm {
  void event() throws RainedOut;
  void rainHard() throws RainedOut;
}

public class StormyInning extends Inning
    implements Storm {
  // OK to add new exceptions for constructors,
  // but you must deal with the base constructor
  // exceptions:
  StormyInning() throws RainedOut,
    BaseballException {}
  StormyInning(String s) throws Foul,
    BaseballException {}
  // Regular methods must conform to base class:
//! void walk() throws PopFoul {} //Compile error
  // Interface CANNOT add exceptions to existing
  // methods from the base class:
//! public void event() throws RainedOut {}
  // If the method doesn't already exist in the
  // base class, the exception is OK:
  public void rainHard() throws RainedOut {}
  // You can choose not to throw any exceptions,
  // even if base version does:
  public void event() {}
  // Overridden methods can throw
  // inherited exceptions:
  void atBat() throws PopFoul {}
  public static void main(String args[]) {
    try {
      StormyInning si = new StormyInning();
      si.atBat();
    } catch(PopFoul e) {
    } catch(RainedOut e) {
    } catch(BaseballException e) {}
    // Strike not thrown in derived version.
    try {
      // What happens if you upcast?
      Inning i = new StormyInning();
      i.atBat();
      // You must catch the exceptions from the
      // base-class version of the method:
    } catch(Strike e) {
    } catch(Foul e) {
    } catch(RainedOut e) {
    } catch(BaseballException e) {}
  }
} ///:~
```

In **Inning**, you can see that both the constructor and the **event( )** method say they will throw an exception, but they never actually do. This is legal because it allows you to force the user to catch any exceptions that you may add in overridden versions of **event( )**. The same idea holds for **abstract** methods, as seen in **atBat( )**.

The **interface Storm** is interesting because it contains one method (**event( )**)that is defined in **Inning**, and one method that isn't. Both methods throw a new type of exception, **RainedOut**. When **StormyInning extends Inning** and **implements Storm**, you'll see that the **event( )** method in **Storm** *cannot* change the exception interface of **event( )** in **Inning**. Again, this makes sense because otherwise you'd never know if you were catching the right thing when working with the base class. Of course, if a method described in an **interface** is not in the base class, like **rainHard( )**, then there's no problem if it throws exceptions.

The restriction on exceptions does not apply to constructors. You can see in **StormyInning** that a constructor can throw anything it wants, regardless of what the base-class constructor throws. However, since a base-class constructor must always be called one way or another (here, the default constructor is called automatically), the derived-class constructor must declare any base-class constructor exceptions in its exception specification.

The reason **StormyInning.walk( )** will not compile is that it throws an exception, while **Inning.walk( )** does not. If this was allowed, then you could write code that called **Inning.walk( )** and that didn't have to handle any exceptions, but then when you substituted an object of a class derived from **Inning**, exceptions would be thrown so your code would break. By forcing the derived-class methods to conform to the exception specifications of the base-class methods, substitutability of objects is maintained.

The overridden **event( )** method shows that a derived-class version of a method may choose not to throw any exceptions, even if the base class version does. Again, this is fine since it doesn't break any code that is written assuming the base-class version throws exceptions. Similar logic applies to **atBat( )**, which throws **PopFoul**, an exception that is derived from **Foul** thrown by the base-class version of **atBat( )**. This way, if someone writes code that works with **Inning** and calls **atBat( )**, they must catch the **Foul** exception. Since **PopFoul** is derived from **Foul**, the exception handler will also catch **PopFoul**.

The last point of interest is in **main( )**. Here you can see that if you're dealing with exactly a **StormyInning** object, the compiler only forces you to catch exceptions that are specific to that class, but if you upcast to the base type then the compiler (correctly) forces you to catch the exceptions for the base type. All these constraints produce much more robust exception-handling code[3].

performing cleanup with finally

There's often some piece of code that you may want executed whether or not an exception occurs in a **try** block. This usually pertains to some operation other than memory recovery (since that's taken care of by the garbage collector). To achieve this effect, you use a **finally** clause[4] at the end of all the exception handlers. The full picture of an exception-handling section is thus:

---

[3] ANSI/ISO C++ added similar constraints that require derived-method exceptions to be the same as, or derived from, the exceptions thrown by the base-class method. This is one case where C++ is actually able to check exception specifications at compile time.

[4] C++ exception handling does not have the **finally** clause because it relies on destructors to accomplish this sort of cleanup.

```
try {
 // The guarded region:
 // Dangerous stuff that may throw A, B, or C
} catch (A a1) {
 // Handle A
} catch (B b1) {
 // Handle B
} catch (C c1) {
 // Handle C
} finally {
 // Stuff that happens every time
}
```

To demonstrate to yourself that the **finally** clause always runs, try this program:

```
//: FinallyWorks.java
// The finally clause is always executed

public class FinallyWorks {
  static int count = 0;
  public static void main(String args[]) {
    while(true) {
      try {
        // post-increment is zero first time:
        if(count++ == 0)
          throw new Exception();
        System.out.println("No exception");
      } catch(Exception e) {
        System.out.println("Exception thrown");
      } finally {
        System.out.println("in finally clause");
        if(count == 2) break; // out of "while"
      }
    }
  }
} ///:~
```

This program also gives a hint for how you can deal with the fact that exceptions in Java (like exceptions in C++) do not allow you to resume back to where the exception was thrown, as discussed earlier. If you place your **try** block in a loop, you can establish a condition that must be met before continuing the program. You can also add a **static** counter or some other device to allow the loop to try several different approaches before giving up. This way you can build a greater level of robustness into your programs.

The output is:

```
Exception thrown
in finally clause
No exception
in finally clause
```

Whether an exception is thrown or not, the **finally** clause is always executed.

what's "finally" for?

In a language without garbage collection *and* without automatic destructor calls,[5] **finally** is important because it allows the programmer to guarantee the release of memory regardless of what happens in

[5] A destructor is a function that's always called when an object becomes unused. You always know exactly where and when the destructor gets called. C++ has automatic destructor calls, but Delphi's

the **try** block. But Java has garbage collection, so releasing memory is virtually never a problem. Also, it has no destructors to call. When do you need to use **finally** in Java, then?

finally is necessary when you need to set something *other* than memory back to its original state. This is usually something like an open file or network connection, something you've drawn on the screen or even a switch in the outside world, as modeled in the following example:

```
//: OnOffSwitch.java
// Why use finally?

class Switch {
  boolean state = false;
  boolean read() { return state; }
  void on() { state = true; }
  void off() { state = false; }
}

public class OnOffSwitch {
  static Switch sw = new Switch();
  public static void main(String args[]) {
    try {
      sw.on();
      // code that may throw exceptions...
      sw.off();
    } catch(NullPointerException e) {
      System.out.println("NullPointerException");
      sw.off();
    } catch(IllegalArgumentException e) {
      System.out.println("IOException");
      sw.off();
    }
  }
} ///:~
```

The goal here is to make sure that the switch is off when **main( )** is completed, so **sw.off( )** is placed at the end of the try block and the end of each exception handler. But it's possible that an exception could be thrown that isn't caught here, and so **sw.off( )** would be missed. However, with **finally** you can place the closure code from a try block in just one place:

```
//: WithFinally.java
// Finally Guarantees cleanup

class Switch2 {
  boolean state = false;
  boolean read() { return state; }
  void on() { state = true; }
  void off() { state = false; }
}

public class WithFinally {
  static Switch2 sw = new Switch2();
  public static void main(String args[]) {
    try {
      sw.on();
      // code that may throw exceptions...
    } catch(NullPointerException e) {
```

Object Pascal versions 1 & 2 do not (which changes the meaning and use of the concept of a destructor for that language).

```
      System.out.println("NullPointerException");
    } catch(IllegalArgumentException e) {
      System.out.println("IOException");
    } finally {
      sw.off();
    }
  }
} ///:~
```

Here the **sw.off( )** has been moved to just one place, where it's guaranteed to run no matter what happens.

Even in cases where the exception is not caught in the current set of **catch** clauses, **finally** will be executed before the exception-handling mechanism continues its search for a handler at the next higher level:

```
//: AlwaysFinally.java
// Finally is always executed

class Ex extends Exception {}

public class AlwaysFinally {
  public static void main(String args[]) {
    System.out.println(
      "Entering first try block");
    try {
      System.out.println(
        "Entering second try block");
      try {
        throw new Ex();
      } finally {
        System.out.println(
          "finally in 2nd try block");
      }
    } catch(Ex e) {
      System.out.println(
        "Caught Ex in first try block");
    } finally {
      System.out.println(
        "finally in 1st try block");
    }
  }
} ///:~
```

The output for this program shows you what happens:

```
Entering first try block
Entering second try block
finally in 2nd try block
Caught Ex in first try block
finally in 1st try block
```

The **finally** statement will also be executed in situations where **break** and **continue** statements are involved. Note that, along with the labeled **break** and labeled **continue**, **finally** eliminates the need for a **goto** statement in Java.

pitfall: the lost exception

In general Java's exception implementation is quite outstanding, but unfortunately there's a flaw. Although exceptions are an indication of a crisis in your program and should never be ignored, it's

possible for an exception to simply be lost. This happens with a particular configuration using a **finally** clause:

```
//: LostMessage.java
// How an exception can be lost

class VeryImportantException extends Exception {
  public String toString() {
    return "A very important exception!";
  }
}

class HoHumException extends Exception {
  public String toString() {
    return "A trivial exception";
  }
}

public class LostMessage {
  void f() throws VeryImportantException {
    throw new VeryImportantException();
  }
  void dispose() throws HoHumException {
    throw new HoHumException();
  }
  public static void main(String args[])
      throws Exception {
    LostMessage lm = new LostMessage();
    try {
      lm.f();
    } finally {
      lm.dispose();
    }
  }
} ///:~
```

The output is:

```
A trivial exception
        at LostMessage.dispose(LostMessage.java:21)
        at LostMessage.main(LostMessage.java:29)
```

You can see that there's no evidence of the **VeryImportantException**, which is simply replaced by the **HoHumException** in the **finally** clause. This is a rather serious pitfall, since it means an exception can be completely lost, and in a far more subtle and difficult-to-detect fashion than the example above. In contrast, C++ treats the situation where a second exception is thrown before the first one is handled as a dire programming error. Perhaps a future version of Java will repair the problem (the above results were produced with Java 1.1).

constructors

When writing code with exceptions, it's particularly important that you always ask: "If an exception occurs, will this be properly cleaned up?" Most of the time you're fairly safe, but in constructors there's a problem. The constructor puts the object into a safe starting state, but it may perform some operation – such as opening a file – that doesn't get cleaned up until the user is finished with the object and calls a special cleanup method. If you throw an exception from inside a constructor, these cleanup behaviors may not occur properly. This means you must be especially diligent while writing your constructor.

Since you've just learned about **finally**, you may think that's the correct solution. But it's not quite that simple because **finally** performs the cleanup code *every time*, even in the situations where we don't want the cleanup code executed until the cleanup method runs. Thus, if you do perform cleanup in **finally**, you must set some kind of flag when the constructor finishes normally and don't do anything in the finally block if the flag is set. Because this isn't particularly elegant (you are coupling your code from one place to another), it's best if you try to avoid performing this kind of cleanup in **finally** unless you are forced to.

In the following example, a class called **InputFile** is created that opens a file and allows you to read it one line (converted into a **String**) at a time. It uses the classes **FileReader** and **BufferedReader** from the Java standard IO library which will be discussed in Chapter 10, but which are simple enough that you probably won't have any trouble understanding their basic use:

```
//: Cleanup.java
// Paying attention to exceptions
// in constructors
import java.io.*;

class InputFile {
  private BufferedReader in;
  InputFile(String fname) throws Exception {
    try {
      in =
        new BufferedReader(
          new FileReader(fname));
      // other code that may throw exceptions
    } catch(FileNotFoundException e) {
      System.out.println(
        "Could not open " + fname);
      // Wasn't open, so don't close it
      throw e;
    } catch(Exception e) {
      // All other exceptions must close it
      try {
        in.close();
      } catch(IOException e2) {
        System.out.println(
          "in.close() unsuccessful");
      }
      throw e;
    } finally {
      // Don't close it here!!!
    }
  }
  String getline() {
    String s;
    try {
      s = in.readLine();
    } catch(IOException e) {
      System.out.println(
        "readLine() unsuccessful");
      s = "failed";
    }
    return s;
  }
  void cleanup() {
    try {
      in.close();
    } catch(IOException e2) {
      System.out.println(
```

```
          "in.close() unsuccessful");
      }
    }
  }

  public class Cleanup {
    public static void main(String args[]) {
      try {
        InputFile in = new InputFile("Cleanup.java");
        String s;
        int i = 1;
        while((s = in.getline()) != null)
          System.out.println(""+ i++ + ": " + s);
        in.cleanup();
      } catch(Exception e) {
        System.out.println(
          "Caught in main, e.printStackTrace()");
        e.printStackTrace();
      }
    }
  } ///:~
```

This example uses Java 1.1 IO classes.

The constructor for **InputFile** takes a **String** argument, which is the name of the file you want to open. Inside a **try** block, it creates a **FileReader** using the file name. A **FileReader** isn't particularly useful until you turn around and use it to create a **BufferedReader** that you can actually talk to – notice that one of the benefits of **InputFile** is that it combines these two actions.

If the **FileReader** constructor is unsuccessful it throws a **FileNotFoundException**, which must be caught separately because that's the one case where you don't want to close the file since it wasn't successfully opened. Any *other* catch clauses must close the file because it *was* opened by the time those catch clauses are entered (of course, this is trickier if more than one method can throw a **FileNotFoundException**. In that case, you may want to break things up into several **try** blocks). The **close( )** method itself throws an exception which is tried and caught even though it's within the block of another **catch** clause – it's just another pair of curly braces to the Java compiler. After performing local operations, the exception is re-thrown, which is appropriate because this constructor failed and you wouldn't want the calling program to assume that the object had been properly created and was valid.

In this example, which doesn't use the aforementioned flagging technique, the **finally** clause is definitely *not* the place to **close( )** the file, since that would close it every time the constructor completed. Since we want the file to be open for the useful lifetime of the **InputFile** object this would not be appropriate.

The **getline( )** method returns a **String** containing the next line in the file. The **readline( )** method that it calls can throw an exception which is caught and dealt with so that **getline( )** doesn't throw any exceptions. One of the design issues with exceptions is whether to handle an exception completely at this level, to handle it partially and pass the same exception (or a different one) on, or whether to simply pass it on. Passing it on, when appropriate, can certainly simplify coding. The **getline( )** method becomes:

```
String getline() throws IOException {
  return in.readLine();
}
```

But of course, the caller is now responsible for handling any **IOException** that might arise.

The **cleanup( )** method must be called by the user when they are finished using the **InputFile** object, to release the system resources (such as file handles) that are used by the **BufferedReader** and/or

FileReader objects[6]. You don't want to do this until you're finished with the **InputFile** object, at the point you're going to let it go. You might think of putting such functionality into a **finalize( )** method, but as mentioned in Chapter 4 you can't always be sure that **finalize( )** will be called (or even if you can be sure that it will be called, you don't know *when*). This is one of the downsides to Java – all cleanup other than memory cleanup doesn't happen automatically, so you must inform the client programmer that they are responsible, and possibly guarantee that cleanup occurs using **finalize( )**.

In **Cleanup.java** an **InputFile** is created to open the same source file that creates the program, and this file is read in a line at a time, and line numbers are added. All exceptions are caught generically in **main( )**, although you could choose greater granularity.

One of the benefits of this example is to show you why exceptions are introduced at this point in the book. Exceptions are so integral to programming in Java, especially because the compiler enforces them, that you can only accomplish so much without knowing how to work with them.

exception matching

When an exception is thrown, the exception-handling system looks through the "nearest" handlers in the order they are written. When it finds a match, the exception is considered handled, and no further searching occurs.

Matching an exception doesn't require a perfect match between the exception and its handler. A derived-class object will match a handler for the base class, as shown in this example:

```
//: Human.java
// Catching Exception Hierarchies

class Annoyance extends Exception {}
class Sneeze extends Annoyance {}

public class Human {
  public static void main(String args[]) {
    try {
      throw new Sneeze();
    } catch(Sneeze d) {
      System.out.println("Caught Sneeze");
    } catch(Annoyance e) {
      System.out.println("Caught Annoyance");
    }
  }
} ///:~
```

The **Sneeze** exception will be caught by the first **catch** clause that it matches, which is the first one, of course. However, if you remove the first catch clause:

```
    try {
      throw new Sneeze();
    } catch(Annoyance e) {
      System.out.println("Caught Annoyance");
    }
```

The remaining catch clause will still work because it's catching the base class of **Sneeze**. Put another way, **catch(Annoyance e)** will catch a **Annoyance** *or any class derived from it*. This is very useful, because it means that if you decide to add more exceptions to a method, if they're all inherited from the same base class then the client programmer's code will not need changing, assuming they catch the base class, at the very least.

[6] In C++, a *destructor* would handle this for you.

If you try to "mask" the derived-class exceptions by putting the base-class catch clause first, like this:

```
try {
  throw new Sneeze();
} catch(Annoyance e) {
  System.out.println("Caught Annoyance");
} catch(Sneeze d) {
  System.out.println("Caught Sneeze");
}
```

The compiler will give you an error message, since it sees that the **Sneeze** catch-clause can never be reached.

exception guidelines

Use exceptions to

1. Fix the problem and call the method (which caused the exception) again.
2. Patch things up and continue without retrying the method.
3. Calculate some alternative result instead of what the method was supposed to produce.
4. Do whatever you can in the current context and rethrow the *same* exception to a higher context.
5. Do whatever you can in the current context and throw a *different* exception to a higher context.
6. Terminate the program.
7. Simplify. If your exception scheme makes things more complicated, then it is painful and annoying to use.
8. Make your library and program safer. This is a short-term investment (for debugging) and a long-term investment (for application robustness).

summary

Improved error recovery is one of the most powerful ways you can increase the robustness of your code. Error recovery is a fundamental concern for every program you write, and it's especially important in Java, where one of the primary goals is to create program components for others to use. To create a robust system, each component must be robust.

The goals for exception handling in Java are to simplify the creation of large, reliable programs using less code than currently possible, with more confidence that your application doesn't have an unhandled error.

Exceptions are not terribly difficult to learn, and are one of those features that provide immediate and significant benefits to your project. Fortunately, Java enforces all aspects of exceptions so it's guaranteed that they will be used consistently, both by library designer and client programmer.

exercises

1. Create a class with a **main( )** that throws an object of class **Exception** inside a **try** block. Give the constructor for **Exception** a string argument. Catch the exception inside a **catch** clause and print out the string argument. Add a **finally** clause and print a message to prove you were there.

2. Create your own exception class using the **extends** keyword. Write a constructor for this class that takes a **String** argument and stores it inside the object with a **String** handle. Write a method that prints out the stored **String**. Create a try-catch clause to exercise your new exception.

3. Write a class with a method that throws an exception of the type created in exercise 2. Try compiling it without an exception specification to see what the compiler says. Add the appropriate exception specification. Try out your class and its exception inside a try-catch clause.

10

10: the Java IO system

Creating a good input/output (IO) system is one of the more difficult tasks for the language designer.

This is evidenced by the number of different approaches taken. The problem seems to be covering all eventualities. Not only are there different kinds of IO that you want to communicate with (files, the console, network connections), but you need to talk to them in a wide variety of ways (sequential, random-access, binary, character, by lines, by words, etc.).

The Java library designers attacked the problem by creating lots of classes. In fact, there are so many classes for Java's IO system that it can be intimidating at first. In addition there has been a significant change in the IO library between Java 1.0 and Java 1.1. But instead of simply replacing the old library with a new one, the designers at JavaSoft extended the old library and added the new one alongside it. As a result you may sometimes end up mixing the old and new libraries and creating even more intimidating code. After a while it becomes familiar and possibly even less offensive, and you might even gain some insight into library design as you ponder how it might have been restructured.

This chapter will help you understand the variety of IO classes in the standard Java library and how to use them. The first portion of the chapter will introduce the "old" Java 1.0 IO stream library, since there is a significant amount of existing code that uses that library, and the remainder of the chapter will introduce the new features in the Java 1.1 IO library. Notice that when you compile some of the code in the first part of the chapter with a Java 1.1 compiler you may get a "deprecated feature" warning message at compile time. The code still works; the compiler is just suggesting that you use certain new features that are described in the latter part of the chapter. It is valuable, however, to see the

difference between the old and new way of doing things and that's why it was left in – to increase your understanding.

input and output

The Java library classes for IO are divided by input and output, as you can see by looking at the online Java class hierarchy with your Web browser. By inheritance, all classes derived from **InputStream** have basic methods called **read( )** for reading a single byte or array of bytes. Likewise, all classes derived from **OutputStream** have basic methods called **write( )** for writing a single byte or array of bytes. However, you won't generally use these methods – they exist so more sophisticated classes can use them as they provide a more useful interface. Thus, you'll rarely create your stream object by using a single class, but instead will string together two or more to provide your desired functionality. The fact that you create more than one object to create a single resulting stream is the primary reason that Java's stream library is confusing.

To gain control it's helpful to categorize the classes by their functionality. The library designers started by deciding that all classes that had anything to do with input would be inherited from **InputStream** and all classes that were associated with output would be inherited from **OutputStream**. On further inspection you'll see this is a bit naïve.

types of InputStream

This category includes the classes that decide where your input is coming from:

1. An array of bytes
2. A **String** object
3. A file
4. A "pipe," which works like a physical pipe: you put things in one end and they come out the other
5. A sequence of other streams, so you can collect them together into a single stream
6. Other sources, such as an Internet connection (this will be discussed in a later chapter).

In addition, the **FilterInputStream** provides a base class for classes that attach (1) attributes or (2) useful interfaces to input streams. This is discussed later.

Table 10-1. Types of InputStream

Class	Function	Constructor Arguments	How to use it
ByteArrayInputStream	Allows a buffer in memory to be used as an **InputStream**.	The buffer from which to extract the bytes.	As a source of data. Connect it to a **FilterInputStream** object to provide a useful interface.
StringBufferInputStream	Converts a **String** into an **InputStream**.	A **String**. The underlying implementation actually uses a **StringBuffer**.	As a source of data. Connect it to a **FilterInputStream** object to provide a useful interface.
FileInputStream	For reading information from a file.	A **String** representing the file name, or a **File** or **FileDescriptor** object.	As a source of data. Connect it to a **FilterInputStream** object to provide a useful interface.

Class	Function	Constructor Arguments	How to use it
PipedInputStream	Produces the data that's being written to the associated **PipedOutputStream**. Implements the "piping" concept.	**PipedOutputStream**	As a source of data in multithreading. Connect it to a **FilterInputStream** object to provide a useful interface.
SequenceInputStream	Coverts two or more **InputStream** objects into a single **InputStream**.	Two **InputStream** objects or an **Enumeration** for a container of **InputStream** objects.	As a source of data. Connect it to a **FilterInputStream** object to provide a useful interface.
FilterInputStream	Abstract class providing an interface for useful functionality to the other **InputStream** classes. See Table 10-3.	See Table 10-3.	See Table 10-3.

types of OutputStream

This category includes the classes that decide where your output will go: an array of bytes (no **String**, however: presumably you can create one yourself using the array of bytes), a file, or a "pipe."

In addition, the **FilterOutputStream** provides a base class for classes that attach (1) attributes or (2) useful interfaces to output streams. This is discussed later.

Table 10-2. Types of OutputStream

Class	Function	Constructor Arguments	How to use it
ByteArrayOutputStream	Creates a buffer in memory. All the data you send to the stream is placed in this buffer.	Optional initial size of the buffer.	To designate the destination of your data. Connect it to a **FilterOutputStream** object to provide a useful interface.
FileOutputStream	For sending information to a file.	A **String** representing the file name, or a **File** or **FileDescriptor** object.	To designate the destination of your data. Connect it to a **FilterOutputStream** object to provide a useful interface.
PipedOutputStream	Any information you write to this automatically ends up as input for the associated **PipedInputStream**. Implements the "piping" concept.	**PipedInputStream**	To designate the destination of your data for multithreading. Connect it to a **FilterOutputStream** object to provide a useful interface.
FilterOutputStream	Abstract class providing an interface for useful functionality to the other **OutputStream** classes. See	See Table 10-4.	See Table 10-4.

Class	Function	Constructor Arguments	How to use it
	Table 10-4.		

adding attributes & useful interfaces

The designers decided that everything should be an **InputStream** if it had anything to do with input, and an **OutputStream** if it had anything to do with output. This sounds like it makes sense, as a first cut design, but with deeper thought (and by referring to the C++ iostream library as an example) they would have seen that there were really two separate issues involved: what kind of device you are talking to (disk, console, memory) and the *way* you want to talk to it (with characters or bytes, random access, formatted, etc.). Then it would have made sense to create two separate class hierarchies, one for each need. The hierarchy dealing with devices wouldn't have any way for the client programmer to talk to a device until they attached an interface class to the device object.

However, the Java IO library doesn't work this way. The device classes contain rudimentary read/write functionality, thus you can easily be confused into thinking that they should be used on their own in some situations. The interface classes were blindly shoehorned into the **InputStream** and **OutputStream** hierarchies, so it's not very clear they actually have a separate use. This is a good lesson; because of this poor design new Java programmers will always have a struggle to figure out:

1. What classes are supposed to be used in what situations. It's not clear you're supposed to mix more than one class of the same type together.

2. Are they foolish for not understanding why the classes are designed this way? Perhaps there's something about class design they don't understand, hidden away in this library. It's pretty hard to know this as a novice, especially since you're bound to assume the Java library designers knew what they were doing.

The classes that provide the convenient interface to control a particular **InputStream** or **OutputStream** are the **FilterInputStream** and **FilterOutputStream** – which are in themselves not very intuitive names. They are derived, respectively, from **InputStream** and **OutputStream**, and they are themselves abstract classes, in theory to provide a common interface for all the different ways you want to talk to a stream. In fact, **FilterInputStream** and **FilterOutputStream** simply mimic their base classes, while the derived classes have wildly different interfaces, another factor to suggest poorly-considered class design (see the summary of this chapter for further analysis).

reading from an InputStream with FilterInputStream

On closer inspection you'll discover that **FilterInputStream** classes also fall into the "grab-bag" category. The description in the online documentation states that this base class is "the basis for enhancing input stream functionality." Translated, this means "we couldn't figure out where else to put this stuff, but it seemed like it belonged together." To be somewhat merciful here, this isn't a bad approach to use as a starting point for library design – as long as, at some point, you perform a design review and discover ideas that really don't hang together. And it's important that you realize what went on here, so you don't mistake this for good library design and emulate it, or feel inadequate because you don't see the design benefits of this library.

The **FilterInputStream** classes accomplish two significantly different things. **DataInputStream** allows you to read different types of primitive data and **String** objects from a stream. This, along with its companion DataOutputStream allow you to portably move primitive data from one place to another via a stream. These "places," are determined by the classes in Table 10-1. If you're reading data in blocks and parsing it yourself you won't need **DataInputStream**, but in most other cases you will want to use it to automatically format the data you read.

The remaining classes modify the way an **InputStream** behaves internally: whether it's buffered or unbuffered, if it keeps track of the lines it's reading (allowing you to ask for line numbers, or set the line number), and whether you can push back a single character. The last two classes look an awful lot like support for building a compiler (that is, they were added to support the construction of the Java compiler), so you probably won't use them in general programming. You'll probably want to buffer your input every time for files and probably console IO, so it might have made more sense to make a special case for unbuffered input rather than buffered input.

Table 10-3. Types of FilterInputStream

Class	Function	Constructor Arguments	How to use it
DataInputStream	Used in concert with **DataOutputStream**, so you can read primitives (int, char, long, etc.) from a stream in a portable fashion.	**InputStream**	Contains a full interface to allow you to read primitive types.
BufferedInputStream	Use this to prevent a physical read every time you want more data. You're saying "use a buffer"	**InputStream**, with optional buffer size.	This doesn't provide an interface *per se*, just a requirement that a buffer be used. Attach an interface object.
LineNumberInputStream	Keeps track of line numbers in the input stream; you can call **getLineNumber( )** and **setLineNumber(int)**.	**InputStream**	This just adds line numbering, so you'll probably attach an interface object.
PushbackInputStream	Has a one-byte push-back buffer so you can push back the last character read.	**InputStream**	Generally used in the scanner for a compiler and probably included because the Java compiler needed it. You probably won't use this.

writing to a OutputStream with FilterOutputStream

The same comments made about **FilterInputStream** being a poorly-designed "grab bag" apply here as well.

The complement to **DataInputStream** is **DataOutputStream**, which formats each of the primitive types and **String** objects onto a stream in such a way that any **DataInputStream**, on any machine, can read them. All the methods start with "write," such as **writeByte( )**, **writeFloat( )**, etc.

If you want to do true formatted output, for example to the console, use a **PrintStream**. This is the endpoint that allows you to print all the primitive data types and **String** objects in a viewable format, as opposed to **DataOutputStream** whose goal is to put them on a stream in a way that **DataInputStream** can portably reconstruct them. The **System.out** static object is a **PrintStream**.

The two important methods in **PrintStream** are **print( )** and **println( )**, which are overloaded to print out all the various types. The difference between **print( )** and **println( )** is the latter adds a newline when it's done.

BufferedOutputStream is a modifier and tells the stream to use buffering so you don't get a physical write every time you write to the stream. You'll probably always want to use this with files, and possibly console IO.

Table 10-4. Types of FilterOutputStream

Class	Function	Constructor Arguments	How to use it
DataOutputStream	Used in concert with **DataInputStream**, so you can write primitives (int, char, long, etc.) to a stream in a portable fashion.	**OutputStream**	Contains full interface to allow you to write primitive types.
PrintStream	For producing formatted output. While **DataOutputStream** handles the *storage* of data, **PrintStream** handles *display*.	**OutputStream**, with optional boolean indicating that the buffer is flushed with every newline.	Should be the "final" wrapping for your **OutputStream** object. You'll probably use this a lot.
BufferedOutputStream	Use this to prevent a physical write every time you send a piece of data. You're saying "use a buffer." You can call **flush( )** to flush the buffer.	**OutputStream**, with optional buffer size.	This doesn't provide an interface *per se*, just a requirement that a buffer is used. Attach an interface object.

off by itself: RandomAccessFile

RandomAccessFile is used for files containing records of known size so you can move from one record to another using **seek( )**, then read or change the records. The records don't all have to be the same size, you just have to be able to determine how big they are and where they are placed in the file.

At first it's a little bit hard to believe this: **RandomAccessFile** is not part of the **InputStream** or **OutputStream** hierarchy. It has no association with those hierarchies other than it happens to implement the **DataInput** and **DataOutput** interfaces (which are also implemented by **DataInputStream** and **DataOutputStream**). It doesn't even use any of the functionality of the existing **InputStream** or **OutputStream** classes – it's a completely separate class, written from scratch, with all its own (mostly native) methods. The value of access to source code is evident here, since we can discover that the bulk of the stream library was written by Arthur van Hoff and Jonathan Payne (with a little help by James Gosling on the piping classes, which one can infer he wanted for some special purpose). But **RandomAccessFile** is the sole contribution made by David Brown, which suggests it may have been created in a vacuum and added much later, with no time to properly fit it into the hierarchy. Or perhaps no one could figure out where it belonged. In any event, it stands alone, as a direct descendant of **Object**.

Essentially, a **RandomAccessFile** works like a **DataInputStream** pasted together with a **DataOutputStream** and the methods **getFilePointer( )** to find out where you are in the file, **seek( )** to move to a new point in the file, and **length( )** to determine the maximum size of the file. In addition, the constructors require a second argument indicating whether you are just randomly reading (**"r"**) or reading and writing (**"rw"**). There's no support for write-only files, which could suggest that **RandomAccessFile** might have worked well if it were inherited from **DataInputStream**.

What's even more frustrating is that you could easily imagine wanting to seek within other types of streams, such as a **ByteArrayInputStream**, but the seeking methods are only available in **RandomAccessFile**, which only works for files. **BufferedInputStream** does allow you to **mark( )** a position (whose value is held in a single internal variable) and **reset( )** to that position, but this is very limited and not too useful.

the File class

The **File** class has a deceiving name – you might think it refers to an actual file, but it doesn't. It can represent either the name of a particular file, or the names of a set of files in a directory. If it's a set of files, you can ask for the set with the **list( )** method, and this returns an array of **String**. It makes sense to return an array rather than one of the flexible container classes because the number of elements is fixed, and if you want a different directory listing you just create a different **File** object.. In fact, "FilePath" would have been a better name. This section shows a complete example of the use of this class, including the associated **FilenameFilter interface**.

a directory lister

Suppose you'd like to see a directory listing. The **File** object can be listed in two ways. If you call **list( )** with no arguments, you'll get the full list that the **File** object contains. However, if you want a restricted list, say for example all the files with an extension of **.java**, then you use a "directory filter" which is a class that tells how to select the **File** objects for display.

Here's the code for the example:

```
//: DirList.java
// Displays directory listing
import java.io.*;

public class DirList {
  public static void main(String args[]) {
    try {
      File path = new File(".");
      String[] list;
      if(args.length == 0)
        list = path.list();
      else
        list = path.list(new DirFilter(args[0]));
      for(int i = 0; i < list.length; i++)
        System.out.println(list[i]);
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
}

class DirFilter implements FilenameFilter {
  String afn;
  DirFilter(String afn) { this.afn = afn; }
  public boolean accept(File dir, String name) {
    // Strip path information:
    String f = new File(name).getName();
    return f.indexOf(afn) != -1;
  }
} ///:~
```

The **DirFilter** class "implements" the **interface FilenameFilter**. Interfaces were covered in Chapter 7. It's useful to see how simple the **FilenameFilter interface** is:

```
public interface FilenameFilter {
  boolean accept(File dir, String name);
}
```

It says that all that this type of object does is provide a method called **accept( )**. The whole reason behind the creation of this class is to provide the **accept( )** method to the **list( )** method, so that **list( )**

can *call back* **accept( )** to determine which file names should be included in the list. Thus this technique is often referred to as a *callback* or sometimes a *functor* (that is, **DirFilter** is a functor because its only job is to hold a method). Because **list( )** takes a **FilenameFilter** object as its argument, it means you can pass an object of any class that implements **FilenameFilter** to choose (even at run-time) how the **list( )** method will behave. Thus, the purpose of a callback is to provide flexibility in the behavior of code.

DirFilter shows that just because an **interface** only contains a set of methods, you're not restricted to only writing those methods (you must at least provide definitions for all the methods in an interface, however). In this case, the **DirFilter** constructor is also created.

The **accept( )** method must accept a **File** object representing the directory that a particular file is found in, and a **String** containing the name of that file. You may choose to use or ignore either of these arguments, but you will probably at least use the file name. Remember that the **list( )** method is calling **accept( )** for each of the file names in the directory object to see which one should be included – this is indicated by the **boolean** result returned by **accept( )**.

To make sure that what you're working with is only the name and contains no path information, all you have to do is take the **String** object and create a **File** object out of it, then call **getName( )** which strips away all the path information. Then **accept( )** uses the **String** class **indexOf( )** method to see if the search string **afn** appears anywhere in the name of the file. If **afn** is found within the string, the return value is the starting index of **afn**, but if it's not found the return value is -1. Keep in mind that this is a simple string search and does not have regular-expression "wildcard" matching like "fo?.b?r*" which is much more difficult to implement.

Note that the **list( )** method returns an array. You can query this array for its length and then move through it, selecting the array elements. This ability to pass an array in and out of a method is a tremendous improvement over the behavior of C and C++.

a sorted directory listing

Ah, you say you want the file names *sorted*? Well, there's no support for sorting in Java so it will have to be added in the program directly. The following code extends the previous example using the classic Quicksort, which is known for its speed:

```
//: SortedDirList.java
// Displays sorted directory listing
import java.io.*;

public class SortedDirList {
  private File path;
  private String[] list;
  public SortedDirList(String afn) {
    path = new File(".");
    if(afn == null)
      list = path.list();
    else
      list = path.list(new DirFilter2(afn));
    sort();
  }
  void print() {
    for(int i = 0; i < list.length; i++)
      System.out.println(list[i]);
  }
  private void sort() {
    quickSort(0, list.length - 1);
  }
  private void quickSort(int left, int right) {
    if(right > left) {
      String s1 = list[right];
      int i = left - 1;
      int j = right;
```

```
      while(true) {
        while(list[++i].toLowerCase().compareTo(
              s1.toLowerCase()) < 0)
          ;
        while(j > 0)
          if(list[--j].toLowerCase().compareTo(
             s1.toLowerCase()) <= 0)
            break; // out of while
        if(i >= j) break;
        swap(i, j);
      }
      swap(i , right);
      quickSort(left, i-1);
      quickSort(i+1, right);
    }
  }
  private void swap(int loc1, int loc2) {
    String tmp = list[loc1];
    list[loc1] = list[loc2];
    list[loc2] = tmp;
  }
  // Test it:
  public static void main(String[] args) {
    SortedDirList sd;
    if(args.length == 0)
      sd = new SortedDirList(null);
    else
      sd = new SortedDirList(args[0]);
    sd.print();
  }
}

class DirFilter2 implements FilenameFilter {
  String afn;
  DirFilter2(String afn) {
    this.afn = afn;
  }
  public boolean accept(File dir, String name) {
    // Strip path information:
    String f = new File(name).getName();
    return f.indexOf(afn) != -1;
  }
} ///:~
```

The **sort( )** method turns around and calls **quickSort( )**, which performs the sorting recursively and uses the **swap( )** method. I leave the details of the Quicksort to an appropriate book on algorithms[1].

A few other improvements have been made: instead of creating **path** and **list** as local variables to **main( )**, they are members of the class so their values can be accessible for the lifetime of the object. In fact, **main( )** is now just a way to test the class. You can see that the constructor of the class automatically sorts the list once that list has been created.

You'll see that the comparisons in the sort are all made after calling the **String.toLowerCase( )** method so that the sort is case-insensitive. This way you don't end up with a list of all the words starting with capital letters, followed by the rest of the words starting with all the lowercase letters.

[1] The best book I've found on this subject is "Practical Algorithms for Programmers," by Andrew Binstock and John Rex (Addison-Wesley 1995). The algorithms are in C, but that doesn't generally cause any trouble in understanding them and implementing them in C++ or Java.

checking for and creating directories

The **File** class is more than just a representation for an existing directory path, file or group of files. You can also use a **File** object to create a new directory or an entire directory path if it doesn't exist. You can also look at the characteristics of files (size, last modification date, read/write), whether a **File** object represents a file or a directory, and you can delete a file. This program shows the remaining methods available with the **File** class:

```
//: MakeDirectories.java
// Demonstrates the use of the File class to
// create directories and manipulate files.
import java.io.*;

public class MakeDirectories {
  private final static String usage =
    "Usage:MakeDirectories path1 ...\n" +
    "Creates each path\n" +
    "Usage:MakeDirectories -d path1 ...\n" +
    "Deletes each path\n" +
    "Usage:MakeDirectories -r path1 path2\n" +
    "Renames from path1 to path2\n";
  private static void usage() {
    System.err.println(usage);
    System.exit(1);
  }
  private static void fileData(File f) {
    System.out.println(
      "Absolute path: " + f.getAbsolutePath() +
      "\n Can read: " + f.canRead() +
      "\n Can write: " + f.canWrite() +
      "\n getName: " + f.getName() +
      "\n getParent: " + f.getParent() +
      "\n getPath: " + f.getPath() +
      "\n length: " + f.length() +
      "\n lastModified: " + f.lastModified());
    if(f.isFile())
      System.out.println("it's a file");
    else if(f.isDirectory())
      System.out.println("it's a directory");
  }
  public static void main(String args[]) {
    if(args.length < 1) usage();
    if(args[0].equals("-r")) {
      if(args.length != 3) usage();
      File
        old = new File(args[1]),
        rname = new File(args[2]);
      old.renameTo(rname);
      fileData(old);
      fileData(rname);
      return; // Exit main
    }
    int count = 0;
    boolean del = false;
    if(args[0].equals("-d")) {
      count++;
      del = true;
    }
    for( ; count < args.length; count++) {
      File f = new File(args[count]);
```

```
      if(f.exists()) {
        System.out.println(f + " exists");
        if(del) {
          System.out.println("deleting..." + f);
          f.delete();
        }
      }
      else { // Doesn't exist
        if(!del) {
          f.mkdirs();
          System.out.println("created " + f);
        }
      }
      fileData(f);
    }
  }
} ///:~
```

In **fileData( )** you can see the various file investigation methods put to use to display information about the file or directory path.

The first method that's exercised by **main( )** is **renameTo( )**, which allows you to rename (or move) a file to an entirely new path represented by the argument which is another **File** object. This also works with directories of any length.

If you experiment with the above program, you'll find you can make a directory path of any complexity because **mkdirs( )** will do all the work for you. In Java 1.0, the **-d** flag reports that the directory is deleted but it's still there; in Java 1.1 the directory is actually deleted.

typical uses of IO streams

Although there are lots of IO stream classes in the library that can be combined in many different ways, there are just a few ways that you'll probably end up using them. However, they require attention to get the right combinations. The following rather long example shows the creation and use of typical configurations of files, so you can use it as a reference when writing your own code. Notice that each configuration begins with a commented number and title that corresponds to the heading for the appropriate explanation that follows in the text.

```
//: IOStreamDemo.java
// Typical IO Stream Configurations
import java.io.*;
import COM.EckelObjects.tools.*;

public class IOStreamDemo {
  public static void main(String args[]) {
    try {
      // 1. Buffered input file
      DataInputStream in =
        new DataInputStream(
          new BufferedInputStream(
            new FileInputStream(args[0])));
      String s, s2 = new String();
      while((s = in.readLine())!= null)
        s2 += s + "\n";
      in.close();

      // 2. Input from memory
      StringBufferInputStream in2 =
          new StringBufferInputStream(s2);
```

```
        int c;
        while((c = in2.read()) != -1)
          System.out.print((char)c);

        // 3. Formatted memory input
        try {
          DataInputStream in3 =
            new DataInputStream(
              new StringBufferInputStream(s2));
          while(true)
            System.out.print((char)in3.readByte());
        } catch(EOFException e) {
          System.out.println(
            "End of stream encountered");
        }

        // 4. Line numbering & file output
        try {
          LineNumberInputStream li =
            new LineNumberInputStream(
              new StringBufferInputStream(s2));
          DataInputStream in4 =
            new DataInputStream(li);
          PrintStream out1 =
            new PrintStream(
              new BufferedOutputStream(
                new FileOutputStream("IODemo.out")));
          while((s = in4.readLine()) != null )
            out1.println(
              "Line " + li.getLineNumber() + s);
          out1.close(); // finalize() not reliable!
        } catch(EOFException e) {
          System.out.println(
            "End of stream encountered");
        }

        // 5. Storing & recovering data
        try {
          DataOutputStream out2 =
            new DataOutputStream(
              new BufferedOutputStream(
                new FileOutputStream("Data.txt")));
          out2.writeBytes(
            "Here's the value of pi: \n");
          out2.writeDouble(3.14159);
          out2.close();
          DataInputStream in5 =
            new DataInputStream(
              new BufferedInputStream(
                new FileInputStream("Data.txt")));
          System.out.println(in5.readLine());
          System.out.println(in5.readDouble());
        } catch(EOFException e) {
          System.out.println(
            "End of stream encountered");
        }

        // 6. Reading and writing random access files
        RandomAccessFile rf =
          new RandomAccessFile("rtest.dat", "rw");
```

```
      for(int i = 0; i < 10; i++)
        rf.writeDouble(i*1.414);
      rf.close();

      rf =
        new RandomAccessFile("rtest.dat", "rw");
      rf.seek(5*8);
      rf.writeDouble(47.0001);
      rf.close();

      rf =
        new RandomAccessFile("rtest.dat", "r");
      for(int i = 0; i < 10; i++)
        System.out.println(
          "Value " + i + ": " +
          rf.readDouble());
      rf.close();

      // 7. File input shorthand
      InFile in6 = new InFile(args[0]);
      String s3 = new String();
      System.out.println(
        "First line in file: " +
        in6.readLine());
      in.close();

      // 8. Formatted file output shorthand
      PrintFile out3 = new PrintFile("Data2.txt");
      out3.print("Test of PrintFile");
      out3.close();

      // 9. Data file output shorthand
      OutFile out4 = new OutFile("Data3.txt");
      out4.writeBytes("Test of outDataFile\n\r");
      out4.writeChars("Test of outDataFile\n\r");
      out4.close();

    } catch(FileNotFoundException e) {
      System.out.println(
        "File Not Found:" + args[1]);
    } catch(IOException e) {
      System.out.println("IO Exception");
    }
  }
} ///:~
```

input streams

Of course, one very common thing you'll want to do is print formatted output to the console, but that's already been simplified in the package **COM.EckelObjects.tools** created in Chapter 5.

Parts 1-4 demonstrate the creation and use of input streams (although part 4 also shows the simple use of an output stream as a testing tool).

1. buffered input file

To open a file for input, you use a **FileInputStream** with a **String** or a **File** object as the file name. For speed, you'll want that file to be buffered so you give the resulting handle to the constructor for a **BufferedInputStream**. To read input in a formatted fashion, you give that resulting handle to the constructor for a **DataInputStream**, which is your final object and the interface you read from.

In this example only the **readLine( )** method is used, but of course any of the **DataInputStream** methods are available. When you reach the end of the file, **readLine( )** returns **null**, so that is used to break out of the **while** loop.

The **String s2** is used to accumulate the entire contents of the file (including newlines which must be added since **readLine( )** strips them off). **s2** is then used in the later portions of this program. Finally, **close( )** is called to close the file. Technically, **close( )** will be called when **finalize( )** is run, and this is supposed to happen (whether or not garbage collection occurs) as the program exits. However, Java 1.0 has a rather important bug so this doesn't happen. In Java 1.1 you must explicitly call **System.runFinalizersOnExit(true)** to guarantee that **finalize( )** will be called for every object in the system. The safest approach is to explicitly call **close( )** for files.

2. input from memory

This piece takes the **String s2** that now contains the entire contents of the file and uses it to create a **StringBufferInputStream** (a **String**, not a **StringBuffer**, is required as the constructor argument). Then **read( )** is used to read each character one at a time and send it out to the console. Note that **read( )** returns the next byte as an **int** and thus it must be cast to a **char** to print properly.

3. formatted memory input

The interface for **StringBufferInputStream** is very limited, so you usually enhance it by wrapping it inside a **DataInputStream**. However, if you choose to read the characters out a byte at a time using **readByte( )**, any value is valid so the return value cannot be used to detect the end of input. Instead, you can use the **available( )** method to find out how many more characters are available. Here's an example that shows how to read a file a byte at a time:

```
//: TestEOF.java
// Testing for the end of file while reading
// a byte at a time.
import java.io.*;

public class TestEOF {
  public static void main(String args[]) {
    try {
      DataInputStream in =
        new DataInputStream(
         new BufferedInputStream(
          new FileInputStream("TestEof.java")));
      while(in.available() != 0)
        System.out.print((char)in.readByte());
    } catch (IOException e) {
      System.err.println("IOException");
    }
  }
} ///:~
```

Note that **available( )** works differently depending on what sort of medium you're reading from – it's literally "the number of bytes that can be read *without blocking*." With a file this means the whole file, but with a different kind of stream this may not be true, so use it thoughtfully.

You could also detect the end of input in cases like these by catching an exception. However, the use of exceptions for control flow is considered a misuse of that feature.

4. line numbering & file output

This example shows the use of the **LineNumberInputStream** to keep track of the input line numbers. Here, you cannot simply gang all the constructors together, since you have to keep a handle to the **LineNumberInputStream** (note this is *not* an inheritance situation, so you cannot simply cast **in4** to a **LineNumberInputStream**). Thus, **li** holds the handle to the **LineNumberInputStream** which is then used to create a **DataInputStream** for easy reading.

LineNumberInputStream may be more useful as an example of how to add functionality to a stream by inheriting a new **FileInputStream** (remember, you have access to the source code for the Java libraries). It's not particularly hard to keep track of line numbers yourself while reading input lines.

This example also shows how to write formatted data to a file. First, a **FileOutputStream** is created to connect to the file. For efficiency, this is made a **BufferedOutputStream**, which is what you'll virtually always want to do, but you're forced to do it explicitly. Then for the formatting it's turned into a **PrintStream**. The data file created this way is readable as an ordinary text file.

One of the methods that actually indicates when a **DataInputStream** is exhausted is **readLine( )**, which returns **null** when there are no more strings to read. Each line is printed to the file along with its line number, which is acquired through **li**.

You'll see an explicit **close( )** for **out1**, which would make sense *if* the program were to turn around and read the same file again. However, this program ends without ever looking at the file **IODemo.out**. When a program ends, all the **finalize( )** methods for all the objects are supposed to be called, and the **FileOutputStream finalize( )** flushes the buffers and closes the file, so everything should come to a fine finish. Unfortunately, **finalize( )** is not reliable – it does *not* get called in Java 1.0, and is only guaranteed to be called in Java 1.1 if you call **System.runFinalizersOnExit(true)**. Thus, If you don't call **close( )** for all your output files, you may discover they're incomplete.

output streams

The two primary kinds of output streams are separated by the way they write data: one writes it for human consumption, and the other writes it to be re-acquired by a **DataInputStream**. The **RandomAccessFile** stands alone, although its data format is compatible with the **DataInputStream** and **DataOutputStream**.

5. storing & recovering data

A **PrintStream** formats data so it's readable by a human. To output data so that it can be recovered by another stream, you use a **DataOutputStream** to write the data, and a **DataInputStream** to recover the data. Of course these streams could be anything, but here a file is used, buffered for both reading and writing.

Note that the character string is written using **writeBytes( )** and not **writeChars( )**. If you use the latter, you'll be writing the 16-bit Unicode characters. Since there is no complementary "readChars" method in **DataInputStream**, you're stuck pulling these characters off one at a time with **readChar( )**. So for ASCII, it's easier to write the characters as bytes; then **readLine( )** gets a regular ASCII line.

The **writeDouble( )** stores the **double** number to the stream and the complementary **readDouble( )** recovers it. But for any of the reading methods to work correctly, you must know the exact placement of the data item in the stream, since it would be equally possible to read the stored **double** as a simple sequence of bytes, or as a **char**, etc. So you must either have a fixed format for the data in the file or extra information must be stored in the file that you parse to determine where the data is located.

6. reading and writing random access files

As previously noted, the **RandomAccessFile** is almost totally isolated from the rest of the IO hierarchy, save for the fact that it implements the **DataInput** and **DataOutput** interfaces. Thus you cannot combine it with any of the aspects of the **InputStream** and **OutputStream** subclasses. Even though it might make sense to treat a **ByteArrayInputStream** as a random-access element, you can only open a file with a **RandomAccessFile**. You must assume a **RandomAccessFile** is properly buffered, since you cannot add that. Again this is an indicator of poor design.

The one option you have is in the second constructor argument: you can open a **RandomAccessFile** to read (**"r"**) or read and write (**"rw"**).

Using a **RandomAccessFile** is like using a combined **DataInputStream** and **DataOutputStream** (because it implements the equivalent interfaces). In addition, you can see that **seek( )** is used to move about in the file and change one of the values.

shorthand for file manipulation

Since there are certain canonical forms that you'll be using very regularly with files, why do all that typing? This portion shows the creation and use of shorthand versions of typical file reading and writing configurations. These shorthands are placed in the **package** COM.EckelObjects.tools that was begun in Chapter 5 (See page 126). To add each class to the library, you simply place it in the appropriate directory and add the **package** statement.

7. file input shorthand

The creation of a file that's buffered and can be read from to as a **DataInputStream** can be encapsulated into a class called **InFile**:

```
//: InFile.java
// Shorthand class for opening an input file
package COM.EckelObjects.tools;
import java.io.*;

public class InFile extends DataInputStream {
  public InFile(String filename)
    throws FileNotFoundException {
    super(
      new BufferedInputStream(
        new FileInputStream(filename)));
  }
  public InFile(File file)
    throws FileNotFoundException {
    this(file.getPath());
  }
} ///:~
```

Both the **String** versions of the constructor and the **File** versions are included, to parallel the creation of a **FileInputStream**.

Now you can reduce your chances of repetitive-stress syndrome while creating files, as seen in the example.

8. formatted file output shorthand

The same kind of approach can be taken to create a **PrintStream** that writes to a buffered file. Here's the extension to COM.EckelObjects.tools:

```
//: PrintFile.java
// Shorthand class for opening an output file
// for formatted printing.
package COM.EckelObjects.tools;
import java.io.*;

public class PrintFile extends PrintStream {
  public PrintFile(String filename)
    throws IOException {
    super(
      new BufferedOutputStream(
        new FileOutputStream(filename)));
  }
  public PrintFile(File file)
    throws IOException {
    this(file.getPath());
  }
} ///:~
```

Note that it is not possible for a constructor to catch an exception that's thrown by a base-class constructor.

9. data file output shorthand

Finally, the same kind of shorthand can create a buffered output file for data storage (as opposed to human-readable storage):

```
//: OutFile.java
// Shorthand class for opening an output file
// for formatted printing.
package COM.EckelObjects.tools;
import java.io.*;

public class OutFile extends DataOutputStream {
  public OutFile(String filename)
    throws IOException {
    super(
      new BufferedOutputStream(
        new FileOutputStream(filename)));
  }
  public OutFile(File file)
    throws IOException {
    this(file.getPath());
  }
} ///:~
```

It is curious (and unfortunate) that the Java library designers didn't think to provide these conveniences as part of their standard.

reading from standard input

Following the approach pioneered in Unix of "standard input," "standard output," and "standard error output," Java has **System.in**, **System.out** and **System.err**. Throughout the book you've seen how to write to standard output, using **System.out** which is already pre-wrapped as a **PrintStream** object. **System.err** is likewise a **PrintStream**, but **System.in** is a raw **InputStream**, with no wrapping. This means that, while you can use **System.out** and **System.err** right away, **System.in** must be wrapped before you can read from it.

Typically you'll want to read input a line at a time using **readLine( )**, so you'll want to wrap **System.in** in a **DataInputStream**. This is the "old" Java 1.0 way to do line input; a bit later in the chapter you'll see the new Java 1.1 solution. Here's an example that simply echoes each line that you type in:

```
//: Echo.java
// How to read from standard input
import java.io.*;

public class Echo {
  public static void main(String args[]) {
    DataInputStream in =
      new DataInputStream(
        new BufferedInputStream(System.in));
    String s;
    try {
      while((s = in.readLine()).length() != 0)
        System.out.println(s);
      // An empty line terminates the program
    } catch(IOException e) {
      e.printStackTrace();
    }
  }
} ///:~
```

The reason for the **try** block is that **readLine( )** can throw an **IOException**. Note that **System.in** should also be buffered, as with most streams

It's a bit inconvenient that you're forced to wrap **System.in** in a **DataInputStream** in each program, but perhaps it was designed this way to allow maximum flexibility.

piped streams

The **PipedInputStream** and **PipedOutputStream** have only been mentioned briefly in this chapter. This is not to suggest that they aren't useful, but their value is not apparent until you begin to understand about multithreading, since the piped streams are used to communicate between threads. This is covered along with an example in Chapter 14.

StreamTokenizer

Although **StreamTokenizer** is not derived from **InputStream** or **OutputStream**, it only works with **InputStream** objects so it rightfully belongs in the IO portion of the library.

The **StreamTokenizer** class is used to break any **InputStream** into a sequence of "tokens," which are bits of text delimited by whatever you choose. For example, your tokens could be words, and then they would be delimited by white space and punctuation.

Consider a program to count the occurrence of words in a text file:

```
//: SortedWordCount.java
// Counts words in a file, outputs
// results in sorted form.
import java.io.*;
import java.util.*;
import c08.*; // Contains StrSortVector

class Counter {
  private int i = 1;
  int read() { return i; }
  void increment() { i++; }
}

public class SortedWordCount {
  private FileInputStream file;
  private StreamTokenizer st;
  private Hashtable counts = new Hashtable();
  SortedWordCount(String filename)
    throws FileNotFoundException {
    try {
      file = new FileInputStream(filename);
      st = new StreamTokenizer(file);
      st.ordinaryChar('.');
      st.ordinaryChar('-');
    } catch(FileNotFoundException e) {
      System.out.println(
        "Could not open " + filename);
      throw e;
    }
  }
  void cleanup() {
    try {
      file.close();
    } catch(IOException e) {
      System.out.println(
```

```
        "st.close() unsuccessful");
    }
  }
  void countWords() {
    try {
      while(st.nextToken() !=
            StreamTokenizer.TT_EOF) {
        String s;
        switch(st.ttype) {
          case StreamTokenizer.TT_EOL:
            s = new String("EOL");
            break;
          case StreamTokenizer.TT_NUMBER:
            s = Double.toString(st.nval);
            break;
          case StreamTokenizer.TT_WORD:
            s = new String(st.sval);
            break;
          default: // single character in ttype
            s = String.valueOf((char)st.ttype);
        }
        if(counts.containsKey(s))
          ((Counter)counts.get(s)).increment();
        else
          counts.put(s, new Counter());
      }
    } catch(IOException e) {
      System.out.println(
        "nextToken() unsuccessful");
    }
  }
  Enumeration values() {
    return counts.elements();
  }
  Enumeration keys() { return counts.keys(); }
  Counter getCounter(String s) {
    return (Counter)counts.get(s);
  }
  Enumeration sortedKeys() {
    Enumeration e = counts.keys();
    StrSortVector sv = new StrSortVector();
    while(e.hasMoreElements())
      sv.addElement((String)e.nextElement());
    sv.sort();
    return sv.elements();
  }
  public static void main(String args[]) {
    try {
      SortedWordCount wc =
        new SortedWordCount(args[0]);
      wc.countWords();
      Enumeration keys = wc.sortedKeys();
      while(keys.hasMoreElements()) {
        String key = (String)keys.nextElement();
        System.out.println(key + ": "
                 + wc.getCounter(key).read());
      }
      wc.cleanup();
    } catch(Exception e) {
      e.printStackTrace();
```

```
    }
  }
} ///:~
```

It makes sense to present these in a sorted form, but since Java doesn't have any sorting methods, that will have to be mixed in. This is easy enough to do with a **StrSortVector** (which was created in Chapter 8, and is part of the package created in that chapter – remember that the starting directory for all the subdirectories in this book must be in your class path for the program to compile successfully).

To open the file, a **FileInputStream** is used, and to turn the file into words a **StreamTokenizer** is created from the **FileInputStream**. In **StreamTokenizer**, there is a default list of separators, and you can add more with a set of methods. Here, **ordinaryChar( )** is used to say: "this character has no significance that I'm interested in," so the parser doesn't include it as part of any of the words it creates. You can find more information in the on-line documentation that comes with Java.

In **countWords( )**, the tokens are pulled one at a time from the stream, and the **ttype** information is used to determine what to do with each token, since a token can be an end-of-line, a number, a string, or a single character.

Once a token is found, the **Hashtable counts** is queried to see if it already contains the token as a key. If it does, the corresponding **Counter** object is incremented to indicate that another instance of this word has been found. If not, a new **Counter** is created – since the **Counter** constructor initializes its value to one, this also acts to count the word.

SortedWordCount is not a type of **Hashtable**, so it wasn't inherited. It performs a specific type of functionality, so even though the **keys( )** and **values( )** methods must be re-exposed, that still doesn't mean that inheritance should be used since there are a number of **Hashtable** methods that are inappropriate here. In addition, other methods **getCounter( )**, which get the **Counter** for a particular **String**, and **sortedKeys( )**, which produces an **Enumeration**, finish the change in the shape of **SortedWordCount**'s interface.

In **main( )** you can see the use of a **SortedWordCount** to open and count the words in a file – it just takes two lines of code. Then an enumeration to a sorted list of keys (words) is extracted, and this is used to pull out each key and associated **Count**. Note that in this case, **cleanup( )** is not necessary, since upon exiting the program the file will be closed. However, you can't normally assume that the **SortedWordCount** object is being called from **main( )**.

A second example using **StreamTokenizer** can be found in Chapter 17.

StringTokenizer

Although it isn't part of the IO library, the **StringTokenizer** has sufficiently similar functionality to **StreamTokenizer** that it will be described here.

The **StringTokenizer** returns, one at a time, the tokens within a string. These tokens are consecutive characters delimited by tabs, spaces, and newlines. Thus the tokens of the string "Where is my cat?" are "Where", "is", "my", and "cat?" Unlike the **StreamTokenizer**, however, you *cannot* tell the **StringTokenizer** to break up the input in any way that you want. It's very limited, and you can only use the rules that it has hard-wired into it. Thus, if it's a very simple tokenization you need, **StringTokenizer** is fine, but if you need more sophistication you'll have to use a **StreamTokenizer**.

You ask a **StringTokenizer** object for the next token in the string using the **next( )** method, which either returns the token or an empty string to indicate that no tokens remain.

As an example, the following program performs a very limited analysis of a sentence, looking for key phrase sequences to indicate whether happiness or sadness is implied.

```
//: AnalyzeSentence.java
// Look for particular sequences
// within sentences.
import java.util.*;
```

```
public class AnalyzeSentence {
  public static void main(String args[]) {
    analyze("I am happy about this");
    analyze("I am not happy about this");
    analyze("I am not! I am happy");
    analyze("I am sad about this");
    analyze("I am not sad about this");
    analyze("I am not! I am sad");
    analyze("Are you happy about this?");
    analyze("Are you sad about this?");
    analyze("It's you! I am happy");
    analyze("It's you! I am sad");
  }
  static StringTokenizer st;
  static void analyze(String s) {
    prt("\nnew sentence >> " + s);
    boolean sad = false;
    st = new StringTokenizer(s);
    while (st.hasMoreTokens()) {
      String token = next();
      // Look until you find one of the
      // two starting tokens:
      if(!token.equals("I") &&
         !token.equals("Are"))
        continue; // Top of while loop
      if(token.equals("I")) {
        String tk2 = next();
        if(!tk2.equals("am")) // Must be after I
          break; // Out of while loop
        else {
          String tk3 = next();
          if(tk3.equals("sad")) {
            sad = true;
            break; // Out of while loop
          }
          if (tk3.equals("not")) {
            String tk4 = next();
            if(tk4.equals("sad"))
              break; // Leave sad false
            if(tk4.equals("happy")) {
              sad = true;
              break;
            }
          }
        }
      }
      if(token.equals("Are")) {
        String tk2 = next();
        if(!tk2.equals("you"))
          break; // Must be after Are
        String tk3 = next();
        if(tk3.equals("sad"))
          sad = true;
        break; // Out of while loop
      }
    }
    if(sad) prt("Sad detected");
  }
  static String next() {
    if(st.hasMoreTokens()) {
```

```
      String s = st.nextToken();
      prt(s);
      return s;
    }
    else
      return "";
  }
  static void prt(String s) {
    System.out.println(s);
  }
} ///:~
```

For each string being analyzed, a **while** loop is entered and tokens are pulled off the string. Notice the first **if** statement, which says to **continue** (go back to the beginning of the loop and start again) if the token is neither an "I" or an "Are". This means it will get tokens until an "I" or an "Are" is found. You might think to use the **==** instead of the **equals( )** method, but that won't work correctly, since **==** compares handle values while **equals( )** compares contents.

The logic of the rest of the **analyze( )** method is that the pattern that's being searched for is "I am sad" or "I am not happy" or "Are you sad." Without the **break** statement, the code for this would be even messier than it is. You should be aware that a typical parser (this is a very primitive example of one) normally has a table of these tokens and a piece of code that moves through the states in the table as new tokens are read.

You should only think of the **StringTokenizer** as shorthand for a very simple and specific kind of **StreamTokenizer**. However, if you have a **String** that you want to tokenize and **StringTokenizer** is too limited, all you have to do is turn it into a stream with **StringBufferInputStream** and then use that to create a much more powerful **StreamTokenizer**.

Java 1.1 IO streams

At this point you may be scratching your head, wondering "is there another design for IO streams that could possibly require *more* typing to create a new stream? Could someone have come up with an odder design?" Prepare yourself: Java 1.1 makes some significant modifications to the IO stream library. When you see the **Reader** and **Writer** classes your first thought (like mine) might be that these were meant to replace the **InputStream** and **OutputStream** classes. But that's not the case. Although some aspects of the original streams library are deprecated (if you use them you will receive a warning from the compiler), the old streams have been left in for backwards compatibility and:

1. New classes have been put into the old hierarchy, so it's obvious that Javasoft is not abandoning the old streams.

2. There are times when you're supposed to use classes in the old hierarchy *in combination* with classes in the new hierarchy, and to accomplish this there are "bridge" classes: **InputStreamReader** converts an **InputStream** to a **Reader** and **OutputStreamWriter** converts an **OutputStream** to a **Writer**.

As a result there are situations where you actually have *more* layers of wrapping with the new IO stream library than the old. It's fairly apparent that someone at Javasoft is quite fond of this design direction so we'll all have to get used to it.

The most important reason for adding the **Reader** and **Writer** hierarchies in Java 1.1 is for internationalization. The old IO stream hierarchy only supports 8-bit byte streams, and doesn't handle the 16-bit Unicode characters very well. Since Unicode is used for internationalization (and Java's native **char** is 16-bit Unicode), the **Reader** and **Writer** hierarchies were added to support Unicode in all IO operations. In addition, the new libraries are designed for faster operations than the old.

As is the practice in this book, I will attempt to provide an overview of the classes but assume that you will use online documentation to determine all the details such as the exhaustive list of methods.

sources and sinks of data

Almost all the Java 1.0 IO stream classes have corresponding Java 1.1 classes to provide native Unicode manipulation. It would be easiest to say: "always use the new classes, never use the old ones" but things are not that simple. Sometimes you are forced into using the Java 1.0 IO stream classes because of the library design; in particular the **java.util.zip** libraries are new additions to the old stream library and they rely on old stream components. So the most sensible approach to take is to *try* to use the **Reader** and **Writer** classes whenever you can, and you'll find out about the situations when you have to drop back into the old libraries because your code won't compile.

Here is a table that shows the correspondence between the sources and sinks of information (that is, where the data physically comes from or goes to) in the old and new libraries.

Sources & Sinks: Java 1.0 class	Corresponding Java 1.1 class
InputStream	**Reader** converter: **InputStreamReader**
OutputStream	**Writer** converter: **OutputStreamWriter**
FileInputStream	**FileReader**
FileOutputStream	**FileWriter**
StringBufferInputStream	**StringReader**
(no corresponding class)	**StringWriter**
ByteArrayInputStream	**CharArrayReader**
ByteArrayOutputStream	**CharArrayWriter**
PipedInputStream	**PipedReader**
PipedOutputStream	**PipedWriter**

In general you'll find that the interfaces in the old library components and the new ones are similar if not identical.

modifying stream behavior

In Java 1.0, streams were adapted for particular needs using subclasses of **FilterInputStream** and **FilterOutputStream** (in "design patterns" parlance, described later in the book, these would be called *adapters*, although this particular design shows a poor understanding of that pattern). Java 1.1 IO streams continues the use of this idea, but the model of deriving all the adapters from the same base class is not followed, which can make it a bit confusing if you're trying to understand it by looking at the class hierarchy.

In the following table, the correspondence is a rougher approximation than in the previous table. The difference is because of the class organization: while **BufferedOutputStream** is a subclass of **FilterOutputStream**, **BufferedWriter** is *not* a subclass of **FilterWriter** (which, even though it is **abstract**, has no subclasses and so appears to have been put in either as a placeholder or simply so you wouldn't wonder where it was). However, the interfaces to the classes themselves are quite a close match and it's apparent that you're supposed to use the new versions instead of the old whenever possible (that is, except in cases where you're forced to produce a **Stream** instead of a **Reader** or **Writer**).

Filters: Java 1.0 class	Corresponding Java 1.1 class
FilterInputStream	**FilterReader**
FilterOutputStream	**FilterWriter** (**abstract** class with no subclasses)
BufferedInputStream	**BufferedReader** (also has **readLine( )**)
BufferedOutputStream	**BufferedWriter**
DataInputStream	use **DataInputStream** (Except when you need to use **readLine( )**, when you should use a **BufferedReader**)
PrintStream	**PrintWriter**
LineNumberInputStream	**LineNumberReader**
StreamTokenizer	**StreamTokenizer** (use constructor that takes a **Reader** instead)
PushBackInputStream	**PushBackReader**

There's one direction that's quite clear: whenever you want to use **readLine( )**, you shouldn't do it with a **DataInputStream** any more (this is met with a deprecation message at compile time), but instead use a **BufferedReader**. Other than this, **DataInputStream** is still a "preferred" member of the new Java 1.1 IO library.

To make the transition to using a **PrintWriter** easier, it has constructors that take any **OutputStream** object. However, **PrintWriter** itself has no more support for formatting that **PrintStream** does; the interfaces are virtually the same.

unchanged classes

Apparently, the Javasoft library designers felt that they got some of the classes right the first time so there were no changes to these and you can go on using them as they are:

Java 1.0 classes w/o corresponding Java 1.1 classes
DataOutputStream
File
RandomAccessFile
SequenceInputStream

The **DataOutputStream**, in particular, is used without change, so for storing and retrieving data in a transportable format you're forced to stay in the **InputStream** and **OutputStream** hierarchies.

an example

To see the effect of the new classes, let's look at the appropriate portion of the **IOStreamDemo.java** example modified to use the **Reader** and **Writer** classes:

```
//: NewIODemo.java
// Java 1.1 IO typical usage
import java.io.*;

public class NewIODemo {
  public static void main(String args[]) {
    try {
      // 1. Reading input by lines:
      BufferedReader in =
        new BufferedReader(
          new FileReader(args[0]));
      String s, s2 = new String();
      while((s = in.readLine())!= null)
        s2 += s + "\n";
      in.close();

      // 1b. Reading standard input:
      BufferedReader stdin =
        new BufferedReader(
          new InputStreamReader(System.in));
      System.out.print("Enter a line:");
      System.out.println(stdin.readLine());

      // 2. Input from memory
      StringReader in2 = new StringReader(s2);
      int c;
      while((c = in2.read()) != -1)
        System.out.println((char)c);

      // 3. Formatted memory input
      try {
        DataInputStream in3 =
          new DataInputStream(
            // Oops: must use deprecated class:
            new StringBufferInputStream(s2));
        while(true)
          System.out.print((char)in3.readByte());
      } catch(EOFException e) {
        System.out.println("End of stream");
      }

      // 4. Line numbering & file output
      try {
        LineNumberReader li =
          new LineNumberReader(
            new StringReader(s2));
        BufferedReader in4 =
          new BufferedReader(li);
        PrintWriter out1 =
          new PrintWriter(
            new BufferedWriter(
              new FileWriter("IODemo.out")));
        while((s = in4.readLine()) != null )
          out1.println(
            "Line " + li.getLineNumber() + s);
```

```
        out1.close();
      } catch(EOFException e) {
        System.out.println("End of stream");
      }

      // 5. Storing & recovering data
      try {
        DataOutputStream out2 =
          new DataOutputStream(
            new BufferedOutputStream(
              new FileOutputStream("Data.txt")));
        out2.writeBytes(
          "Here's the value of pi: \n");
        out2.writeDouble(3.14159);
        out2.close();
        DataInputStream in5 =
          new DataInputStream(
            new BufferedInputStream(
              new FileInputStream("Data.txt")));
        BufferedReader in5br =
          new BufferedReader(
            new InputStreamReader(in5));
        // Can now use the "proper" readLine():
        System.out.println(in5br.readLine());
        // Must use DataInputStream elswhere:
        System.out.println(in5.readDouble());
      } catch(EOFException e) {
        System.out.println("End of stream");
      }

      // 6. Reading and writing random access
      // files is the same as before.

    } catch(FileNotFoundException e) {
      System.out.println(
        "File Not Found:" + args[1]);
    } catch(IOException e) {
      System.out.println("IO Exception");
    }
  }
} ///:~
```

In general, you'll see that the conversion is fairly straightforward and the code looks quite similar. There are some important differences, though.

Section 1 actually shrinks a bit since, if all you're doing is reading line input you only need to wrap a **BufferedReader** around a **FileReader**. Section 1b shows the new way to wrap **System.in** for reading line input, and this expands because **System.in** is a **DataInputStream** and **BufferedReader** needs a **Reader** argument, so **InputStreamReader** is brought in to perform the translation.

In section 2 you can see that if you have a **String** and want to read from it you just use a **StringReader** instead of a **StringBufferInputStream** and the rest of the code is identical.

Section 3 shows a bug in the design of the new IO stream library. If you have a **String** and you want to read from it, you're *not* supposed to use a **StringBufferInputStream** any more. When you compile code involving a **StringBufferInputStream** constructor, you get a deprecation message telling you not to use it. Instead you're supposed to use a **StringReader**. However, if you want to do formatted memory input as in Section 3, you're forced to use a **DataInputStream** – there is no "DataInputReader" to replace it – and a **DataInputStream** constructor requires an **InputStream** argument. Thus you have no choice but to use the deprecated **StringBufferInputStream** class. The compiler will give you a deprecation message but there's nothing you can do about it.

Section 4 is a reasonably straightforward translation from the old streams to the new, with no surprises. In section 5, you're forced to use all the old streams classes because **DataOutputStream** and **DataInputStream** require them and there are no alternatives. However, you don't get any deprecation messages at compile time. If a stream is deprecated, typically its constructor produces a deprecation message to prevent you from using the entire class, but in the case of **DataInputStream** only the **readLine( )** method is deprecated since you're supposed to use a **BufferedReader** for **readLine( )** (but a **DataInputStream** for all other formatted input).

Since random access files have not changed, Section 6 is not repeated.

redirecting standard IO

Java 1.1 has added methods in class **System** that allow you to redirect the standard input, output and error IO streams using simple static method calls:

setIn(InputStream)
setOut(PrintStream)
setErr(PrintStream)

Redirecting output is especially useful if you suddenly start creating a large amount of output on your screen and it's scrolling past faster than you can read it. Redirecting input is valuable for a command-line program when you want to test a particular user-input sequence again and again. Here's a simple example that shows the use of these methods:

```
//: Redirecting.java
// Demonstrates the use of redirection for
// standard IO in Java 1.1
import java.io.*;

class Redirecting {
  public static void main(String args[]) {
    try {
      BufferedInputStream in =
        new BufferedInputStream(
          new FileInputStream(
            "Redirecting.java"));
      // Produces deprecation message:
      PrintStream out =
        new PrintStream(
          new BufferedOutputStream(
            new FileOutputStream("test.out")));
      System.setIn(in);
      System.setOut(out);
      System.setErr(out);

      BufferedReader br =
        new BufferedReader(
          new InputStreamReader(System.in));
      String s;
      while((s = br.readLine()) != null)
        System.out.println(s);
      out.close(); // Remember this!
    } catch(IOException e) {
      e.printStackTrace();
    }
  }
} ///:~
```

This program simply attaches standard input to a file, and redirects standard output and standard error to another file.

This is another example where a deprecation message is inevitable. The message you can get when compiling with the **-deprecation** flag is:

Note: The constructor java.io.PrintStream(java.io.OutputStream) has been deprecated.

However, both **System.setOut( )** and **System.setErr( )** require a **PrintStream** object as an argument, so you are forced to call the **PrintStream** constructor. You might wonder, if Java 1.1 deprecates the entire **PrintStream** class by deprecating the constructor, why would the library designers, at the same time as they added this deprecation, also add new methods to **System** that required a **PrintStream** rather than a **PrintWriter** which is the new and preferred replacement? It's a mystery.

compression

Java 1.1 has also added some classes to support reading and writing streams in a compressed format. These are simply wrapped around existing IO classes to provide compression functionality.

One aspect of these new Java 1.1 classes stands out: they are not derived from the new **Reader** and **Writer** classes, but instead are part of the **InputStream** and **OutputStream** hierarchies. Thus you may be forced to mix the two types of streams (remember that you can use **InputStreamReader** and **OutputStreamWriter** to provide easy conversion between one type and another).

Java 1.1 Compression class	Function
CheckedInputStream	**getCheckSum( )** produces checksum for any **InputStream** (not just decompression)
CheckedOutputStream	**getCheckSum( )** produces checksum for any **OutputStream** (not just compression)
DeflaterOutputStream	Base class for compression classes
ZipOutputStream	A **DeflaterOutputStream** that compresses data into the Zip file format
GZIPOutputStream	A **DeflaterOutputStream** that compresses data into the GZIP file format
InflaterInputStream	Base class for decompression classes
ZipInputStream	A **DeflaterInputStream** that Decompresses data that has been stored in the Zip file format
GZIPInputStream	A **DeflaterInputStream** that Decompresses data that has been stored in the GZIP file format

Although there are many compression algorithms, these two (Zip and GZIP) are possibly the most commonly used, and as a result there are many tools for reading and writing these formats.

simple compression with GZIP

The GZIP interface is very simple and thus is probably more appropriate when you have a single stream of data you want to compress (rather than a collection of dissimilar pieces of data). Here's an example that compresses a single file:

```
//: GZIPcompress.java
// Uses Java 1.1 GZIP compression to compress
// a file whose name is passed on the command
// line.
import java.io.*;
import java.util.zip.*;

public class GZIPcompress {
  public static void main(String args[]) {
    try {
      BufferedReader in =
        new BufferedReader(
          new FileReader(args[0]));
      BufferedOutputStream out =
        new BufferedOutputStream(
          new GZIPOutputStream(
            new FileOutputStream("test.gz")));
      System.out.println("Writing file");
      int c;
      while((c = in.read()) != -1)
        out.write(c);
      in.close();
      out.close();
      System.out.println("Reading file");
      BufferedReader in2 =
        new BufferedReader(
          new InputStreamReader(
            new GZIPInputStream(
              new FileInputStream("test.gz"))));
      String s;
      while((s = in2.readLine()) != null)
        System.out.println(s);
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
} ///:~
```

The use of the compression classes is straightforward – you simply wrap your output stream in a **GZIPOutputStream** or **ZipOutputStream** and your input stream in a **GZIPInputStream** or **ZipInputStream**. All else is ordinary IO reading and writing. This is, however, a good example of where you're forced to mix the old IO streams with the new: **in** uses the **Reader** classes, whereas **GZIPOutputStream**'s constructor can only accept an **OutputStream** object, not a **Writer** object.

multi-file storage with Zip

The Java 1.1 library supporting the Zip format is much more extensive. With it you can easily store multiple files, and there's even a separate class to make the process of reading a Zip file very easy. The library uses the standard Zip format so it works seamlessly with all the tools currently downloadable on the Internet. In addition, the Zip format is used in the Java 1.1 JAR (Java ARchive) file format, which consists of a single file containing a collection of zipped files along with a "manifest" that describes them. JAR files are used to speed up Web transactions. You can find out more about JAR manifests in the online documentation – there's a utility called **jar** that automatically compresses the files of your choice and creates a manifest.

The following example has the same form as the previous example, but it handles as many command-line arguments as you wish. In addition, it shows the use of the **Checksum** classes to calculate and verify the checksum for the file. There are two **Checksum** types: **Adler32** (which is faster) and **CRC32** (which is slower but slightly more accurate).

```
//: ZipCompress.java
// Uses Java 1.1 Zip compression to compress
// any number of files whose names are passed
// on the command line.
import java.io.*;
import java.util.*;
import java.util.zip.*;

public class ZipCompress {
  public static void main(String args[]) {
    try {
      FileOutputStream f =
        new FileOutputStream("test.zip");
      CheckedOutputStream csum =
        new CheckedOutputStream(
          f, new Adler32());
      ZipOutputStream out =
        new ZipOutputStream(
          new BufferedOutputStream(csum));
      out.setComment("A test of Java Zipping");
      // Can't read the above comment, though
      for(int i = 0; i < args.length; i++) {
        System.out.println(
          "Writing file " + args[i]);
        BufferedReader in =
          new BufferedReader(
            new FileReader(args[i]));
        out.putNextEntry(new ZipEntry(args[i]));
        int c;
        while((c = in.read()) != -1)
          out.write(c);
        in.close();
      }
      out.close();
      // Checksum only valid after the file
      // has been closed!
      System.out.println("Checksum: " +
        csum.getChecksum().getValue());
      // Now extract the files:
      System.out.println("Reading file");
      FileInputStream fi =
         new FileInputStream("test.zip");
      CheckedInputStream csumi =
        new CheckedInputStream(
          fi, new Adler32());
      ZipInputStream in2 =
        new ZipInputStream(
          new BufferedInputStream(csumi));
      ZipEntry ze;
      System.out.println("Checksum: " +
        csumi.getChecksum().getValue());
      while((ze = in2.getNextEntry()) != null) {
        System.out.println("Reading file " + ze);
        int x;
        while((x = in2.read()) != -1)
```

```
            System.out.write(x);
        }
        in2.close();
        // Alternative way to open and read
        // zip files:
        ZipFile zf = new ZipFile("test.zip");
        Enumeration e = zf.entries();
        while(e.hasMoreElements()) {
          ZipEntry ze2 = (ZipEntry)e.nextElement();
          System.out.println("File: " + ze2);
          // ... and extract the data as before
        }
      } catch(Exception e) {
        e.printStackTrace();
      }
    }
} ///:~
```

For each file to add to the archive, you must call **putNextEntry( )** and pass it a **ZipEntry** object. The **ZipEntry** object contains an extensive interface that allows you to get and set all the data available on that particular entry in your Zip file: name, compressed and uncompressed sizes, date, CRC checksum, extra field data, comment, compression method, and whether it's a directory entry. However, even though the Zip format has a way to set a password, this is not supported in Java's Zip library. In addition, although **CheckedInputStream** and **CheckedOutputStream** support both **Adler32** and **CRC32** checksums, the **ZipEntry** class only supports an interface for CRC. This is a restriction of the underlying Zip format, but it may limit you from using the faster **Adler32**.

To extract files, **ZipInputStream** has a **getNextEntry( )** method that returns the next **ZipEntry** if there is one. As a more succinct alternative, you can read the file using a **ZipFile** object, which has a method **entries( )** to return an **Enumeration** to the **ZipEntries**.

In order to read the checksum, you must somehow have access to the associated **Checksum** object. Here, a handle to the **CheckedOutputStream** and **CheckedInputStream** objects is retained, but you could also just hold on to a handle to the **Checksum** object itself.

A baffling method in Zip streams is **setComment( )**. As shown above, you can set a comment when you're writing a file, but there's no way to recover the comment in the **ZipInputStream**. Comments only appear to be fully supported on an entry-by-entry basis via **ZipEntry**.

Of course you are not limited to files when using the **GZIP** or **Zip** libraries – you can compress anything, including data to be sent through a network connection.

object serialization

Java 1.1 has added a very interesting feature called *object serialization* that allows you to take any object that implements the **Serializable** interface and turn it into a sequence of bytes that can later be fully restored into the original object. This is even true across a network, which means that the serialization mechanism automatically compensates for differences in operating systems. That is, you can create an object on a Windows machine, serialize it, and send it across the network to a Unix machine where it will be correctly reconstructed. You don't have to worry about the data representations on the different machines, the byte ordering, or any other details.

By itself, object serialization is interesting because it allows you to implement *lightweight persistence*. Remember that persistence means an object's lifetime is not determined by whether a program is executing or not – the object lives *in between* invocations of the program. By taking a serializable object and writing it to disk, then restoring that object when the program is re-invoked, you're able to produce the effect of persistence. The reason it's called "lightweight" is that you can't simply define an object using some kind of "persistent" keyword and let the system take care of the details (although this may happen in the future).

Object serialization was added to the language to support two major features. Java 1.1's *remote method invocation* (RMI) allows objects that live on other machines to behave as if they live on your local machine. When sending messages to remote objects, object serialization is necessary to transport the arguments and return values. RMI is discussed later, in Chapter 15.

Object serialization is also necessary for Java Beans, introduced in Java 1.1. When a Bean is used its state information is generally configured at design time. This state information must be stored and later recovered when the program is started; object serialization performs this task.

Serializing an object is quite simple, as long as the object implements the **Serializable** interface (this interface is just a flag, and has no methods). In Java 1.1, many standard library classes have been changed so they're serializable, including all the wrappers for the primitive types, all the container classes, and many others. Even **Class** objects can be serialized (see Chapter 11 for the implications of this).

To serialize an object, you create some sort of **OutputStream** object and then wrap it inside an **ObjectOutputStream** object. At this point you only need to call **writeObject( )** and your object is magically serialized and sent to the **OutputStream**. To reverse the process, you wrap an **InputStream** inside an **ObjectInputStream** and call **readObject( )**. What comes back is, as usual, a handle to an upcast **Object**, so you must downcast to set things straight.

A particularly clever aspect of object serialization is that it not only saves an image of your object but it also follows all the handles contained in your object and saves *those* objects, and follows all the handles in each of those objects, etc. This is sometimes referred to as the "web of objects" that a single object may be connected to, and it includes arrays of handles to objects as well as member objects. The process of following all these links is a bit mind-boggling, but object serialization seems to pull it off flawlessly. The following example tests the serialization mechanism by making a "worm" of linked objects, each of which has a link to the next segment in the worm as well as an array of handles to objects of a different class, **Data**:

```
//: Worm.java
// Demonstrates object serialization in Java 1.1
import java.io.*;

class Data implements Serializable {
  private int i;
  Data(int x) { i = x; }
  public String toString() {
    return Integer.toString(i);
  }
}

public class Worm implements Serializable {
  // Generate a random int value:
  private static int r() {
    return (int)(Math.random() * 10);
  }
  private Data[] d = {
    new Data(r()), new Data(r()), new Data(r())
  };
  private Worm next;
  private char c;
  // Value of i == number of segments
  Worm(int i, char x) {
    System.out.println(" Worm constructor: " + i);
    c = x;
    if(--i > 0)
      next = new Worm(i, (char)(x + 1));
  }
  Worm() {
    System.out.println("Default constructor");
```

```
  }
  public String toString() {
    String s = ":" + c + "(";
    for(int i = 0; i < d.length; i++)
      s += d[i].toString();
    s += ")";
    if(next != null)
      s += next.toString();
    return s;
  }
  public static void main(String args[]) {
    Worm w = new Worm(6, 'a');
    System.out.println("w = " + w);
    try {
      ObjectOutputStream out =
        new ObjectOutputStream(
          new FileOutputStream("worm.out"));
      out.writeObject("Worm storage");
      out.writeObject(w);
      out.close(); // Also flushes output
      ObjectInputStream in =
        new ObjectInputStream(
          new FileInputStream("worm.out"));
      String s = (String)in.readObject();
      Worm w2 = (Worm)in.readObject();
      System.out.println(s + ", w2 = " + w2);
    } catch(Exception e) {
      e.printStackTrace();
    }
    try {
      ByteArrayOutputStream bout =
        new ByteArrayOutputStream();
      ObjectOutputStream out =
        new ObjectOutputStream(bout);
      out.writeObject("Worm storage");
      out.writeObject(w);
      out.flush();
      ObjectInputStream in =
        new ObjectInputStream(
          new ByteArrayInputStream(
            bout.toByteArray()));
      String s = (String)in.readObject();
      Worm w3 = (Worm)in.readObject();
      System.out.println(s + ", w3 = " + w3);
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
} ///:~
```

To make things interesting, the array of **Data** objects inside **Worm** are initialized with random numbers (this way you don't suspect the compiler of keeping some kind of meta-information around). Each **Worm** segment is labeled with a **char** that's automatically generated in the process of recursively generating the linked list of **Worm**s. When you create a **Worm**, you tell the constructor how long you want it to be. To make the **next** handle it calls the **Worm** constructor with a length of one less, etc. The last **next** handle is left as **null**, indicating the end of the **Worm**.

The point of all this was to make something reasonably complex that couldn't easily be serialized. The actual act of serializing, however, is quite simple. Once the **ObjectOutputStream** is created from some other stream, **writeObject( )** serializes the object. Notice the call to **writeObject( )** for a **String**,

as well. You can also write all the primitive data types using the same methods as **DataOutputStream** (they share the same interface).

There are two separate **try** blocks that look very similar. The first writes and reads a file and the second, for variety, writes and reads a **ByteArray**. You can read and write an object using serialization to any **DataInputStream** or **DataOutputStream** including, as you shall see in the networking chapter, a network. The output is:

```
Worm constructor: 6
Worm constructor: 5
Worm constructor: 4
Worm constructor: 3
Worm constructor: 2
Worm constructor: 1
w = :a(262):b(100):c(396):d(480):e(316):f(398)
Worm storage, w2 = :a(262):b(100):c(396):d(480):e(316):f(398)
Worm storage, w3 = :a(262):b(100):c(396):d(480):e(316):f(398)
```

You can see that the deserialized object really does contain all the links that were in the original object.

Notice that no constructor, not even the default constructor, is called in the process of deserializing a **Serializable** object. The entire object is restored by recovering data from the **InputStream**.

Object serialization is another new Java 1.1 feature that is not part of the new **Reader** and **Writer** hierarchies, but instead the old **InputStream** and **OutputStream** hierarchies. Thus you may encounter situations where you're forced to mix the two hierarchies.

controlling serialization

As you can see, the default serialization mechanism is trivial to use. But what if you have special needs? Perhaps you have special security issues and you don't want to serialize portions of your object, or perhaps it just doesn't make sense for one sub-object to be serialized if that part needs to be created anew when the object is recovered.

You can control the process of serialization by implementing the **Externalizable** interface instead of the **Serializable** interface. The **Externalizable** interface extends the **Serializable** interface and adds two methods, **writeExternal( )** and **readExternal( )** which are automatically called for your object during serialization and deserialization so that you can perform your special operations.

The following example shows very simple implementations of the **Externalizable** interface methods. Notice that **Blip1** and **Blip2** are nearly identical except for a subtle difference (see if you can discover it by looking at the code):

```
//: Blips.java
// Simple use of Externalizable & a pitfall
import java.io.*;
import java.util.*;

class Blip1 implements Externalizable {
  public Blip1() {
    System.out.println("Blip1 Constructor");
  }
  public void writeExternal(ObjectOutput out)
      throws IOException {
    System.out.println("Blip1.writeExternal");
  }
  public void readExternal(ObjectInput in)
     throws IOException, ClassNotFoundException {
    System.out.println("Blip1.readExternal");
  }
}
```

```
class Blip2 implements Externalizable {
  Blip2() {
    System.out.println("Blip2 Constructor");
  }
  public void writeExternal(ObjectOutput out)
      throws IOException {
    System.out.println("Blip2.writeExternal");
  }
  public void readExternal(ObjectInput in)
     throws IOException, ClassNotFoundException {
    System.out.println("Blip2.readExternal");
  }
}

public class Blips {
  public static void main(String args[]) {
    System.out.println("Constructing objects:");
    Blip1 b1 = new Blip1();
    Blip2 b2 = new Blip2();
    try {
      ObjectOutputStream o =
        new ObjectOutputStream(
          new FileOutputStream("Blips.out"));
      System.out.println("Saving objects:");
      o.writeObject(b1);
      o.writeObject(b2);
      o.close();
      // Now get them back:
      ObjectInputStream in =
        new ObjectInputStream(
          new FileInputStream("Blips.out"));
      System.out.println("Recovering b1:");
      b1 = (Blip1)in.readObject();
      // OOPS! Throws an exception:
//!   System.out.println("Recovering b2:");
//!   b2 = (Blip2)in.readObject();
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
} ///:~
```

The output for this program is:

```
Constructing objects:
Blip1 Constructor
Blip2 Constructor
Saving objects:
Blip1.writeExternal
Blip2.writeExternal
Recovering b1:
Blip1 Constructor
Blip1.readExternal
```

The reason that the **Blip2** object is not recovered is that trying to do so causes an exception. Can you see the difference between **Blip1** and **Blip2**? The constructor for **Blip1** is **public**, while the constructor for **Blip2** is not, and that causes the exception upon recovery. Try making **Blip2**'s constructor **public** and removing the //**!** comments to see the correct results.

When **b1** is recovered, the **Blip1** default constructor is called. This is different from recovering a **Serializable** object, where the object is entirely constructed from its stored bits, with no constructor calls. With an **Externalizable** object, all the normal default construction behavior occurs (including the initializations at the point of field definition), and *then* **readExternal( )** is called. You need to be aware of this – in particular the fact that all the default construction always takes place – to produce the correct behavior in your **Externalizable** objects.

Here's an example that shows what you must do to fully store and retrieve an **Externalizable** object:

```
//: Blip3.java
// Reconstructing an externalizable object
import java.io.*;
import java.util.*;

class Blip3 implements Externalizable {
  int i;
  String s; // No initialization
  public Blip3() {
    System.out.println("Blip3 Constructor");
    // s, i not initialized
  }
  public Blip3(String x, int a) {
    System.out.println("Blip3(String x, int a)");
    s = x;
    i = a;
    // s & i only initialized in non-default
    // constructor.
  }
  public String toString() { return s + i; }
  public void writeExternal(ObjectOutput out)
      throws IOException {
    System.out.println("Blip3.writeExternal");
    // You must do this:
    out.writeObject(s); out.writeInt(i);
  }
  public void readExternal(ObjectInput in)
     throws IOException, ClassNotFoundException {
    System.out.println("Blip3.readExternal");
    // You must do this:
    s = (String)in.readObject();
    i =in.readInt();
  }
  public static void main(String args[]) {
    System.out.println("Constructing objects:");
    Blip3 b3 = new Blip3("A String ", 47);
    System.out.println(b3.toString());
    try {
      ObjectOutputStream o =
        new ObjectOutputStream(
          new FileOutputStream("Blip3.out"));
      System.out.println("Saving object:");
      o.writeObject(b3);
      o.close();
      // Now get it back:
      ObjectInputStream in =
        new ObjectInputStream(
          new FileInputStream("Blip3.out"));
      System.out.println("Recovering b3:");
      b3 = (Blip3)in.readObject();
      System.out.println(b3.toString());
```

```
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
} ///:~
```

The fields **s** and **i** are only initialized in the second constructor, but not in the default constructor. This means that if you don't initialize **s** and **i** in **readExternal**, it will be **null** (since the storage for the object gets wiped to zero in the very first step of object creation). If you comment out the two lines of code following the phrases "You must do this" and run the program, you'll see that when the object is recovered, **s** is **null** and **i** is zero.

Thus, to make things work correctly you must not only write the important data from the object during the **writeExternal( )** method (there is no default behavior that writes any of the member objects for an **Externalizable** object), but you must also recover that data in the **readExternal( )** method. This can be a bit confusing at first because the default construction behavior for an **Externalizable** object can make it seem like some kind of storage and retrieval takes place automatically; it does not.

If you are inheriting from an **Externalizable** object, you'll typically call the base-class versions of **writeExternal( )** and **readExternal( )** to provide proper storage and retrieval of the base-class components.

the transient keyword

When you're controlling serialization, there may be a particular subobject that you don't want Java's serialization mechanism to automatically save and restore. This is commonly the case if that subobject represents sensitive information that you don't want to serialize, such as a password. Even if that information is **private** in the object, once it's serialized it's possible for someone to access it by reading a file or intercepting a network transmission.

One way to prevent sensitive parts of your object from being serialized is to implement your class as **Externalizable**. Then, nothing is automatically serialized and you can explicitly serialize only the necessary parts inside **writeExternal( )**.

If you're working with a **Serializable** object, however, all serialization happens automatically. To control this, you can turn off serialization on a field-by-field basis using the **transient** keyword, which says: "don't bother saving or restoring this – I'll take care of it."

For example, consider a **Login** object that keeps information about a particular login session. Suppose that, once you verify the login, you want to store the data but without the password. The easiest way to do this is by implementing **Serializable** and marking the **password** field as **transient**. Here's what it looks like:

```
//: Logon.java
// Demonstrates the "transient" keyword
import java.io.*;
import java.util.*;

class Logon implements Serializable {
  private Date date = new Date();
  private String username;
  private transient String password;
  Logon(String name, String pwd) {
    username = name;
    password = pwd;
  }
  public String toString() {
    String pwd =
      (password == null) ? "(n/a)" : password;
    return "logon info: \n   " +
      "username: " + username +
      "\n   date: " + date.toString() +
```

```
      "\n   password: " + pwd;
  }
  public static void main(String args[]) {
    Logon a = new Logon("Hulk", "myLittlePony");
    System.out.println( "logon a = " + a);
    try {
      ObjectOutputStream o =
        new ObjectOutputStream(
          new FileOutputStream("Logon.out"));
      o.writeObject(a);
      o.close();
      // Delay:
      int seconds = 5;
      long t = System.currentTimeMillis()
             + seconds * 1000;
      while(System.currentTimeMillis() < t)
        ;
      // Now get them back:
      ObjectInputStream in =
        new ObjectInputStream(
          new FileInputStream("Logon.out"));
      System.out.println(
        "Recovering object at " + new Date());
      a = (Logon)in.readObject();
      System.out.println( "logon a = " + a);
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
} ///:~
```

You can see that the **date** and **username** fields are ordinary (not **transient**), and thus are automatically serialized. However, the **password** is **transient**, and so is not stored to disk; also the serialization mechanism makes no attempt to recover it. The output is:

```
logon a = logon info:
   username: Hulk
   date: Sun Mar 23 18:25:53 PST 1997
   password: myLittlePony
Recovering object at Sun Mar 23 18:25:59 PST 1997
logon a = logon info:
   username: Hulk
   date: Sun Mar 23 18:25:53 PST 1997
   password: (n/a)
```

When the object is recovered, the **password** field is **null**. Notice that **toString( )** must check for a **null** value of **password** because if you simply try to assemble a **String** object using the overloaded '**+**' operator, and that operator encounters a **null** handle, you'll get a **NullPointerException**.

You can also see that the **date** field is stored to and recovered from disk, and not generated anew.

Since **Externalizable** objects do not store any of their fields by default, the **transient** keyword is only for use with **Serializable** objects.

versioning

It's possible you may want to change the version of a serializable class (objects of the original class may be stored in a database, for example). This is supported but you'll probably only do it in special cases, and it requires an extra depth of understanding that we shall not attempt to achieve here. The JDK1.1 HTML documents downloadable from Javasoft (which may be part of your Java package's online documents) cover this topic quite thoroughly.

summary

The Java IO stream library does seem to satisfy the basic requirements: you can perform reading and writing with the console, a file, with a block of memory, or even across the Internet (as you shall see in Chapter 15). It's possible (by inheriting from **InputStream** and **OutputStream**) to create new types of input and output objects. And you can even add a very simple extensibility to the kinds of objects a stream will accept by redefining the **toString( )** method that's automatically called when you pass an object to a method that's expecting a **String** (Java's limited "automatic type conversion").

The use of layered objects to add responsibilities to individual objects dynamically and transparently is referred to as the *decorator* pattern (patterns are covered in a later chapter). Decorators are often used when subclassing is impractical since it would create a large number of subclasses to support every possible combination needed. Since the Java IO library provides a number of these possibilities, the decorator pattern would seem like a good approach. However, the decorator pattern specifies[2] that all objects that wrap around your initial object have the same interface, so the use of the decorators is transparent – you send the same message to an object whether it's been decorated or not. The Java IO "decorators" change the interfaces, sometimes very significantly. The very reason that the Java IO library is awkward to use is that it only has the appearance of following the decorator pattern, but then immediately breaks that design. The Java 1.1 IO library change missed the opportunity to completely change the library design, and instead it has added even more special cases and complexity.

There are other questions left unanswered by the documentation and design of the IO stream library. For example, is it simply not possible to fail when you open an output file? Also, some programming systems allow you to specify that you want to open an output file, but only if it doesn't already exist. In Java it appears you are supposed to use a **File** object to determine whether a file exists, because if you open it as an **FileOutputStream** or **FileWriter** it will always get overwritten. By representing both files and directory paths, the **File** class also suggests poor design by violating the maxim "don't try to do too much in a single class."

The IO stream library brings up mixed feelings. It does much of the job, it's portable, and it's there so you don't have to write it yourself. But the design is poor and non-intuitive so there's extra overhead in learning and teaching it, and it's incomplete: there's no support for the kind of output formatting that almost every other language IO package supports (and this was not remedied in Java 1.1). The Java 1.1 changes to the IO library haven't been replacements, but rather additions, and it seems that the library designers couldn't quite get it straight about which features are deprecated and which are preferred, resulting in annoying deprecation messages that show up the contradictions in the library design.

The IO library is usable, but it's frustrating. At least it might be possible to learn from it, as an example of what *not* to do in your own class designs.

exercises

1. Open a text file so that you can read the file a line at a time. Read each line as a **String** and place that **String** object into a **Vector**. Print out all the lines in the **Vector** in reverse order.

2. Modify exercise one so that the name of the file you read is provided as a command-line argument.

3. Modify exercise two to also open a text file so you can write text into it. Write the lines in the **Vector**, along with line numbers, out to the file.

4. Modify exercise two to force all the lines in the **Vector** to upper case and send the results to **System.out**.

---

[2] In *Design Patterns*, Erich Gamma *et al.*, Addison-Wesley 1995. Described later in this book.

5. Modify exercise two to take additional arguments of words to find in the file. Print out any lines where the words match.

6. In **Blips.java**, copy the file and rename it to **BlipCheck.java** and rename the class **Blip2** to **BlipCheck** (making it **public** in the process). Remove the **//!** marks in the file and execute the program including the offending lines. Next, comment out the default constructor for **BlipCheck**. Run it and explain why it works.

7. In **Blip3.java**, comment out the two lines after the phrases "You must do this:" and run the program. Explain the result and why it differs from when the 2 lines are in the program.

8. Convert the **SortedWordCount.java** program to use the Java 1.1 IO Streams.

11

11: run-time type identification

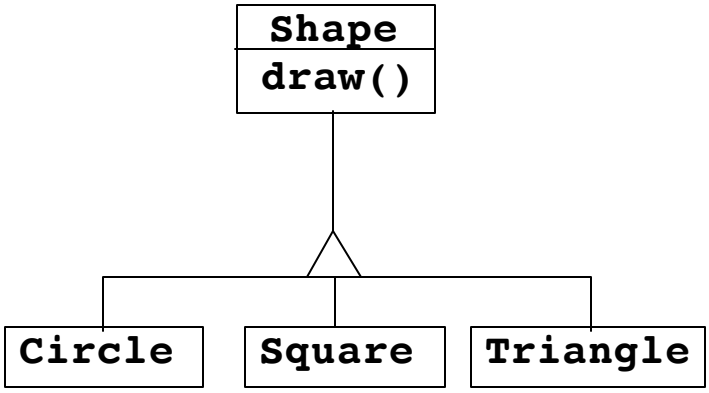
The idea of run-time type identification (RTTI) seems fairly simple at first: it lets you find the exact type of an object when you have only a handle to the base type.

However, the *need* for RTTI uncovers a whole plethora of interesting (and often perplexing) OO design issues, and raises fundamental questions of how you should structure your programs.

This chapter looks at the ways that Java allows you to discover information about objects and classes at run-time. This takes two forms: “traditional” RTTI that assumes you have all the types available at compile-time and run-time, and the “reflection” mechanism in Java 1.1 that allows you to discover class information solely at run-time. The “traditional” RTTI will be covered first, followed by a discussion of reflection.

The need for RTTI

Consider the by-now familiar example of a class hierarchy that uses polymorphism. The generic type is the base class **Shape**, and the specific derived types are **Circle**, **Square**, and **Triangle**:



This is a typical class-hierarchy diagram, with the base class at the top and the derived classes growing downward. The normal goal in object-oriented programming is for the bulk of your code to manipulate handles to the base type (**Shape**, in this case) so if you decide to extend the program by adding a new class (**Rhomboid**, derived from **Shape**, for example), the bulk of the code is not affected. In this example, the dynamically-bound method in the **Shape** interface is **draw( )**, so the intent is for the client programmer to call **draw( )** through a generic **Shape** handle. **draw( )** is redefined in all the derived classes, and because it is a dynamically-bound method, the proper behavior will occur even though it is called through a generic **Shape** handle. That's polymorphism.

Thus, you generally create a specific object (**Circle**, **Square**, or **Triangle**), upcast it to a **Shape** (forgetting the specific type of the object), and use that anonymous **Shape** handle in the rest of the program.

As a brief review of polymorphism and upcasting, you might code the above example as follows:

```
//: Shapes.java
import java.util.*;

interface Shape {
  public void draw();
}

class Circle implements Shape {
  public void draw() {
    System.out.println("Circle.draw()");
  }
}

class Square implements Shape {
  public void draw() {
    System.out.println("Square.draw()");
  }
}

class Triangle implements Shape {
  public void draw() {
    System.out.println("Triangle.draw()");
  }
}
```

```
public class Shapes {
  public static void main(String args[]) {
    Vector s = new Vector();
    s.addElement(new Circle());
    s.addElement(new Square());
    s.addElement(new Triangle());
    Enumeration e = s.elements();
    while(e.hasMoreElements())
      ((Shape)e.nextElement()).draw();
  }
} ///:~
```

The base class could be coded as an **interface**, an **abstract** class or an ordinary class. Since **Shape** has no concrete members (that is, members with definitions) and it's not intended that you ever create a plain **Shape** object, the most appropriate and flexible representation is an **interface**. It's also cleaner because you don't have all those **abstract** keywords lying about.

Each of the derived classes overrides the base-class **draw** method so it behaves differently. In **main( )**, specific types of **Shape** are created and then added to a **Vector**. This is the point where the upcast occurs because the **Vector** only holds **Object**s. Since everything in Java (with the exception of primitives) is an **Object**, a **Vector** can also hold **Shape** objects. But during an upcast to **Object**, it also loses any specific information, including the fact that the objects are **shape**s. To the **Vector**, they are just **Object**s.

At the point you fetch an element out of the **Vector** with **nextElement( )**, things get a little busy. Since **Vector** only holds **Object**s, **nextElement( )** naturally produces an **Object** handle. But we know it's actually a **Shape** handle, and we want to send **Shape** messages to that object. So a cast to **Shape** is necessary using the traditional "**(Shape)**" cast. This is actually the most basic form of RTTI, since in Java all casts are checked at run-time for correctness. That's exactly what RTTI means: at run-time, the type of an object is identified.

In this case, the RTTI cast is only partial: the **Object** is cast to a **Shape**, and not all the way to a **Circle**, **Square** or **Triangle**. That's because the only thing we *know* at this point is that the **Vector** is full of **Shape**s. This is only enforced at compile-time by your own self-imposed rules, but the cast at run-time ensures it.

Now polymorphism takes over and the exact method that's called for the **Shape** is determined by whether the handle is for a **Circle**, **Square** or **Triangle**. And this is how it should be, in general: you want the bulk of your code to know as little as possible about *specific* types of objects, and just deal with the general representation of a family of objects (in this case, **Shape**). As a result, your code will be easier to write, read and maintain and your designs will be easier to implement, understand and change. So polymorphism is the general goal in object-oriented programming.

But what if you have a special programming problem that's easiest to solve if you know the exact type of a generic handle? For example, suppose you want to allow your users to highlight all the shapes of any particular type by turning them purple. This way, they can find all the triangles on the screen by highlighting them. This is what RTTI accomplishes: you can ask a handle to a **Shape** exactly what type it's referring to.

the Class object

To understand how RTTI works in Java, you must first know how type information is represented at run time. This is accomplished through a special kind of object called the *Class object* that contains information about the class itself. In fact, the **Class** object is used to create all the "regular" objects of your class.

There's a **Class** object for each class that is part of your program. That is, each time you write a new class, a single **Class** object is also created (and stored, appropriately enough, in an identically-named **.class** file). At run time, when you want to make an object of that class the Java Virtual Machine (JVM) that's executing your program first checks to see if the **Class** object for that type is loaded. If not, the

JVM loads it by finding the **.class** file with that name. Thus a Java program isn't completely loaded before it begins, which is different than many traditional languages.

Once the **Class** object for that type is in memory, it is used to create all objects of that type.

If this seems shadowy or you don't really believe it, here's a demonstration program to prove it:

```
//: SweetShop.java
// Examination of the way the class loader works

class Candy {
  static {
    System.out.println("Loading Candy");
  }
}

class Gum {
  static {
    System.out.println("Loading Gum");
  }
}

class Cookie {
  static {
    System.out.println("Loading Cookie");
  }
}

public class SweetShop {
  public static void main(String args[]) {
    System.out.println("inside main");
    new Candy();
    System.out.println("After creating Candy");
    try {
      Class.forName("Gum");
    } catch(ClassNotFoundException e) {
      e.printStackTrace();
    }
    System.out.println(
      "After Class.forName(\"Gum\")");
    new Cookie();
    System.out.println("After creating Cookie");
  }
} ///:~
```

Each of the classes **Candy**, **Gum** and **Cookie** has a **static** section which is executed as the class is loaded for the first time. This means information will be printed out to tell you when loading occurs for that class. In **main( )**, the object creations are spread out between print statements to help detect the time of loading.

A particularly interesting line is:

```
Class.forName("Gum");
```

This method is a **static** member of **Class** (to which all Class objects belong). A Class object is like any other object and thus you can get and manipulate a handle to it (that's what the loader does). One of the ways to get a handle to the Class object is **forName( )**, which takes a **String** containing the textual name (watch the spelling and capitalization!) of the particular class you want a handle for. It returns a **Class** handle.

The output of this program for one JVM is:

```
inside main
Loading Candy
After creating Candy
Loading Gum
After Class.forName("Gum")
Loading Cookie
After creating Cookie
```

So you can see that each class object is loaded only as it's needed, and the **static** initialization is performed upon class loading.

Interestingly enough, a different JVM yielded:

```
Loading Candy
Loading Cookie
inside main
After creating Candy
Loading Gum
After Class.forName("Gum")
After creating Cookie
```

It appears that this JVM anticipated the need for **Candy** and **Cookie** by examining the code in **main( )**, but could not see **Gum** because it was created by a call to **forName( )** and not through a more typical call to **new**. While this JVM produces the desired effect because it does get the classes loaded before they're needed, it's uncertain whether the behavior shown is precisely correct. As you'll see later in this chapter, the different ways various JVMs load classes and initialize them could cause surprises.

class literals

In Java 1.1 you have a second way to produce the handle to the **Class** object: with the *class literal*. In the above program this would look like:

```
Gum.class;
```

which is not only simpler, but also safer since it's checked at compile time. Because it eliminates the method call, it's also more efficient.

Class literals work with regular classes as well as interfaces, arrays and primitive types. In addition, there's a standard field called **TYPE** that exists for each of the primitive wrapper classes. The **TYPE** field produces a handle to the **Class** object for the associated primitive type, such that:

… is equivalent to …	
`boolean.class`	`Boolean.TYPE`
`char.class`	`Character.TYPE`
`byte.class`	`Byte.TYPE`
`short.class`	`Short.TYPE`
`int.class`	`Integer.TYPE`
`long.class`	`Long.TYPE`
`float.class`	`Float.TYPE`
`double.class`	`Double.TYPE`
`void.class`	`Void.TYPE`

checking before a cast

So far, you've seen RTTI forms including:

1. The classic cast, e.g. "**(Shape)**" which uses RTTI to make sure the cast is correct and throws a **ClassCastException** if you've performed a bad cast.

2. The **Class** object representing the type of your object. The **Class** object can be queried for useful runtime information.

In C++, the classic cast "**(Shape)**" does *not* perform RTTI. It simply tells the compiler to treat the object as the new type. In Java, which does perform the type check, this cast is often called a "type-safe downcast." The reason for the term "downcast" is the historical arrangement of the class hierarchy diagram. If casting a **Circle** to a **Shape** is an upcast, then casting a **Shape** to a **Circle** is a downcast. However, you know a **Circle** is also a **Shape**, and the compiler freely allows an upcast assignment, but you *don't* know that a **Shape** is necessarily a **Circle**, so the compiler doesn't allow you to perform a downcast assignment without using an explicit cast.

There's a third form of RTTI in Java. This is the keyword **instanceof** which tells you if an object is an instance of a particular type. It returns a **boolean** so you use it in the form of a question, like this:

```
if(x instanceof Dog)
  ((Dog)x).bark();
```

The above **if** statement checks to see if the object **x** belongs to the class **Dog** *before* casting **x** to a **Dog**. It's important to use **instanceof** before a downcast when you don't have other information that tells you the type of the object, otherwise you'll end up with a **ClassCastException**.

Normally you might be hunting for one type (triangles to turn purple, for example), but the following program shows how to tally *all* of the objects using **instanceof**.

```
//: PetCount.java
// Using instanceof
package c11.PetCount;
import java.util.*;

class Pet {}
class Dog extends Pet {}
class Pug extends Dog {}
class Cat extends Pet {}
class Rodent extends Pet {}
class Gerbil extends Rodent {}
class Hamster extends Rodent {}

class Counter { int i; }

public class PetCount {
  static String[] typenames = {
    "Pet", "Dog", "Pug", "Cat",
    "Rodent", "Gerbil", "Hamster",
  };
  public static void main(String args[]) {
    Vector pets = new Vector();
    try {
      Class[] petTypes = {
        Class.forName("Dog"),
        Class.forName("Pug"),
        Class.forName("Cat"),
        Class.forName("Rodent"),
        Class.forName("Gerbil"),
        Class.forName("Hamster"),
      };
```

```
      for(int i = 0; i < 15; i++)
        pets.addElement(
          petTypes[
            (int)(Math.random() * petTypes.length)]
            .newInstance());
    } catch(InstantiationException e) {}
      catch(IllegalAccessException e) {}
      catch(ClassNotFoundException e) {}
    Hashtable h = new Hashtable();
    for(int i = 0; i < typenames.length; i++)
      h.put(typenames[i], new Counter());
    for(int i = 0; i < pets.size(); i++) {
      Object o = pets.elementAt(i);
      if(o instanceof Pet)
        ((Counter)h.get("Pet")).i++;
      if(o instanceof Dog)
        ((Counter)h.get("Dog")).i++;
      if(o instanceof Pug)
        ((Counter)h.get("Pug")).i++;
      if(o instanceof Cat)
        ((Counter)h.get("Cat")).i++;
      if(o instanceof Rodent)
        ((Counter)h.get("Rodent")).i++;
      if(o instanceof Gerbil)
        ((Counter)h.get("Gerbil")).i++;
      if(o instanceof Hamster)
        ((Counter)h.get("Hamster")).i++;
    }
    for(int i = 0; i < pets.size(); i++)
      System.out.println(
        pets.elementAt(i).getClass().toString());
    for(int i = 0; i < typenames.length; i++)
      System.out.println(
        typenames[i] + " quantity: " +
        ((Counter)h.get(typenames[i])).i);
  }
} ///:~
```

There's a rather narrow restriction on **instanceof** in Java 1: you can only compare it to a named type, and not to a **Class** object. In the above example you may feel like it's tedious to write out all those **instanceof** expressions, and you're right. But in Java 1 there's no way to cleverly automate it by creating a **Vector** of **Class** objects and comparing to those instead. This isn't as great a restriction as you might think, because you'll eventually understand that your design is probably flawed if you end up writing a lot of **instanceof** expressions.

Of course this example is contrived — you'd probably put a **static** data member in each type and increment it in the constructor to keep track of the counts. You would do something like that *if* you had control of the source code for the class and could change it. Since this is not always the case, RTTI can come in handy.

using class literals

It's interesting to see how the **PetCount.java** example can be rewritten using Java 1.1 *class literals*. The result is cleaner in many ways:

```
//: PetCount2.java
// Using Java 1.1 class literals
package c11.PetCount2;
import java.util.*;

class Pet {}
```

```
class Dog extends Pet {}
class Pug extends Dog {}
class Cat extends Pet {}
class Rodent extends Pet {}
class Gerbil extends Rodent {}
class Hamster extends Rodent {}

class Counter { int i; }

public class PetCount2 {
  public static void main(String args[]) {
    Vector pets = new Vector();
    Class[] petTypes = {
      // Class literals work in Java 1.1 only:
      Pet.class,
      Dog.class,
      Pug.class,
      Cat.class,
      Rodent.class,
      Gerbil.class,
      Hamster.class,
    };
    try {
      for(int i = 0; i < 15; i++) {
        // Offset by one to eliminate Pet.class:
        int rnd = 1 + (int)(
          Math.random() * (petTypes.length - 1));
        pets.addElement(
          petTypes[rnd].newInstance());
      }
    } catch(InstantiationException e) {}
      catch(IllegalAccessException e) {}
    Hashtable h = new Hashtable();
    for(int i = 0; i < petTypes.length; i++)
      h.put(petTypes[i].toString(),
        new Counter());
    for(int i = 0; i < pets.size(); i++) {
      Object o = pets.elementAt(i);
      if(o instanceof Pet)
        ((Counter)h.get(
          "class c11.PetCount2.Pet")).i++;
      if(o instanceof Dog)
        ((Counter)h.get(
          "class c11.PetCount2.Dog")).i++;
      if(o instanceof Pug)
        ((Counter)h.get(
          "class c11.PetCount2.Pug")).i++;
      if(o instanceof Cat)
        ((Counter)h.get(
          "class c11.PetCount2.Cat")).i++;
      if(o instanceof Rodent)
        ((Counter)h.get(
          "class c11.PetCount2.Rodent")).i++;
      if(o instanceof Gerbil)
        ((Counter)h.get(
          "class c11.PetCount2.Gerbil")).i++;
      if(o instanceof Hamster)
        ((Counter)h.get(
          "class c11.PetCount2.Hamster")).i++;
    }
```

```
    for(int i = 0; i < pets.size(); i++)
      System.out.println(
        pets.elementAt(i).getClass().toString());
    Enumeration keys = h.keys();
    while(keys.hasMoreElements()) {
      String nm = (String)keys.nextElement();
      Counter cnt = (Counter)h.get(nm);
      System.out.println(
        nm.substring(nm.lastIndexOf('.') + 1) +
        " quantity: " + cnt.i);
    }
  }
} ///:~
```

Here, the **typenames** array has been removed in favor of getting the type name strings from the **Class** object. Note the extra work for this: the class name is not, for example, **Gerbil** but instead **c11.PetCount2.Gerbil** since the package name is included. Note also that the system can distinguish between classes and interfaces.

You can also see that the creation of **petTypes** does not need to be surrounded by a **try** block, since it's evaluated at compile time and thus won't throw any exceptions, unlike **Class.forName( )**.

When the **Pet** objects are dynamically created, you can see that the random number is restricted so it is between 1 and **petTypes.length** and does not include zero. That's because zero refers to **Pet.class**, and presumably a generic **Pet** object is not interesting. However, since **Pet.class** is part of **petTypes** the result is that all the pets get counted.

a dynamic instanceof

Java 1.1 has added the **isInstance** method to the class **Class**. This allows you to dynamically call the **instanceof** operator which you could only do statically in Java 1 (as previously shown). Thus, all those tedious **instanceof** statements can be removed in the **PetCount** example:

```
//: PetCount3.java
// Using Java 1.1 isInstance()
package c11.PetCount3;
import java.util.*;

class Pet {}
class Dog extends Pet {}
class Pug extends Dog {}
class Cat extends Pet {}
class Rodent extends Pet {}
class Gerbil extends Rodent {}
class Hamster extends Rodent {}

class Counter { int i; }

public class PetCount3 {
  public static void main(String args[]) {
    Vector pets = new Vector();
    Class[] petTypes = {
      Pet.class,
      Dog.class,
      Pug.class,
      Cat.class,
      Rodent.class,
      Gerbil.class,
      Hamster.class,
    };
    try {
      for(int i = 0; i < 15; i++) {
```

```
        // Offset by one to eliminate Pet.class:
        int rnd = 1 + (int)(
          Math.random() * (petTypes.length - 1));
        pets.addElement(
          petTypes[rnd].newInstance());
      }
    } catch(InstantiationException e) {}
      catch(IllegalAccessException e) {}
    Hashtable h = new Hashtable();
    for(int i = 0; i < petTypes.length; i++)
      h.put(petTypes[i].toString(),
        new Counter());
    for(int i = 0; i < pets.size(); i++) {
      Object o = pets.elementAt(i);
      // Using isInstance to eliminate individual
      // instanceof expressions:
      for (int j = 0; j < petTypes.length; ++j)
        if (petTypes[j].isInstance(o)) {
          String key = petTypes[j].toString();
          ((Counter)h.get(key)).i++;
        }
    }
    for(int i = 0; i < pets.size(); i++)
      System.out.println(
        pets.elementAt(i).getClass().toString());
    Enumeration keys = h.keys();
    while(keys.hasMoreElements()) {
      String nm = (String)keys.nextElement();
      Counter cnt = (Counter)h.get(nm);
      System.out.println(
        nm.substring(nm.lastIndexOf('.') + 1) +
        " quantity: " + cnt.i);
    }
  }
} ///:~
```

You can see that the Java 1.1 **isInstance( )** method has eliminated the need for the **instanceof** expressions. In addition, this means that you can add new types of pets by simply changing the **petTypes** array; the rest of the program does not need modification (as it did when using the **instanceof** expressions).

RTTI syntax

Java performs its RTTI using the Class object, even if you're doing something like a cast. The class **Class** also has a number of other ways you can use RTTI.

First, you must get a handle to the appropriate **Class** object. One way to do this, as shown in the previous example, is to use a string and the **Class.forName( )** method. This is convenient because you don't need an object of that type in order to get the **Class** handle. However, if you do already have an object of the type you're interested in you can fetch the **Class** handle by calling a method that's part of the root class **Object**: **getClass( )**. This returns the **Class** handle representing the actual type of the object. **Class** itself has several interesting and sometimes useful methods, demonstrated in the following example:

```
//: ToyTest.java
// Testing class Class

interface HasBatteries {}
interface Waterproof {}
```

```
interface ShootsThings {}
class Toy {
  // Comment out the following default
  // constructor to see
  // NoSuchMethodError from (*1*)
  Toy() {}
  Toy(int i) {}
}

class FancyToy extends Toy
    implements HasBatteries,
      Waterproof, ShootsThings {
  FancyToy() { super(1); }
}

public class ToyTest {
  public static void main(String args[]) {
    Class c = null;
    try {
      c = Class.forName("FancyToy");
    } catch(ClassNotFoundException e) {}
    printInfo(c);
    Class faces[] = c.getInterfaces();
    for(int i = 0; i < faces.length; i++)
      printInfo(faces[i]);
    Class cy = c.getSuperclass();
    Object o = null;
    try {
      o = cy.newInstance(); // (*1*)
    } catch(InstantiationException e) {}
      catch(IllegalAccessException e) {}
    printInfo(o.getClass());
  }
  static void printInfo(Class cc) {
    System.out.println(
      "Class name: " + cc.getName() +
      " is interface? [" +
      cc.isInterface() + "]");
  }
} ///:~
```

You can see that **class FancyToy** is quite complicated, since it inherits from **Toy** and **implements** the **interface**s of **HasBatteries**, **Waterproof** and **ShootsThings**. In **main( )**, a **Class** handle is created and initialized to the **FancyToy Class** using **forName( )** inside an appropriate **try** block.

The **Class.getInterfaces( )** method returns an array of **Class** representing the interfaces that are contained in the **Class** object of interest.

If you have a **Class** object you can also ask it for its direct base class using **getSuperclass( )**. This, of course, returns a **Class** handle which you can further query. This means that, at run time, you can discover an object's entire class hierarchy.

The **newInstance( )** method of **Class** can at first seem like just another way to **clone( )** an object. However, you can create a new object with **newInstance( )** *without* an existing object, as seen here, because there is no **Toy** object, only **cy** which is a handle to **y**'s **Class** object. This is a way to implement a "virtual constructor," which allows you to say: "I don't know exactly what type you are, but create yourself properly anyway." In the above example, **cy** is just a **Class** handle with no further type information known at compile time. And when you create a new instance, you get back an **Object** handle. But that handle is actually pointing to a **Toy** object. Of course, before you can send any messages other than those accepted by **Object**, you have to investigate it a bit and do some casting. In addition, the class that's being created with **newInstance( )** must have a default constructor. There's

no way to use **newInstance( )** to create objects that have non-default constructors, so this can be a bit limiting in Java 1. However, the *reflection* API in Java 1.1 (discussed in the next section) allows you to dynamically use any constructor in a class.

The final method in the listing is **printInfo( )** which takes a **Class** handle and gets its name with **getName( )** and finds out whether it's an interface with **isInterface( )**.

The output from this program is:

```
Class name: FancyToy is interface? [false]
Class name: HasBatteries is interface? [true]
Class name: Waterproof is interface? [true]
Class name: ShootsThings is interface? [true]
Class name: Toy is interface? [false]
```

Thus, with the **Class** object you can find out just about everything you want to know about an object.

reflection: run-time class information

If you don't know the precise type of an object, RTTI will tell you that type. However, there's a limitation: that type must be in your local system and loaded (or loadable) into your program space in order for you to be able to detect it and do something useful with the object.

This doesn't seem like that much of a limitation at first, but suppose you're given a handle to an object that's not in your program space. In fact, the class of the object isn't even available to your program. For example, suppose you get a bunch of bytes from a disk file or from a network connection, and you're told those bytes represent a class. In this discussion, an object of such a class will be referred to as a *non-local object*. How can you even attempt to cast that handle to the appropriate class, much less use it?

In a traditional programming environment this seems like a far-fetched scenario. But as we move into a larger programming world there are important cases where this happens. The first is component-based programming where you build projects using *rapid-application-development* (RAD) in an application builder tool. This is a visual approach to creating a program (represented as a *form*) by moving icons that represent components onto the form. These components are then configured by setting some of their values at program time. This program-time configuration requires that the component be instantiable and that it expose some part of itself and allow its values to be read and set. In addition, components that handle GUI events must expose information about appropriate methods so the RAD environment may assist the programmer in overriding these event-handling methods. Reflection provides the mechanism to detect the available methods and produce the method names. Java 1.1 provides a structure for component-based programming through JavaBeans (described in Chapter 13).

Another compelling motivation for discovering class information at run-time is to provide the ability to create and execute objects on remote platforms across a network. This is called *Remote Method Invocation* (RMI) and it allows a Java program (version 1.1 and higher) to have objects distributed across many machines. This distribution may happen for a number of reasons: perhaps you're doing a computation-intensive task and you want to break it up and put pieces on machines that are idle in order to speed things up. In some situations you may want to place code that handles particular types of tasks (e.g. "Business Rules" in a multi-tier client/server architecture) on a particular machine so that machine becomes a common repository describing those actions and it can be easily changed to affect everyone in the system (This is an interesting development since the machine exists solely to make software changes easy!) Along these lines, distributed computing also supports specialized hardware that may be good at a particular task – matrix inversions, for example – but inappropriate or too expensive for general purpose programming.

In Java 1.1, the **Class** (described previously in this chapter) is extended to support the concept of *reflection*, and there's an additional library **java.lang.reflect** with classes **Field**, **Method** and **Constructor** (each of which implement the **Member interface**). Objects of these types are created by the Java Virtual Machine, at run-time, to represent the corresponding member in the non-local class. You may then use the **Constructor**s to create new objects, **get( )** and **set( )** methods to read and modify the fields associated with **Field** objects, and the **invoke( )** method to call a method associated with a **Method** object. In addition, you can call the convenience methods **getFields( )**, **getMethods( )**, **getConstructors( )**, etc., to return arrays of the objects representing the fields, methods and constructors (you can find out more by looking up the class **Class** in your online documentation). Thus, the class information for non-local objects can be completely determined at run time, and nothing need be known at compile time.

a class method extractor

You'll rarely need to use the reflection tools directly – they're in the language to support the other Java features such as object serialization (described in Chapter 10), JavaBeans and RMI (described later in the book). However, there are times when it's quite useful to be able to dynamically extract information about a class. One extremely useful tool is a class method extractor. As mentioned before, looking at a class definition source code or online documentation only shows the methods that are defined or overridden *within that class definition*. But there may be dozens more available to you that have come from base classes. To locate these is tedious and time-consuming. Fortunately, reflection provides a way to write a very simple tool that will automatically show you the entire interface. Here's the way it works:

```
//: ShowMethods.java
// Using Java 1.1 reflection to show all the
// methods of a class, even if the methods are
// defined in the base class.
import java.lang.reflect.*;

public class ShowMethods {
  static final String usage =
    "usage: \n" +
    "ShowMethods qualified.class.name\n" +
    "To show all methods in class or: \n" +
    "ShowMethods qualified.class.name word\n" +
    "To search for methods involving 'word'";
  public static void main(String args[]) {
    if(args.length < 1) {
      System.out.println(usage);
      System.exit(0);
    }
    try {
      Class c = Class.forName(args[0]);
      Method[] m = c.getMethods();
      Constructor[] ctor = c.getConstructors();
      if(args.length == 1) {
        for (int i = 0; i < m.length; i++)
          System.out.println(m[i].toString());
        for (int i = 0; i < ctor.length; i++)
          System.out.println(ctor[i].toString());
      }
      else {
        for (int i = 0; i < m.length; i++)
          if(m[i].toString()
              .indexOf(args[1])!= -1)
            System.out.println(m[i].toString());
        for (int i = 0; i < ctor.length; i++)
          if(ctor[i].toString()
              .indexOf(args[1])!= -1)
```

```
            System.out.println(ctor[i].toString());
        }
      } catch (ClassNotFoundException e) {
        System.out.println("No such class: " + e);
      }
    }
} ///:~
```

The **Class** methods **getMethods( )** and **getConstructors( )** return an array of **Method** and **Constructor**, respectively. Each of these classes has further methods to dissect the names, arguments and return values of the methods they represent. But you can also just use **toString( )**, as is done here, to produce a **String** with the entire method signature. The rest of the code is just for extracting command-line information, determining if a particular signature matches with your target string (using **indexOf( )**) and printing the results.

This shows reflection in action, since the result produced by **Class.forName( )** cannot be known at compile-time, and therefore all the method signature information is being extracted at run-time. The class could also appear through a network using object serialization, so you may not even have the **.class** file anywhere on your machine. If you investigate your online documentation on reflection, you'll see there is enough support to actually set up and make a method call on an object that's totally unknown at compile-time. Again, this is something you'll probably never need to do yourself – the support is there for Java itself and for visual programming libraries (in particular, for JavaBeans) – but it's interesting.

An interesting experiment is to run **java ShowMethods ShowMethods**. This produces a listing that includes a **public** default constructor, even though you can see from the code that no constructor was defined. The constructor you see is the one that's automatically synthesized by the compiler. If you then make **ShowMethods** a non-**public** class (that is, friendly), the synthesized default constructor no longer shows up in the output. The synthesized default constructor is automatically given the same access as the class itself.

The output for **ShowMethods** is still a little tedious. For example, here's a portion of the output produced by invoking **java ShowMethods java.lang.String**:

```
public boolean
  java.lang.String.startsWith(java.lang.String,int)
public boolean
  java.lang.String.startsWith(java.lang.String)
public boolean
  java.lang.String.endsWith(java.lang.String)
```

It would be even nicer if the qualifiers like **java.lang** could be stripped off. The **StreamTokenizer** class introduced in the previous chapter can help solve this problem:

```
//: ShowMethodsClean.java
// ShowMethods with the qualifiers stripped
// to make the results easier to read
import java.lang.reflect.*;
import java.io.*;

public class ShowMethodsClean {
  static final String usage =
    "usage: \n" +
    "ShowMethodsClean qualified.class.name\n" +
    "To show all methods in class or: \n" +
    "ShowMethodsClean qualified.class.name word\n" +
    "To search for methods involving 'word'";
  public static void main(String args[]) {
    if(args.length < 1) {
      System.out.println(usage);
      System.exit(0);
```

```
    }
    try {
      Class c = Class.forName(args[0]);
      Method[] m = c.getMethods();
      Constructor[] ctor = c.getConstructors();
      // Convert to an array of cleaned Strings:
      String n[] =
        new String[m.length + ctor.length];
      for(int i = 0; i < m.length; i++) {
        String s = m[i].toString();
        n[i] = StripQualifiers.strip(s);
      }
      for(int i = 0; i < ctor.length; i++) {
        String s = ctor[i].toString();
        n[i + m.length] =
          StripQualifiers.strip(s);
      }
      if(args.length == 1)
        for (int i = 0; i < n.length; i++)
          System.out.println(n[i]);
      else
        for (int i = 0; i < n.length; i++)
          if(n[i].indexOf(args[1])!= -1)
            System.out.println(n[i]);
    } catch (ClassNotFoundException e) {
      System.out.println("No such class: " + e);
    }
  }
}

class StripQualifiers {
  private StreamTokenizer st;
  public StripQualifiers(String qualified) {
      st = new StreamTokenizer(
        new StringReader(qualified));
      st.ordinaryChar(' '); // Keep the spaces
  }
  public String getNext() {
    String s = null;
    try {
      if(st.nextToken() !=
            StreamTokenizer.TT_EOF) {
        switch(st.ttype) {
          case StreamTokenizer.TT_EOL:
            s = null;
            break;
          case StreamTokenizer.TT_NUMBER:
            s = Double.toString(st.nval);
            break;
          case StreamTokenizer.TT_WORD:
            s = new String(st.sval);
            break;
          default: // single character in ttype
            s = String.valueOf((char)st.ttype);
        }
      }
    } catch(IOException e) {
      System.out.println(e);
    }
    return s;
```

```
  }
  public static String strip(String qualified) {
    StripQualifiers sq =
      new StripQualifiers(qualified);
    String s = "", si;
    while((si = sq.getNext()) != null) {
      int lastDot = si.lastIndexOf('.');
      if(lastDot != -1)
        si = si.substring(lastDot + 1);
      s += si;
    }
    return s;
  }
} ///:~
```

The class **ShowMethodsClean** is quite similar to the previous **ShowMethods**, except that it takes the arrays of **Method** and **Constructor** and converts them into a single array of **String**. Each of these **String** objects is then passed through **StripQualifiers.Strip( )** to remove all the method qualification. As you can see, this uses the **StreamTokenizer** and **String** manipulation to do its work.

This tool can be a real timesaver while you're programming, when you can't remember if a class has a particular method and you don't want to go walking through the class hierarchy in the online documentation, or if you don't know whether that class can do anything with (for example) **Color** objects.

Chapter 17 contains a GUI version of this program that you can leave running while you're programming, to allow quick lookups.

summary

RTTI allows you to discover type information from an anonymous base-class handle. Thus, it's ripe for misuse by the novice since it may make sense before polymorphic method calls do. For many people coming from a procedural background, it's very difficult not to organize their programs into sets of **switch** statements. They could accomplish this with RTTI and thus lose the very important value of polymorphism in code development and maintenance. The intent of Java is that you use polymorphic method calls throughout your code, and you only use RTTI when you must.

However, using polymorphic method calls as they are intended requires that you have control of the base-class definition because at some point in the extension of your program you may discover the base class doesn't include the method you need. If the base class comes from a library or is otherwise controlled by someone else, a solution to the problem is RTTI: You can inherit a new type and add your extra method. Elsewhere in the code you can detect your particular type and call that special method. This doesn't destroy the polymorphism and extensibility of the program, because adding a new type will not require you to hunt for switch statements in your program. However, when you add new code in your main body that requires your new feature, you'll have to use RTTI to detect your particular type.

Putting a feature in a base class might mean that, for the benefit of one particular class, all the other classes derived from that base require some meaningless stub of a method. This makes the interface less clear and annoys those who must redefine abstract methods when they derive from that base class. For example, consider a class hierarchy representing musical instruments. Suppose you wanted to clear the spit valves of all the appropriate instruments in your orchestra. One option is to put a **ClearSpitValve( )** method in the base class **Instrument**, but this is confusing because it implies that **Percussion** and **Electronic** instruments also have spit valves. RTTI provides a much more reasonable solution in this case because you can place the method in the specific class (**Wind** in this case) where it's appropriate. However, a more appropriate solution is to put a **prepareInstrument( )** method in the base class, but you might not see this when you're first solving the problem and may mistakenly assume that you must use RTTI.

Finally, RTTI will sometimes solve efficiency problems. If your code nicely uses polymorphism, but it turns out that one of your objects reacts to this general-purpose code in a horribly inefficient way, you can pick out that type using RTTI and write case-specific code to improve the efficiency.

exercises

1. Write a method that takes an object and recursively prints all the classes in that object's hierarchy.

2. In **ToyTest.java**, comment out **Toy**'s default constructor and explain what happens.

3. Inherit **Vector** to create a new type of **Vector** container that captures the type of the first object you put in it, and then will only allow you to insert objects of that type from then on.

4. Write a program to determine whether an array of **char** is a primitive type or a true object.

12

12: passing and returning objects

[[[Chapter 12 directory:c12]]]

By this time you should be reasonably comfortable with the idea that when you're "passing" an object, you're actually passing a handle (a reference).

In many, if not all programming languages, most of the time you can use the "regular" way to pass objects around and everything works fine. But it always seems that there comes a point where you must do something irregular and suddenly things get a bit more complicated (or in the case of C++, very complicated). Java is no exception here, and it's important that you understand exactly what's happening with your object handles as you pass them around and assign to them. This chapter will provide that insight.

Another way to pose the question of this chapter, if you're coming from a programming language so equipped, is "does Java have pointers?" Some have claimed no, pointers are hard and dangerous and therefore bad, and since Java is all goodness and light and will lift your earthly programming burdens it cannot possibly contain such things. However, it's more accurate to say that Java has pointers; indeed, every object that has a name in Java is one of these pointers, but their use is very restricted and guarded not only by the compiler but by the run-time system. Or to put in another way, Java has pointers, but no pointer arithmetic. These are what I've been calling "handles," and you can think of

them as “safety pointers,” not unlike the safety scissors of early elementary school: they aren’t sharp so you cannot hurt yourself without a very great effort, but they can sometimes be slow and tedious.

passing handles around

It’s worth performing an experiment to show that when you pass a handle into a method, you’re still pointing to the same object. This is quite simple:

```
//: PassHandles.java
// Passing handles around

public class PassHandles {
  static void f(PassHandles h) {
    System.out.println("h inside f(): " + h);
  }
  public static void main(String args[]) {
    PassHandles p = new PassHandles();
    System.out.println("p inside main(): " + p);
    f(p);
  }
} ///:~
```

The method **toString( )** is automatically invoked in the print statements, and **PassHandles** inherits directly from **Object** with no redefinition of **toString( )**. Thus, **Object**’s version of **toString( )** is used, which prints out the class of the object followed by the address where that object is located (not the handle, but the actual object storage). The output looks like this:

```
p inside main(): PassHandles@1653748
h inside f(): PassHandles@1653748
```

You can see that both **p** and **h** refer to the same object. This is far more efficient than creating a new **PassHandles** object just so you can send an argument to a method. But it brings up an important issue: aliasing.

aliasing

Aliasing means that more than one handle is tied to the same object, as in the above example. The problem with aliasing occurs when someone *writes* to that object. If the owners of the other handles aren’t expecting that object to change, they’ll be surprised. This can be demonstrated with a simple example:

```
//: Alias1.java
// Aliasing two handles to one object

public class Alias1 {
  int i;
  Alias1(int ii) { i = ii; }
  public static void main(String args[]) {
    Alias1 x = new Alias1(7);
    Alias1 y = x; // Assign the handle
    System.out.println("x: " + x.i);
    System.out.println("y: " + y.i);
    System.out.println("Incrementing x");
    x.i++;
    System.out.println("x: " + x.i);
    System.out.println("y: " + y.i);
  }
} ///:~
```

In the line:

```
Alias1 y = x; // Assign the handle
```

A new **Alias1** handle is created, but instead of being assigned to a fresh object created with **new**, it's assigned to an existing handle. This means that the contents of handle **x**, which is the address of the object **x** is pointing to, is assigned to **y**, and thus both **x** and **y** are attached to the same object. So when **x**'s **i** is incremented in the statement:

```
x.i++;
```

y's **i** will be affected as well. This can be seen in the output:

```
x: 7
y: 7
Incrementing x
x: 8
y: 8
```

One very good solution in this case is simply not to do it: don't consciously alias more than one handle to an object at the same scope. Your code will be much easier to understand and debug. However, when you're passing a handle in as an argument – which is the way Java is supposed to work – you automatically alias because the local handle that's created can modify the "outside object" (the object that was created outside the scope of the method). Here's an example:

```
//: Alias2.java
// Function calls implicitly alias their
// arguments.

public class Alias2 {
  int i;
  Alias2(int ii) { i = ii; }
  static void f(Alias2 handle) {
    handle.i++;
  }
  public static void main(String args[]) {
    Alias2 x = new Alias2(7);
    System.out.println("x: " + x.i);
    System.out.println("Calling f(x)");
    f(x);
    System.out.println("x: " + x.i);
  }
} ///:~
```

The output is:

```
x: 7
Calling f(x)
x: 8
```

The method is changing its argument, the outside object. When this kind of situation arises, you must decide whether it makes sense, whether the user expects it, and whether it's going to cause problems.

Generally, you call a method in order to produce a return value and/or a change of state in the object. It's much less common to call a method in order to manipulate its arguments. Thus, when you create a method that does modify the arguments the user must be clearly instructed and warned as to the use of that method and its potential surprises. Because of the confusion and pitfalls, it's much better to avoid changing the argument.

If you need to modify an argument during a method call and you don't intend to modify the outside argument, then you should protect that argument by making a copy inside your method. That's the subject of much of this chapter.

making local copies

To review: all argument passing in Java is performed by passing handles. That is, when you pass "an object," you're really only passing a handle to an object outside the method, so if you perform any modifications with that handle, you modify the outside object. In addition:

- ***Aliasing happens automatically during argument passing***
- ***There are no local objects, only local handles***
- ***Handles have scopes, objects do not***
- ***Object lifetime is never an issue in Java***
- ***There is no language support* (e.g. const) *to prevent objects from being modified (to prevent negative effects of aliasing)***

If you're only reading information from an object and not modifying it, passing a handle is the most efficient form of argument passing. This is nice: the default way of doing things is also the most efficient. However, sometimes it's necessary to be able to treat the object as if it were "local" so that changes you make only affect a local copy and do not modify the outside object. Many programming languages support the ability to automatically make a local copy of the outside object, inside the method[1]. Java does not, but it allows you to produce this effect.

pass by value

This brings up the terminology issue, which always seems good for an argument. The term is "pass by value," and the meaning really depends on the way you perceive the operation of the program. The general meaning is that you get a local copy of whatever you're passing, but the real question is "how do you think about what you're passing?" There are two fairly distinct camps:

1. Java passes everything by value. When you're passing primitives into a method, you get a distinct copy of the primitive. When you're passing a handle into a method, you get a copy of the handle. Ergo, everything is pass by value. Of course, the assumption is that you're always thinking (and caring) that handles are being passed, but it seems like the Java design has gone a long way towards allowing you to ignore (most of the time) that you're working with a handle. That is, it seems to allow you to think of the handle as "the object," since it implicitly dereferences it whenever you make a method call.

2. Java passes primitives by value (no argument there) but objects are passed by reference. This is the world view that the handle is an alias for the object itself, so you *don't* think about passing handles, but instead say "I'm passing the object." Since you don't get a local copy of the object when you pass it into a method, objects are clearly not passed by value. There appears to be some support for this view within Javasoft itself, since one of the "reserved but not implemented" keywords is **byvalue** (there's no knowing, however, whether that keyword will ever see the light of day).

Having given both camps a good airing and after saying "it depends on how you think of a handle," I will attempt to sidestep the issue for the rest of the book. In the end, it isn't *that* important – what is

---

[1] In C, which generally handles small bits of data, the default is pass-by-value. C++ had to follow this form, but with objects pass-by-value isn't usually the most efficient way. In addition, coding classes to support pass-by-value in C++ is a big headache.

important is that you understand that passing a handle allows the caller's object to be changed unexpectedly.

cloning objects

The most likely reason for making a local copy of an object is if you're going to modify that object and you don't want to modify the caller's object. If you decide that you want to make a local copy, you simply perform the operation that the language doesn't do for you with the **clone( )** method. This is a method that's defined as **protected** in the base class **Object** and which you must redefine as **public** in any derived classes that you want to clone. For example, the standard library class **Vector** redefines **clone( )**, so we can call **clone( )** for **Vector**:

```
//: Cloning.java
// The clone() operation only works for a few
// items in the standard Java library.
import java.util.*;

class Int {
  private int i;
  public Int(int ii) { i = ii; }
  public void increment() { i++; }
  public String toString() {
    return Integer.toString(i);
  }
}

public class Cloning {
  public static void main(String args[]) {
    Vector v = new Vector();
    for(int i = 0; i < 10; i++ )
      v.addElement(new Int(i));
    System.out.println("v: " + v);
    Vector v2 = (Vector)v.clone();
    // Increment all v2's elements:
    for(Enumeration e = v2.elements();
        e.hasMoreElements(); )
      ((Int)e.nextElement()).increment();
    // See if it's changed v's elements:
    System.out.println("v: " + v);
  }
} ///:~
```

The **clone( )** method produces an **Object**, which must then be recast to the proper type. This example shows how **Vector**'s **clone( )** method *does not* automatically try to clone each of the objects that the **Vector** contains – the old **Vector** and the cloned **Vector** are aliased to the same objects. This is often called a *shallow copy*. You can see it in the output, where the actions performed on **v2** affect **v**:

```
v: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
v: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Not trying to **clone( )** the objects contained in the **Vector** is probably a fair assumption because there's no guarantee that those objects *are* cloneable.

adding cloneability to a class

Even though the clone method is defined in the base-of-all-classes **Object**, cloning is *not* automatically available in every class. This would seem to be counterintuitive to the idea that base-class methods are always available in derived classes. The design of cloning in Java is such that it isn't automatically available. If you want it to exist for a class, you must specifically add code to make cloning work.

using a trick with protected

To prevent default clonability in every class you create, the **clone( )** method is **protected** in the base class **Object**. Not only does this mean that it's not available to the typical client programmer by default, but it also means that you cannot call **clone( )** via a handle to the base class (although that might seem to be useful in some situations: polymorphically clone a bunch of **Object**s). It is, in effect, a way to give you at compile time the information that your object is not cloneable – and oddly enough most classes in the standard Java library are not cloneable. Thus, if you say:

```
Integer x = new Integer(1);
x = x.clone();
```

You will get, at compile time, an error message that says **clone( )** is not accessible (since **Integer** doesn't override it and it defaults to the **protected** version).

If, however, you're in a class derived from **Object** (as all classes are) then you have permission to call **Object.clone( )** because it's **protected** and you're an inheritor. The base class **clone( )** has very useful functionality –it performs the actual bitwise duplication *of the derived-class object*, thus acting as the common cloning operation. However, you then need to make *your* clone operation **public** for it to be accessible. Thus two key issues when you clone are: virtually always call **super.clone( )** and make your clone **public**.

You'll probably want to override **clone( )** in any further derived classes, otherwise your (now **public**) **clone( )** will be used, and that may not do the right thing (although, since **Object.clone( )** makes a copy of the actual object, it may). Thus the **protected** trick only works once, the first time you inherit from a class that has no clonability and you want to make a class that's cloneable. In any classes inherited from your class the **clone( )** method is available since it's not possible in Java to reduce the access of a method during derivation. That is, once a class is cloneable, everything derived from it is cloneable unless you use provided mechanisms (described later) to "turn off" cloning.

implementing the Cloneable interface

The second thing you need to do to complete the clonability of an object is to implement the **Cloneable interface**. This **interface** is a bit strange because it's empty!

```
interface Cloneable {}
```

The reason for implementing this empty **interface** is obviously not because you are going to upcast to **Cloneable** and call one of its methods. The use of **interface** here is considered by some to be a "hack" because it's using a feature for something other than its original intent. Implementing the **Cloneable interface** acts as a kind of a flag, wired into the type of the class.

There are two reasons for the existence of the **Cloneable interface**. The first is that you may have an upcast handle to a base type and not know whether it's possible to clone that object. In this case, you can use the **instanceof** keyword (described in Chapter 11) to find out whether the handle is connected to an object that can be cloned:

```
if(myHandle instanceof Cloneable) // ...
```

The second reason is that **Object.clone( )** verifies that a class implements the **Cloneable** interface. If not, it throws an exception. So in general you're forced to **implement Cloneable** as part of support for cloning.

successful cloning

Once you understand the details of implementing the **clone( )** method, you're able to create classes that can be easily duplicated to provide a local copy:

```
//: LocalCopy.java
// Creating local copies with clone()
import java.util.*;

class MyObject implements Cloneable {
```

```
  int i;
  MyObject(int ii) { i = ii; }
  public Object clone() {
    Object o = null;
    try {
      o = super.clone();
    } catch (CloneNotSupportedException e) {
      System.out.println("MyObject can't clone");
    }
    return o;
  }
  public String toString() {
    return Integer.toString(i);
  }
}

public class LocalCopy {
  static MyObject g(MyObject v) {
    // Passing a handle, modifies outside object:
    v.i++;
    return v;
  }
  static MyObject f(MyObject v) {
    v = (MyObject)v.clone(); // Local copy
    v.i++;
    return v;
  }
  public static void main(String args[]) {
    MyObject a = new MyObject(11);
    MyObject b = g(a);
    // Testing object equivalence,
    // not value equivalence:
    if(a == b)
      System.out.println("a == b");
    else
      System.out.println("a != b");
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    MyObject c = new MyObject(47);
    MyObject d = f(c);
    if(c == d)
      System.out.println("c == d");
    else
      System.out.println("c != d");
    System.out.println("c = " + c);
    System.out.println("d = " + d);
  }
} ///:~
```

The necessary structure of **clone( )** and the **toString( )** method for easy printing should now be familiar. The call to **super.clone( )** duplicates the core of the object, including **i**. For simplicity, **int i** is "friendly" so it can be accessed directly.

In **LocalCopy**, the two methods **g( )** and **f( )** demonstrate the difference between the two approaches for argument passing. **g( )** shows passing by reference where it modifies the outside object and returns a reference to that same outside object, while **f( )** clones the argument, thereby decoupling it and leaving the original object alone. It can then proceed to do whatever it wants, and even to return a handle to this new object without any ill effects to the original. Notice the somewhat curious-looking statement:

```
v = (MyObject)v.clone();
```

This is where the local copy is created. To keep from being confused by such a statement, you must remember that this rather strange coding idiom is perfectly feasible in Java because everything that has a name is actually a handle. So the handle **v** is used to **clone( )** a copy of what it refers to, and this returns a handle to the base type **Object** (because it's defined that way in **Object.clone( )**) which must then be cast to the proper type.

In **main( )**, the difference between the effects of the two different argument-passing approaches in the two different methods is tested. The output is:

```
a == b
a = 12
b = 12
c != d
c = 47
d = 48
```

It's important to note that the equivalence tests in Java do not look inside the objects being compared to see if their values are the same. The **==** and **!=** operators are simply comparing the contents of the *handles*: if the addresses inside the handles are the same, that means that the handles are pointing to the same object and are therefore "equal." So what the operators are really testing is whether the handles are aliased to the same object!

the effect of Object.clone( )

What actually happens when **Object.clone( )** is called that makes it so essential to call **super.clone( )** when you override **clone( )** in your class? The **clone( )** method in the root class is responsible for creating the right amount of storage and making the actual bitwise copy of the bits in your object to the new storage. That is, it doesn't just make storage and copy an **Object** – it actually figures out the size of the precise object that's being copied, and duplicates that. Since all this is happening from the code in the **clone( )** method defined in the root class (that has no idea what's being inherited from it), you can guess that the process involves RTTI to determine the actual object that's being cloned. This way, the **clone( )** method can create the correct amount of storage and do the correct bitcopy for that type.

This means that whatever you do, the first part of the cloning process should normally be a call to **super.clone( )**. This establishes the groundwork for the cloning operation by making an exact duplicate. At this point you can perform other operations necessary to complete the cloning.

To know for sure what those other operations are, you need to understand exactly what **Object.clone( )** buys you. In particular, does it automatically clone the destination of all the handles? The following example tests this:

```
//: Snake.java
// Tests cloning to see if destination of
// handles are also cloned.

public class Snake implements Cloneable {
  private Snake next;
  private char c;
  // Value of i == number of segments
  Snake(int i, char x) {
    c = x;
    if(--i > 0)
      next = new Snake(i, (char)(x + 1));
  }
  void increment() {
    c++;
    if(next != null)
      next.increment();
  }
  public String toString() {
```

```
    String s = ":" + c;
    if(next != null)
      s += next.toString();
    return s;
  }
  public Object clone() {
    Object o = null;
    try {
      o = super.clone();
    } catch (CloneNotSupportedException e) {}
    return o;
  }
  public static void main(String args[]) {
    Snake s = new Snake(5, 'a');
    System.out.println("s = " + s);
    Snake s2 = (Snake)s.clone();
    System.out.println("s2 = " + s2);
    s.increment();
    System.out.println(
      "after s.increment, s2 = " + s2);
  }
} ///:~
```

A **Snake** is made up of a bunch of segments, each of type **Snake**. Thus it's a singly-linked list. The segments are created recursively, decrementing the first constructor argument for each segment until zero is reached. To give each segment a unique tag, the second argument, a **char**, is incremented for each recursive constructor call.

The **increment( )** method recursively increments each tag so you can see the change, and the **toString( )** recursively prints each tag. The output is:

```
s = :a:b:c:d:e
s2 = :a:b:c:d:e
after s.increment, s2 = :a:c:d:e:f
```

This means that only the first segment is duplicated by **Object.clone( )**. Thus, if you want the whole snake to be duplicated – a deep copy – you must perform the additional operations inside your overridden **clone( )**.

So you'll typically call **super.clone( )** in any class derived from a cloneable class, to make sure that all the base-class operations (including **Object.clone( )**) take place. This is followed by an explicit call to **clone( )** for every handle in your object – otherwise those handles will be aliased to those of the original object. It's very analogous to the way constructors are called – base-class constructor first, then the next-derived constructor and so on to the most-derived constructor. The difference is that **clone( )** is not a constructor so there's nothing to make it happen automatically. You must make sure to do it yourself.

completing the cloning

What other operations should you perform after **super.clone( )** is called? This depends on your class, and whether you want to perform a shallow copy or a deep copy. If you want to perform a shallow copy, then the operations performed by **Object.clone( )** (bitwise duplication of the "immediate" object) are adequate. However, if you want to do a deep copy, you must also perform cloning on every object handle in your class. In addition, you must be able to assume that the **clone( )** method in those objects will in turn perform a deep copy on *their* handles, and so on. This is quite a commitment. It effectively means that for a deep copy to work you must either control all the code in all the classes, or at least have enough knowledge about all the classes involved in the deep copy to know that they are performing their own deep copy correctly.

Let's look at an example of a deep copy. Here is the **Vector** example from earlier in this chapter, but this time the **Int2** class is cloneable so the **Vector** can be deep copied:

```
//: AddingClone.java
// You must go through a few gyrations to
// add cloning to your own class.
import java.util.*;

class Int2 implements Cloneable {
  private int i;
  public Int2(int ii) { i = ii; }
  public void increment() { i++; }
  public String toString() {
    return Integer.toString(i);
  }
  public Object clone() {
    Object o = null;
    try {
      o = super.clone();
    } catch (CloneNotSupportedException e) {
      System.out.println("Int2 can't clone");
    }
    return o;
  }
}

// Once it's cloneable, inheritance
// doesn't remove cloneability:
class Int3 extends Int2 {
  public Int3(int i) { super(i); }
}

public class AddingClone {
  public static void main(String args[]) {
    Int2 x = new Int2(10);
    Int2 x2 = (Int2)x.clone();
    x2.increment();
    System.out.println(
      "x = " + x + ", x2 = " + x2);
    // Anything inherited is also cloneable:
    Int3 x3 = new Int3(7);
    x3 = (Int3)x3.clone();

    Vector v = new Vector();
    for(int i = 0; i < 10; i++ )
      v.addElement(new Int2(i));
    System.out.println("v: " + v);
    Vector v2 = (Vector)v.clone();
    // Now clone each element:
    for(int i = 0; i < v.size(); i++)
      v2.setElementAt(
        ((Int2)v2.elementAt(i)).clone(), i);
    // Increment all v2's elements:
    for(Enumeration e = v2.elements();
        e.hasMoreElements(); )
      ((Int2)e.nextElement()).increment();
    // See if it's changed v's elements:
    System.out.println("v: " + v);
    System.out.println("v2: " + v2);
  }
} ///:~
```

First of all, **clone( )** must be accessible so you'll need to make it **public**. Second, for the initial part of your **clone( )** operation you should call the base-class version of **clone( )**. The **clone( )** that's being called here is the one that's pre-defined inside **Object**, and you can call it because it's **protected** and thereby accessible in derived classes.

Object.clone( ) figures out how big the actual object is, creates enough memory for a new one, and copies all the bits from the old to the new. This is called a *bitwise copy*, and is typically what you'd expect a **clone( )** method to do. However, mixed into this design for clonability was the thought that maybe you didn't want all types of objects to be cloneable. So before **Object.clone( )** performs its operations, it first checks to see if a class is **Cloneable**, that is, whether it implements the **Cloneable** interface. If it doesn't, **Object.clone( )** throws a **CloneNotSupportedException** to indicate that you can't clone it. Thus, you've got to surround your call to **Object.clone( )** with a try-catch block, to catch an exception that should never happen because you've implemented the **Cloneable** interface.

The remainder of the example demonstrates the cloning by showing that, once an object is cloned, you can change it and the original object is truly left untouched.

You can also see what's necessary in order to do a complete clone of a **Vector**: after the **Vector** itself is cloned, you have to step through and clone each one of the objects pointed to by the **Vector**. You'd have to do something similar to this to do a complete clone (sometimes called a *deep copy*) of a **Hashtable**.

adding cloneability further down a hierarchy

If you create a new class, its base class defaults to **Object** which defaults to non-clonability (as you'll see in the next section). As long as you don't explicitly add clonability, you won't get it. But you can add it in at any layer, like this:

```
//: HorrorFlick.java
// You can insert Cloneability at any
// level of inheritance.
import java.util.*;

class Person {}
class Hero extends Person {}
class Scientist extends Person
    implements Cloneable {
  public Object clone() {
    try {
      return super.clone();
    } catch (CloneNotSupportedException e) {
      // this should never happen:
      // It's Cloneable already!
      throw new InternalError();
    }
  }
}
class MadScientist extends Scientist {}

public class HorrorFlick {
  public static void main(String args[]) {
    Person p = new Person();
    Hero h = new Hero();
    Scientist s = new Scientist();
    MadScientist m = new MadScientist();

    // p = (Person)p.clone(); // Compile error
    // h = (Hero)h.clone(); // Compile error
    s = (Scientist)s.clone();
    m = (MadScientist)m.clone();
  }
```

```
} ///:~
```

Before clonability was added, the compiler stopped you from trying to clone things. When clonability is added in **Scientist**, then **Scientist** and all its descendants are cloneable.

why this strange design?

If all this seems to be a strange scheme, that's because it is. You may wonder why it worked out this way. What is the meaning behind this design? What follows is not a substantiated story – probably because much of the marketing around Java makes it out to be a perfectly-designed language – but it does go a long way towards explaining how things ended up the way they did.

Originally, Java was designed as a language to control hardware boxes, and definitely not with the Internet in mind. In a general-purpose language like this, it makes sense that the programmer be able to clone any object. Thus **clone( )** was placed in the root class **Object**, *but* it was a **public** method so you could always clone any object. This seemed to be the most flexible approach, and after all what could it hurt?

Well, when Java was seen as the ultimate Internet programming language, things changed. Suddenly, there are security issues, and of course these issues are dealt with using objects, and you don't necessarily want anyone to be able to clone your security objects. So what you're seeing is a lot of patches applied on the original simple and straightforward scheme: **clone( )** is now **protected** in **Object.** You must override it *and* **implement Cloneable** *and* deal with the exceptions.

It's worth noting that you must use the **Cloneable** interface *only* if you're going to call **Object**'s **clone( )**, method, since that method checks at run-time to make sure your class implements **Cloneable**. But for consistency (and since **Cloneable** is empty anyway) you should implement it.

controlling cloneability

You might suggest that, to remove clonability, the **clone( )** method simply be made **private**, but this won't work since you cannot take a base-class method and make it more **private** in a derived class. So it's not that simple. And yet, it's necessary to be able to control whether an object can be cloned or not. There's actually a number of attitudes you can take to this in a class that you design:

1. Indifference. You don't do anything about cloning, which means that your class can't be cloned but a class that inherits from you can add cloning if it wants.

2. Support **clone( )**. Follow the standard practice of implementing **Cloneable** and overriding **clone( )**. In the overridden **clone( )**, you call **super.clone( )** and catch all exceptions (so your overridden **clone( )** doesn't throw any exceptions).

3. Conditionally support cloning. If your class holds handles to other objects which may or may not be cloneable (an example of this is a collection class) you may try to clone all the objects that you have handles to as part of your cloning, and if they throw exceptions just pass them through. For example, consider a special sort of **Vector** which tries to clone all the objects it holds. When you write such a **Vector**, you don't know what sort of objects the client programmer may put into your **Vector**, so you don't know whether they can be cloned.

4. Don't implement **Cloneable** but override **clone( )** as **protected**, producing the correct copying behavior for any fields. This way, anyone inheriting from this class can override **clone( )** and call **super.clone( )** to produce the correct copying behavior. Note that your implementation can and should invoke **Object.clone( )** even though that expects a **Cloneable** object (it will throw an exception otherwise), because no one will directly invoke it on an object of your type. It will only get invoked through a derived class which, if it is to work successfully, implements **Cloneable**.

5. Try to prevent cloning by not implementing **Cloneable** and overriding **clone( )** to throw an exception. This is only successful if any class derived from this calls **super.clone( )** in it's redefinition of **clone( )**. Otherwise a programmer can get around it.

6. Prevent cloning by making your class **final**. If **clone( )** has not been overridden then it can't be. If it has, then override it again and throw **CloneNotSupportedException**. Making the class **final** is the only way to guarantee that cloning is prevented. In addition, when dealing with security objects or other situations where you want to control the number of objects created you should make all constructors **private** and provide one or more special methods for creating objects. That way, these methods can restrict the number of objects created and the conditions in which they're created (a particular case of this is the *singleton* pattern shown in Chapter 16).

Here's an example that shows the various ways cloning can be implemented and then, later in the hierarchy, "turned off:"

```
//: CheckCloneable.java
// Checking to see if a handle can be cloned

// Can't clone this because it doesn't
// override clone():
class Ordinary {}

// Overrides clone, but doesn't implement
// Cloneable:
class WrongClone extends Ordinary {
  public Object clone()
      throws CloneNotSupportedException {
    return super.clone(); // Throws exception
  }
}

// Does all the right things for cloning:
class IsCloneable extends Ordinary
    implements Cloneable {
  public Object clone()
      throws CloneNotSupportedException {
    return super.clone();
  }
}

// Turn off cloning by throwing the exception:
class NoMore extends IsCloneable {
  public Object clone()
      throws CloneNotSupportedException {
    throw new CloneNotSupportedException();
  }
}

class TryMore extends NoMore {
  public Object clone()
      throws CloneNotSupportedException {
    // Calls NoMore.clone(), throws exception:
    return super.clone();
  }
}

class BackOn extends NoMore {
  private BackOn duplicate(BackOn b) {
    // Somehow make a copy of b
    // and return that copy. This is a dummy
```

```
    // copy, just to make the point:
    return new BackOn();
  }
  public Object clone() {
    // Doesn't call NoMore.clone():
    return duplicate(this);
  }
}

// Can't inherit from this, so can't override
// the clone method like in BackOn:
final class ReallyNoMore extends NoMore {}

public class CheckCloneable {
  static Ordinary tryToClone(Ordinary ord) {
    String id = ord.getClass().getName();
    Ordinary x = null;
    if(ord instanceof Cloneable) {
      try {
        System.out.println("Attempting " + id);
        x = (Ordinary)((IsCloneable)ord).clone();
        System.out.println("Cloned " + id);
      } catch(CloneNotSupportedException e) {
        System.out.println(
          "Could not clone " + id);
      }
    }
    return x;
  }
  public static void main(String args[]) {
    // Upcasting:
    Ordinary ord[] = {
      new IsCloneable(),
      new WrongClone(),
      new NoMore(),
      new TryMore(),
      new BackOn(),
      new ReallyNoMore(),
    };
    Ordinary x = new Ordinary();
    // This won't compile, since clone() is
    // protected in Object:
    //! x = (Ordinary)x.clone();
    // tryToClone() checks first to see if
    // a class implements Cloneable:
    for(int i = 0; i < ord.length; i++)
      tryToClone(ord[i]);
  }
} ///:~
```

The first class, **Ordinary**, represents the kinds of classes we've been seeing throughout the book: no support for cloning, but as it turns out, no prevention of cloning either. But if you have a handle to an **Ordinary** object that may have been upcast from a more derived class, you can't tell if it can be cloned or not.

The class **WrongClone** shows an incorrect way to implement cloning. It does override **Object.clone( )** and makes that method **public**, but it doesn't implement **Cloneable**, and so when **super.clone( )** is called (which results in a call to **Object.clone( )**), **CloneNotSupportedException** is thrown so the cloning doesn't work.

In **IsCloneable** you can see all the right actions performed for cloning: **clone( )** is overridden and **Cloneable** is implemented. However, this **clone( )** method and several others that follow in this example *do not* catch **CloneNotSupportedException** but instead pass it through to the caller, who must then put a try-catch block around it. In your own **clone( )** methods you will typically catch **CloneNotSupportedException** *inside* **clone( )** rather than passing it through. As you'll see, in this example it's more informative to pass the exceptions through.

Class **NoMore** attempts to "turn off" cloning in the way that the Java designers intended: in the derived class **clone( )**, you throw **CloneNotSupportedException**. The **clone( )** method in class **TryMore** properly calls **super.clone( )**, and this resolves to **NoMore.clone( )** which throws an exception and prevents cloning.

But what if the programmer doesn't follow the "proper" path of calling **super.clone( )** inside the overridden **clone( )** method? In **BackOn**, you can see how this can happen. This class uses a separate method **duplicate( )** to make a copy of the current object, and calls this method inside **clone( )** *instead* of calling **super.clone( )**. The exception is never thrown and the new class is cloneable. This means that you can't rely on throwing an exception to prevent making a cloneable class. The only sure-fire solution is shown in **ReallyNoMore**, which is **final** and thus cannot be inherited. That means if **clone( )** throws an exception in the **final** class, it cannot be modified with inheritance, and the prevention of cloning is assured (you cannot explicitly call **Object.clone( )** from a class that has an arbitrary level of inheritance; you can only call **super.clone( )** which access the direct base class). Thus, if you make any objects that involve security issues, you'll want to make those classes **final**.

The first method you see in class **CheckCloneable** is **tryToClone( )** which takes any **Ordinary** object and checks to see whether it's cloneable with **instanceof**. If so, it casts the object to an **IsCloneable**, calls **clone( )** and casts the result back to **Ordinary**, catching any exceptions that are thrown. Notice the use of run-time type identification (see Chapter 11) to print out the class name so you can see what's happening.

In **main( )**, all different types of **Ordinary** objects are created and upcast to **Ordinary** in the array definition. The first two lines of code after that create a plain **Ordinary** object and try to clone it. However, this code will not compile because **clone( )** is a **protected** method in **Object**. The remainder of the code steps through the array and tries to clone each object, reporting the success or failure of each. The output is:

```
Attempting IsCloneable
Cloned IsCloneable
Attempting NoMore
Could not clone NoMore
Attempting TryMore
Could not clone TryMore
Attempting BackOn
Cloned BackOn
Attempting ReallyNoMore
Could not clone ReallyNoMore
```

So to summarize, if you want a class to be cloneable:

1. Implement the **Cloneable** interface.
2. Override **clone( )**.
3. Call **super.clone( )** inside your **clone( )**.
4. Capture exceptions inside your **clone( )**.

This will produce the most convenient effects.

the copy-constructor

Cloning may seem to be a complicated process to set up. It may seem like there should be an alternative. One approach that might occur to you (especially if you're a C++ programmer) is to make a special constructor whose job it is to duplicate an object. In C++, this is called the *copy constructor*. At first, this seems like the obvious solution. Here's an example:

```
//: CopyConstructor.java
// A constructor for copying an object
// of the same type, as an attempt to create
// a local copy.

class FruitQualities {
  private int weight;
  private int color;
  private int firmness;
  private int ripeness;
  private int smell;
  // etc.
  FruitQualities() { // Default constructor
    // do something meaningful...
  }
  // Other constructors:
  // ...
  // Copy constructor:
  FruitQualities(FruitQualities f) {
    weight = f.weight;
    color = f.color;
    firmness = f.firmness;
    ripeness = f.ripeness;
    smell = f.smell;
    // etc.
  }
}

class Seed {
  // Members...
  Seed() { /* Default constructor */ }
  Seed(Seed s) { /* Copy constructor */ }
}

class Fruit {
  private FruitQualities fq;
  private int seeds;
  private Seed[] s;
  Fruit(FruitQualities q, int seedCount) {
    fq = q;
    seeds = seedCount;
    s = new Seed[seeds];
    for(int i = 0; i < seeds; i++)
      s[i] = new Seed();
  }
  // Other constructors:
  // ...
  // Copy constructor:
  Fruit(Fruit f) {
    fq = new FruitQualities(f.fq);
    // Call all Seed copy-constructors:
    for(int i = 0; i < seeds; i++)
```

```
      s[i] = new Seed(f.s[i]);
    // Other copy-construction activities...
  }
  // To allow derived constructors (or other
  // methods) to put in different qualities:
  protected void addQualities(FruitQualities q) {
    fq = q;
  }
  protected FruitQualities getQualities() {
    return fq;
  }
}

class Tomato extends Fruit {
  Tomato() {
    super(new FruitQualities(), 100);
  }
  Tomato(Tomato t) { // Copy-constructor
    super(t); // Upcast for base copy-constructor
    // Other copy-construction activities...
  }
}

class ZebraQualities extends FruitQualities {
  private int stripedness;
  ZebraQualities() { // Default constructor
    // do something meaningful...
  }
  ZebraQualities(ZebraQualities z) {
    super(z);
    stripedness = z.stripedness;
  }
}

class GreenZebra extends Tomato {
  GreenZebra() {
    addQualities(new ZebraQualities());
  }
  GreenZebra(GreenZebra g) {
    super(g); // Calls Tomato(Tomato)
    // Restore the right qualities:
    addQualities(new ZebraQualities());
  }
  void evaluate() {
    ZebraQualities zq =
      (ZebraQualities)getQualities();
    // Do something with the qualities
    // ...
  }
}

public class CopyConstructor {
  public static void ripen(Tomato t) {
    // Use the "copy constructor":
    t = new Tomato(t);
    System.out.println("In ripen, t is a " +
      t.getClass().getName());
  }
  public static void slice(Fruit f) {
    f = new Fruit(f); // Hmmm... will this work?
```

```
    System.out.println("In slice, f is a " +
      f.getClass().getName());
  }
  public static void main(String args[]) {
    Tomato tomato = new Tomato();
    ripen(tomato); // OK
    slice(tomato); // OOPS!
    GreenZebra g = new GreenZebra();
    ripen(g); // OOPS!
    slice(g); // OOPS!
    g.evaluate();
  }
} ///:~
```

This seems a bit strange at first. Sure, fruit has qualities, but why not just put data members representing those qualities directly into the **Fruit** class? There are two potential reasons. The first is that you may want to easily insert or change the qualities. Notice that **Fruit** has a **protected addQualities( )** method to allow derived classes to do this (you might think the logical thing to do is to have a **protected** constructor in **Fruit** that takes a **FruitQualities** argument, but constructors don't inherit so it wouldn't be available in second or greater level classes). By making the fruit qualities into a separate class, you have greater flexibility, including the ability to change the qualities midway through the lifetime of a particular **Fruit** object.

The second reason for making **FruitQualities** a separate object is in case you want to add new qualities or change the behavior, via inheritance and polymorphism. Notice that for **GreenZebra** (which really is a type of tomato – I've grown them and they're fabulous), the constructor calls **addQualities( )** and passes it a **ZebraQualities** object, which is derived from **FruitQualities** and so can be attached to the **FruitQualities** handle in the base class. Of course, when **GreenZebra** uses the **FruitQualities** it must downcast it to the right type (as seen in **evaluate( )**), but it always knows that type is **ZebraQualities**.

You'll also see that there's a **Seed** class, and that **Fruit** (which by definition carries its own seeds) contains an array of **Seed**s.

Finally, notice that each class has a copy constructor, and that each copy constructor must take care to call the copy constructors for the base class and member objects so as to produce a deep copy. The copy constructor is tested inside the class **CopyConstructor**. The method **ripen( )** takes a **Tomato** argument and performs copy-construction on it in order to duplicate the object:

```
t = new Tomato(t);
```

while **slice( )** takes a more generic **Fruit** object and also duplicates it:

```
f = new Fruit(f);
```

These are tested with different kinds of **Fruit** in **main( )**. Here's the output:

```
In ripen, t is a Tomato
In slice, f is a Fruit
In ripen, t is a Tomato
In slice, f is a Fruit
```

This is where the problem shows up. After the copy-construction that happens to the **Tomato** inside **slice( )**, the result is no longer a **Tomato** object, but just a **Fruit**. It's lost all it's tomato-ness. Further, when you take a **GreenZebra**, both **ripen( )** and **slice( )** turn it into a **Tomato** and a **Fruit**, respectively. Thus, unfortunately, the copy constructor scheme is no good to us in Java when attempting to make a local copy of an object.

why does it work in C++ and not Java?

The copy constructor is a fundamental part of C++, since it automatically makes a local copy of an object. Yet the above example proves it does not work for Java. Why is this? The issue is that in Java everything that we manipulate is a handle, while in C++ you can have handle-like entities but you

can *also* pass around the objects directly. That's what the C++ copy constructor is for: when you want to take an actual object and pass it in by value, thus duplicating the object. So it works fine in C++, but you should keep in mind that this scheme fails in Java, so don't use it.

creating read-only classes

While the local copy produced by **clone( )** gives the desired results in the appropriate cases, it is an example of forcing the programmer (the author of the method) to be responsible for preventing the ill effects of aliasing. What if you're making a library that's so general-purpose and commonly used that you cannot make the assumption that it will always be cloned in the proper places? Or more likely, what if you *want* to allow aliasing for efficiency – to prevent the needless duplication of objects – but you don't want the negative side effects of aliasing?

One solution is to create *immutable objects*. You can define a class such that no methods in the class cause changes to the internal state of the object. In such a class, aliasing has no impact since you can only read the internal state, so if many pieces of code are reading the same object there's no problem.

Here's an example:

```
//: Immutable1.java
// Objects that cannot be modified
// are immune to aliasing.

public class Immutable1 {
  private int data;
  public Immutable1(int initVal) {
    data = initVal;
  }
  public int read() { return data; }
  public boolean nonzero() { return data != 0; }
  public Immutable1 quadruple() {
    return new Immutable1(data * 4);
  }
  static void f(Immutable1 i1) {
    Immutable1 quad = i1.quadruple();
    System.out.println("i1 = " + i1.read());
    System.out.println("quad = " + quad.read());
  }
  public static void main(String args[]) {
    Immutable1 x = new Immutable1(47);
    System.out.println("x = " + x.read());
    f(x);
    System.out.println("x = " + x.read());
  }
} ///:~
```

All data is **private**, and you'll see that none of the **public** methods modify that data. Indeed, the method that does appear to modify an object is **quadruple( )**, but this actually creates a new **Immutable1** object and leaves the original one untouched.

The method **f( )** takes an **Immutable1** object and performs various operations on it, and the output of **main( )** demonstrates that there is no change to **x**. Thus, **x**'s object could be aliased many times without harm, because the **Immutable1** class is designed to guarantee that objects cannot be changed.

the drawback to immutability

Creating an immutable class seems at first to provide an elegant solution. However, whenever you do need a modified object of that new type you must suffer the overhead of a new object creation, as

well as potentially causing more frequent garbage collections. For some classes this is not a problem, but for others (such as the **String** class) it is prohibitive.

The solution is to create a companion class that *can* be modified. Then when you're doing a lot of modifications, you can switch to using the modifiable companion class and then switch back to the immutable class when you're done.

The above example can be modified to show this:

```
//: Immutable2.java
// A companion class for making changes
// to immutable objects.

class Mutable {
  private int data;
  public Mutable(int initVal) {
    data = initVal;
  }
  public Mutable add(int x) {
    data += x;
    return this;
  }
  public Mutable multiply(int x) {
    data *= x;
    return this;
  }
  public Immutable2 makeImmutable2() {
    return new Immutable2(data);
  }
}

public class Immutable2 {
  private int data;
  public Immutable2(int initVal) {
    data = initVal;
  }
  public int read() { return data; }
  public boolean nonzero() { return data != 0; }
  public Immutable2 add(int x) {
    return new Immutable2(data + x);
  }
  public Immutable2 multiply(int x) {
    return new Immutable2(data * x);
  }
  public Mutable makeMutable() {
    return new Mutable(data);
  }
  public static Immutable2 modify1(Immutable2 y){
    Immutable2 val = y.add(12);
    val = val.multiply(3);
    val = val.add(11);
    val = val.multiply(2);
    return val;
  }
  // This produces the same result:
  public static Immutable2 modify2(Immutable2 y){
    Mutable m = y.makeMutable();
    m.add(12).multiply(3).add(11).multiply(2);
    return m.makeImmutable2();
  }
  public static void main(String args[]) {
```

```
    Immutable2 i2 = new Immutable2(47);
    Immutable2 r1 = modify1(i2);
    Immutable2 r2 = modify2(i2);
    System.out.println("i2 = " + i2.read());
    System.out.println("r1 = " + r1.read());
    System.out.println("r2 = " + r2.read());
  }
} ///:~
```

Immutable2 contains methods that, as before, preserve the immutability of the objects by producing new objects whenever a modification is desired. These are the **add( )** and **multiply( )** methods. The companion class is called **Mutable**, and it also has **add( )** and **multiply( )** methods, but these modify the **Mutable** object itself rather than making a new one. In addition, **Mutable** has a method to use its data to produce an **Immutable2** object, and vice versa.

The two static methods **modify1( )** and **modify2( )** show two different approaches to producing the same result. In **modify1( )**, everything is done within the **Immutable2** class and you can see that four new **Immutable2** objects are created in the process (and each time **val** is reassigned, the previous object becomes garbage).

In the method **modify2( )**, you can see that the first action is to take the **Immutable2 y** and produce a **Mutable** from it (this is just like calling **clone( )** as you saw earlier, but this time a different type of object is created). Then the **Mutable** object is used to perform a lot of change operations *without* requiring the creation of many new objects. Finally it's turned back into an **Immutable2**. Here, two new objects are created (the **Mutable** and the result **Immutable2**) instead of four.

This approach makes sense, then, when:

1. You need immutable objects and
2. You often need to make a lot of modifications or
3. It's very expensive to create new immutable objects

immutable Strings

Consider the following code:

```
//: Stringer.java

public class Stringer {
  static String upcase(String s) {
    return s.toUpperCase();
  }
  public static void main(String[] args) {
    String q = new String("howdy");
    System.out.println(q); // howdy
    String qq = upcase(q);
    System.out.println(qq); // HOWDY
    System.out.println(q); // howdy
  }
} ///:~
```

When **q** is passed in to **upcase( )** it's actually a copy of the *handle* to **q**. The object this handle is connected to stays put in a single physical location, while the handles are what are copied and passed around.

Looking at the definition for **upcase( )**, you can see that the handle that's passed in has the name **s**, and it only exists for as long as the body of **upcase( )** is being executed. When **upcase( )** completes, the local handle **s** vanishes. **upcase( )** returns the result which is the original string with all the characters set to uppercase. Of course, it actually returns a handle to the result. But it turns out that the handle that it returns is for a new object, and the original **q** is left alone. How does this happen?

implicit constants

If you say:

```
String s = "asdf";
String x = Stringer.upcase(s);
```

do you really want the **upcase( )** method to *change* the argument? Generally you don't, because an argument usually looks to the reader of the code as a piece of information provided to the method, not something to be modified. This is an important guarantee, since it makes code easier to write and understand.

In C++, the availability of this guarantee was important enough to put in a special keyword, **const**, to allow the programmer to ensure that a handle (pointer or reference in C++) could not be used to modify the original object. But then the C++ programmer was required to be very diligent and remember to use **const** everywhere. It can be confusing and easy to forget.

overloading '+' and the StringBuffer

Objects of the **String** class are designed to be immutable, using the technique shown previously. If you examine the online documentation for the **String** class (which is summarized a little later in this chapter), you'll see that every method in the class that appears to modify a **String** actually creates and returns a brand new **String** object containing the modification. The original **String** is left untouched. Thus, there's no feature in Java like C++'s **const** to make the compiler support the immutability of your objects. If you want it, you have to wire it in yourself, like **String** does.

Since **String** objects are immutable, you can alias to a particular **String** as many times as you want. Because it's read-only there's no possibility that one handle will change something that the other handles will be affected by. So a read-only object solves the aliasing problem nicely.

It also seems possible to handle all the cases where you need a modified object by creating a brand new version of the object with the modifications, as **String** does. However, for some operations this isn't very efficient. A case in point is the operator **'+'** which has been overloaded for **String** objects. Overloading means it has been given an extra meaning when used with a particular class. (The '**+**' and '**+=**' for **String** are the only operators that are overloaded in Java and Java does not allow the programmer to overload any others[2]).

When used with **String** objects, the '**+**' allows you to concatenate **String**s together:

```
String s = "abc" + foo + "def" + Integer.toString(47);
```

You could imagine how this *might* work: the **String** "abc" could have a method **append( )** that creates a new **String** object containing "abc" concatenated with the contents of **foo**. The new **String** object would then create another new **String** that added "def" and so on.

This would certainly work, but it requires the creation of a lot of **String** objects just to put together this new **String**, and then you have a bunch of the intermediate **String** objects that need to be garbage-collected. I suspect that the Java designers actually tried this approach first (which is a lesson in software design – you don't really know anything about a system until you try it out in code and get something working). I also suspect they discovered that it delivered unacceptable performance.

The solution is a mutable companion class similar to the one shown previously. For **String**, this companion class is called **StringBuffer**, and the compiler automatically creates a **StringBuffer** to evaluate certain expressions, in particular when the overloaded operators **+** and **+=** are used with **String** objects. This example shows what happens:

---

[2] C++ allows the programmer to overload operators at will. Because this can often be a complicated process – see Chapter 10 of my book *Thinking in C++* (Prentice-Hall, 1995) – the Java designers deemed it a "bad" feature that shouldn't be included in Java. It wasn't so bad that they didn't end up doing it themselves, and ironically enough, operator overloading would be much easier to use in Java than in C++.

```
//: ImmutableStrings.java
// Demonstrating StringBuffer

public class ImmutableStrings {
  public static void main(String args[]) {
    String foo = "foo";
    String s = "abc" + foo +
      "def" + Integer.toString(47);
    System.out.println(s);
    // The "equivalent" using StringBuffer:
    StringBuffer sb =
      new StringBuffer("abc"); // Creates String!
    sb.append(foo);
    sb.append("def"); // Creates String!
    sb.append(Integer.toString(47));
    System.out.println(sb);
  }
} ///:~
```

In the creation of **String s**, the compiler is actually doing the rough equivalent of the subsequent code that uses **sb**: a **StringBuffer** is created and **append( )** is used to add new characters directly into the **StringBuffer** object (rather than making new copies each time). While this is more efficient, it's worth noting that each time you create a quoted character string like **"abc"** and **"def"**, the compiler actually turns those into **String** objects. Thus, there may be more objects created than you expect, despite the efficiency afforded through **StringBuffer**.

the String and StringBuffer classes

Here is an overview of the methods available for both **String** and **StringBuffer**, so you can get a feel for the way they interact. These tables don't contain every single method, but rather the ones that are important to this discussion. Methods that are overloaded are summarized in a single row.

First, the **String** class:

Method	Arguments, Overloading	Use
Constructor	Overloaded: Default, **String**, **StringBuffer,** **char** arrays, **byte** arrays.	Creating **String** objects.
length( )		Number of characters in **String**.
charAt()	**int Index**	The **char** at a location in the **String**.
getChars( ), **getBytes( )**	The beginning and end from which to copy, the array to copy into, an index into the destination array.	Copy **char**s or **bytes** into an external array.
toCharArray( )		Produces a **char[]** containing the characters in the **String**.
equals( ), **equalsIgnoreCase( )**	A **String** to compare to.	An equality check on the contents of the two **String**s.
compareTo( )	A **String** to compare to.	Result is negative, zero or positive depending on the lexicographical ordering of the **String** and the argument.

Method	Arguments, Overloading	Use
		Uppercase and lowercase are not equal!
regionMatches( )	Offset into this **String**, the other **String** and its offset and length to compare. Overload adds "ignore case."	Boolean result indicates whether the region matches.
startsWith( )	**String** that it might start with. Overload adds offset into argument.	Boolean result indicates whether the **String** starts with the argument.
endsWith( )	**String** that might be a suffix of this **String**.	Boolean result indicates whether the argument is a suffix.
indexOf( ), **lastIndexOf( )**	Overloaded: **char**, **char** and starting index, **String**, **String** and starting index	Returns -1 if the argument is not found within this **String**, otherwise returns the index where the argument starts. **lastIndexOf( )** searches backward from end.
substring( )	Overloaded: Starting index, starting index and ending index.	Returns a new **String** object containing the specified character set.
concat( )	The **String** to concatenate	Returns a new **String** object containing the original **String**'s characters followed by the characters in the argument.
replace( )	The old character to search for, the new character to replace it with.	Returns a new **String** object with the replacements made. Uses the old **String** if no match is found.
toLowerCase( ), **toUpperCase( )**		Returns a new **String** object with the case of all letters changed. Uses the old **String** if no changes need to be made.
trim( )		Returns a new **String** object with the whitespace removed from each end. Uses the old **String** if no changes need to be made.
valueOf( )	Overloaded: **Object**, **char[]**, **char[]** and offset and count, **boolean**, **char**, **int**, **long**, **float**, **double**.	Returns a **String** containing a character representation of the argument.
intern( )		Produces one and only one **String** handle for each unique character sequence

You can see that every **String** method carefully returns a new **String** object when it's necessary to change the contents. Also notice that if the contents don't need changing the method will just return a handle to the original **String**. This saves storage and overhead.

Here's the **StringBuffer** class:

Method	Arguments, overloading	Use
Constructor	Overloaded: default, length of buffer to create, **String** to create from.	Create a new **StringBuffer** object.
toString( )		Creates a **String** from this **StringBuffer**.
length( )		Number of characters in the **StringBuffer**.
capacity( )		Returns current number of spaces allocated
ensureCapacity( )	Integer indicating desired capacity.	Makes the **StringBuffer** hold at least the desired number of spaces.
setLength( )	Integer indicating new length of character string in buffer.	Truncates or expands the previous character string. If expanding, pads with nulls.
charAt( )	Integer indicating the location of the desired element.	Returns the **char** at that location in the buffer.
setCharAt( )	Integer indicating the location of the desired element, and the new **char** value for the element.	Modifies the value at that location.
getChars( )	The beginning and end from which to copy, the array to copy into, an index into the destination array.	Copy **char**s into an external array. There's no **getBytes( )** as in **String**.
append( )	Overloaded: **Object**, **String**, **char[]**, **char[]** with offset and length, **boolean**, **char**, **int**, **long**, **float**, **double**.	The argument is converted to a string and appended to the end of the current buffer, increasing the buffer if necessary.
insert( )	Overloaded, each with a first argument of the offset at which to start inserting: **Object**, **String**, **char[]**, **boolean**, **char**, **int**, **long**, **float**, **double**.	The second argument is converted to a string and inserted into the current buffer beginning at the offset. The buffer is increased if necessary.
reverse( )		The order of the characters in the buffer is reversed.

The most commonly-used method is **append( )**, which is used by the compiler when evaluating **String** expressions containing the '**+**' and '**+=**' operators. The **insert( )** method has a similar form, and both methods perform significant manipulations to the buffer itself rather than creating new objects.

String s are special

By now you've seen that the **String** class is not just another class in Java – there are a lot of special cases in **String**, not the least of which is that it's a built-in class and fundamental to Java. Then there's the fact that a quoted character string is converted to a **String** by the compiler, and the special overloaded operators **+** and **+=**. In this chapter you've seen the remaining special case: the carefully-build immutability by using the companion **StringBuffer** and some extra magic in the compiler.

summary

Because everything is a handle in Java, and because every object is created on the heap and garbage collected only when it is no longer used, the flavor of object manipulation changes, especially when passing and returning objects. For example, in C or C++, if you wanted to initialize some piece of storage in a method, you'd probably request that the user pass the address of that piece of storage into the method. Otherwise you'd have to worry about who was responsible for the destruction of that storage. Thus the interface and understanding of such methods is more complicated. But in Java, you never have to worry about responsibility or whether an object will still exist when it is needed, since that is always taken care of for you. This means that your programs can create an object at the point that it is needed, and no sooner, and never worry about the mechanics of passing around responsibility for that object: you simply pass the handle. Sometimes the simplification that this provides is unnoticed, other times it is staggering.

The down side to all this underlying magic is twofold:

1. You always take the efficiency hit for the extra memory management, and there's always a slight amount of uncertainty about the time something can take to run (since the garbage collector could be forced into action at any point you get dangerously low on memory). For most applications, the benefits outweigh the drawbacks, and particularly time-critical sections can be written using **native** methods (see Appendix A).

2. Aliasing: sometimes you can accidentally end up with two handles to the same object, which is only a problem if both handles are assumed to point to a *distinct* object. This is where you need to pay a little closer attention and, when necessary, **clone( )** an object to prevent the other handle from being surprised by an unexpected change. Alternatively, you can support aliasing for efficiency by creating immutable objects whose operations may return a new object of the same type or some different type, but never change the original object so that anyone aliased to that object sees no change.

Some people say that cloning in Java is a botched design, and to heck with it, so they implement their own version of cloning[3] and never call the **Object.clone( )** method thus eliminating the need to implement **Cloneable** and catch the **CloneNotSupportedException**. This is certainly a reasonable approach and since **clone( )** is supported so rarely within the standard Java library itself, apparently a safe one as well. But as long as you don't call **Object.clone( )** you don't need to implement **Cloneable** or catch the exception so that would seem acceptable as well.

It's interesting to note that one of the "reserved but not implemented" keywords in Java is **byvalue**. After seeing the issues of aliasing and cloning, you can imagine that **byvalue** may someday be used to implement an automatic local copy in Java. This could eliminate the more complex issues of cloning and make coding in these situations simpler and more robust.

[3] Doug Lea, who was helpful in resolving this issue, suggested this to me, saying he just creates a function called **duplicate( )** for each class.

exercises

1. Create a class **myString** containing a **String** object that you initialize in the constructor using the constructor's argument. Add a **toString( )** method, and a method **concatenate( )** that appends a **String** object to your internal string. Implement **clone( )** in **myString**. Create two methods that each take a **String** handle as an argument and call **concatenate( )**, but in the second method call **clone( )** first. Test the two methods and show the different effects.

2. Change **CheckCloneable.java** so that all the **clone( )** methods catch the **CloneNotSupportedException** rather than passing it to the caller.

13

13: creating windows and applets

The original design goal of the graphical user interface (GUI) library in Java 1.0 was to allow the programmer to build a GUI that looks good on all platforms.

That goal was not achieved; instead the Java 1.0 *Abstract Window Toolkit* (AWT) produces a GUI that looks equally mediocre on all systems. In addition it's very restrictive: you can only use four fonts and you cannot access any of the more sophisticated GUI elements that may exist in your operating system (OS) since those elements may not exist in other environments. The Java 1.0 AWT programming model is also awkward and non-object-oriented.

Much of this situation has been improved in the Java 1.1 AWT along with the introduction of Java Beans, a component programming model that is particularly oriented towards the easy creation of visual programming environments. For building applications that are native to a particular OS and look like any other application in that OS by taking advantage of useful native elements, vendors can now provide easy-to-use specialized libraries. The way the new AWT handles events is now a much clearer, object-oriented approach. However, to support existing code the original AWT is still supported.

One of Java's primary design goals is to create *applets*, which are little programs that run inside a Web browser. Because they must be safe, applets are limited in what they can accomplish. However, they are a powerful tool in supporting client-side programming, a major issue for the Web.

Programming within an applet is so restrictive it's often referred to as being "inside the sandbox," since you always have someone – the Java run-time security system – watching over you. Java 1.1 offers digital signing for applets so you can choose to allow trusted applets to have access to your machine. However, you can also step outside the sandbox and write regular applications, in which case you can access the other features of your OS. We've been writing regular applications all along in this book, but they've been *console applications* without any graphical components. The AWT can also be used to build GUI interfaces for regular applications.

In this chapter you'll first learn the use of the original "old" AWT, which is still supported and used by many of the code examples that you will come across. In the second part of the chapter you'll learn about the structure of the "new" AWT in Java 1.1, and see how much better the event model is. You should use the new Java 1.1 AWT when you're creating new programs.

Most of the examples will show the creation of applets, not only because it's easier but also because that's where the AWT's primary usefulness may reside. In addition you'll see how things are different when you want to create a regular application using the AWT.

Please be aware that this is not a comprehensive glossary of all the methods for the described classes. This chapter will just get you started with the essentials. When you're looking for more sophistication, make sure you go to your information browser to look for the classes and methods that will solve your problem (if you're using a development environment your information browser may be built in; if you're using the JavaSoft JDK then you use your Web browser and start in the java root directory).

why use the AWT?

One of the problems with the "old" AWT that you'll learn about in this chapter is that it is a poor example of both object-oriented design and GUI development kit design. It throws us back into the dark ages of programming (some suggest that the 'A' in AWT stands for "awkward," "awful," "abominable," etc.). You must write lines of code to do *everything*, including tasks that are accomplished much more easily using *resources* in other environments.

Many of these problems are reduced or eliminated in Java 1.1 because:

1. The new AWT in Java 1.1 is a much better programming model and a significant step towards a better library. Java Beans is the framework for that library.

2. "GUI builders" will become *de rigeur* for all development systems. Java Beans and the new AWT allow the GUI builder to write code for you as you place components onto forms using graphical tools. Other component technologies like ActiveX will be supported in the same fashion.

So why learn to use the old AWT? "Because it's there." In this case, "there" has a much more ominous meaning and points to a tenet of object-oriented library design: *once you publicize a component in your library, you can never take it out*. If you do, you'll wreck somebody's existing code. In addition, there are many existing code examples out there that you'll read as you learn about Java, all using the old AWT.

The AWT must reach into the GUI components of the native OS, which means that it performs a task that an applet cannot otherwise accomplish. An untrusted applet cannot make any direct calls into an OS because otherwise it could do bad things to the user's machine. The only way an untrusted applet can access important functionality like "draw a window on the screen" is through calls in the standard Java library that's been specially ported and safety-checked for that machine. The original model that Sun created is that this "trusted library" will only be provided by the trusted vendor of the Java system in your Web browser, and that vendor will control what goes into it.

But what if you want to extend the system by adding a new component that accesses functionality in the OS? Waiting for Sun to decide that your extension should be incorporated into the standard Java library isn't going to solve your problem. The new model in Java 1.1 is "trusted code" or "signed code" whereby a special server verifies that a piece of code that you download is in fact "signed" by the stated author using a public-key encryption system. This way, you'll know for sure where the code comes from, that it's Bob's code and not just someone pretending to be Bob. This doesn't prevent Bob from making mistakes or doing something malicious, but it does prevent Bob from shirking responsibility – anonymity is what makes computer viruses possible. A digitally signed applet – a "trusted applet" – in Java 1.1 *can* reach into your machine and manipulate it directly, just like any other application you get from a "trusted" vendor and install onto your computer.

But the point of all this is that the old AWT is *there*. There will always be old AWT code floating around, and new Java programmers learning from old books will encounter that code. Also, the old AWT is worth studying as an example of poor library design. The coverage of the old AWT given here will be relatively painless since it won't go into depth and enumerate every single method and class, but instead give you an overview of the old AWT design.

the basic applet

Libraries are often grouped according to their functionality. Some libraries, for example, are used as is, off the shelf. The standard Java library **String** and **Vector** classes are examples of these. Other libraries are designed specifically as building blocks to build other classes. A certain class of library is the *application framework*, whose goal is to help you build applications by providing a class or set of classes that produces the basic behavior that you need in every application of a particular type. Then, to customize the behavior to your own needs you inherit from the application class and override the methods of interest. The application framework's default control mechanism will call your overridden methods at the appropriate time. An application framework is a good example of "separating the things that change from the things that stay the same," since it attempts to localize all the unique parts of a program in the overridden methods.

applets are built using an application framework. You inherit from class **Applet** and override the appropriate methods. Most of the time you'll only be concerned with a few important methods which have to do with how the applet is built and used on a Web page. These methods are:

Method	Operation
init( )	Called when the applet is first created, to perform first-time initialization of the applet
destroy( )	Called when the applet is being unloaded from the page, to perform final release of resources when the applet is no longer used
stop( )	Called every time the applet moves out of sight on the Web browser, to allow the applet to shut off expensive operations. Also called right before **destroy( )**.
start( )	Called every time the applet moves into sight on the Web browser, to allow the applet to start up its normal operations (especially those that are shut off by **stop( )**). Also called after **init( )**.
paint( )	Part of the base class **Component** (three levels of inheritance up). Called as part of an **update( )** to perform special painting on the canvas of an applet.

For fun, consider the last method, **paint( )**. This method is called automatically when the **Component** (in this case, the applet) decides it needs to update itself – perhaps because it's being moved back onto the screen or placed on the screen for the first time, or because some other window had been temporarily placed over your Web browser. The applet calls its **update( )** method (defined in the base class **Component**) which goes about restoring everything, and as a part of that restoration calls

paint( ). You don't have to override **paint( )** but it turns out to be an easy way to make a very simple applet, so we'll start out with **paint( )**.

When **update( )** calls **paint( )** it hands it a handle to a **Graphics** object that represents the surface on which you can paint. This is important because you're limited to the surface of that particular component and thus cannot paint outside that area, which is a good thing otherwise you'd be painting outside the lines. In the case of an applet, the surface is the area inside the applet itself.

The **Graphics** object also has a set of operations you can perform on it. These operations revolve around painting on the canvas, so most of them have to do with drawing images, shapes, arcs, etc. (note that you can look all this up in your online Java documentation if you're curious). There are some methods that allow you to draw characters, however, and the most commonly-used one of these is **drawString( )**. For this, you must specify the **String** you want to draw and its starting location on the applet's drawing surface. This location is given in pixels, so it will look different on different machines, but at least it's portable.

With this information you can create a very simple applet:

```
//: Applet1.java
// Very simple applet
import java.awt.*;
import java.applet.*;

public class Applet1 extends Applet {
  public void paint(Graphics g) {
    g.drawString("First applet", 10, 10);
  }
} ///:~
```

Notice that applets have no **main( )**. That's all wired in to the application framework; you put any startup code in **init( )**.

To run this program you must place it inside a Web page and view that page inside your Java-enabled Web browser. To place an applet inside a Web page you put a special tag inside the HTML source for that Web page[1], to tell the page how to load and run the applet. This is the **applet** tag, and it looks like this for Applet1:

```
<applet
code=Applet1
width=200
height=200>
</applet>
```

The **code** value gives the name of the **.class** file where the applet resides. The **width** and **height** specify the initial size of the applet (in pixels, as before). There are other items you can place within the applet tag: a place to find other **.class** files on the Internet (**codebase**), alignment information (**align**), A special name for an applet so you tell another applet that name and they can communicate (**name**), and applet parameters to provide information that the applet can retrieve. Parameters are in the form
<param name="identifier" value = "information">
and there can be as many as you want.

But for simple applets all you need to do is place an applet tag in the above form inside your Web page and that will load and run the applet.

The above example isn't too thrilling, so let's try adding a slightly more interesting graphic component:

---

[1] It is assumed that the reader is familiar with the basics of HTML. It's not too hard to figure out, and there are lots of books and resources.

```
//: Applet2.java
// Easy graphics
import java.awt.*;
import java.applet.*;

public class Applet2 extends Applet {
  public void paint(Graphics g) {
    g.drawString("Second applet", 10, 15);
    g.draw3DRect(0, 0, 100, 20, true);
  }
} ///:~
```

This puts a box around the string. Of course, all the numbers are hard-coded and are based on pixels, so on some machines the box will fit nicely around the string and on others it will probably be off, because fonts will be different on different machines.

There are other interesting things you can find in the documentation for the **Graphic** class. Any sort of graphics activity is usually entertaining, so further experiments of this sort are left to the reader.

making a Button

Making a button is quite simple: you just call the **Button** constructor with the label you want on the button (you can also use the default constructor if you want a button with no label, but this is not very useful). Normally you'll want to create a handle for the button so you can refer to it later.

The **Button** is a component, like its own little window that will automatically get repainted as part of an update. This means that you don't explicitly paint a button or any other kind of control; you simply place them on the form and let them automatically take care of painting themselves. Thus to place a button on a form you override **init( )** instead of overriding **paint( )**:

```
//: Button1.java
// Putting buttons on an applet
import java.awt.*;
import java.applet.*;

public class Button1 extends Applet {
  Button b1, b2;
  public void init() {
    b1 = new Button("Button 1");
    b2 = new Button("Button 2");
    add(b1);
    add(b2);
  }
} ///:~
```

It's not enough to create the **Button** (or any other control). You must also call the **Applet add( )** method to cause the button to be placed on the applet's form. This seems a lot simpler than it is, because the call to **add( )** actually decides, implicitly, where to place the control on the form. Controlling the layout of a form is examined shortly.

capturing an event

You'll notice that if you compile and run the above applet, nothing happens when you press the buttons. This is where you must step in and write some code to say what will happen. The basis of event-based programming, which comprises a lot of what a GUI is about, is tying events to code that responds to those events.

After working your way this far through the book and grasping some of the fundamentals of object-oriented programming, you may think that of course there will be some sort of object-oriented approach to handling events. For example, you might have to inherit each button and override some "button pressed" method (this, it turns out, is too tedious and restrictive). You might also think there's some master "event" class that contains a method for each event you want to respond to.

Before objects, the typical approach to handling events was the "giant switch statement." Each event would have a unique integer value and inside the master event handling method you'd write a **switch** on that value.

The AWT doesn't use any object-oriented approach, and instead of a giant **switch** statement (which relies on the assignment of numbers to events) you must create a cascaded set of **if** statements. What you're trying to do with the **if** statements is match on the type of object that was the *target* of the event. That is, if you click on a button, then that particular button is the target, so what you must do inside your **if** statement is typically to figure out who was the target. Typically that's all you care about – if a button is the target of an event, then it was most certainly a mouse click and you can continue based on that assumption (events may also contain other information; for example, if you want to find out the pixel location where a mouse click occurred so you can draw a line to that location the **Event** object will contain the location).

The method inside an **Applet** subclass where your cascaded **if** statement resides is called **action( )**. There are two arguments: the first is of type **Event** and it contains all the information about the event that triggered this call to **action( )**. For example, it could be a mouse click, a normal keyboard press or release, a special method key press or release, the fact that the component got or lost the focus, mouse movements or drags, etc. The second argument is, in the normal case, the target of the event, which you'll often ignore. In addition, the second argument is also encapsulated within the **Event** object and so it is redundant although you may choose not to extract it from the **Event** object (it seems to be given as a second argument to **action( )** for other programming situations and not for regular applets or applications).

The situations where **action( )** gets called are extremely limited: when you place controls on a form, some types of controls (buttons, check boxes, drop-down lists, menus) have a "standard action" that occurs, which causes the call to **action( )** with the appropriate **Event** object. For example, with a button the **action( )** method is called when the button is pressed, and at no other time. Normally this is just fine, since that's what you ordinarily look for with a button. However, it's possible to deal with many other types of events via the **handleEvent( )** method as we shall see later in this chapter.

The previous example can be extended to handle button clicks as follows:

```
//: Button2.java
// Capturing button presses
import java.awt.*;
import java.applet.*;

public class Button2 extends Applet {
  Button b1, b2;
  public void init() {
    b1 = new Button("Button 1");
    b2 = new Button("Button 2");
    add(b1);
    add(b2);
  }
  public boolean action(Event evt, Object arg) {
    if(evt.target.equals(b1))
      getAppletContext().showStatus("Button 1");
    else if(evt.target.equals(b2))
      getAppletContext().showStatus("Button 2");
    // Let the base class handle it:
    else
      return super.action(evt, arg);
    return true; // We've handled it here
```

```
  }
} ///:~
```

To see what the target is, you ask the **Event** object what its **target** member is and then use the **equals( )** method to see if it matches the target object handle you're interested in. When you've written handlers for all the objects you're interested in you must call **super.action(evt, arg)** in the **else** statement at the end, as shown above. Remember from Chapter 7 (polymorphism) that what happens is your overridden method is called instead of the base class version. However, the base-class version contains code to handle all the cases that you're not interested in, and it won't get called unless you do so explicitly. The return value indicates whether you've handled it or not, so if you do match an event you should return true, otherwise return whatever the base-class **event( )** returns.

For this example, the simplest action is just to print what button is pressed. Some systems allow you to pop up a little window with a message in it, but applets discourage this. However, you can put a message at the bottom of the Web browser window on its *status line* by calling the **Applet** method **getAppletContext( )** to get access to the browser and then **showStatus( )** to put a string on the status line. This is really only useful for testing and debugging since the browser itself may overwrite your message.

Strange as it may seem, you can also match an event to the *text* that's on a button through the second argument in **event( )**. Using this technique, the above example becomes:

```
//: Button3.java
// Matching events on button text
import java.awt.*;
import java.applet.*;

public class Button3 extends Applet {
  Button b1, b2;
  public void init() {
    b1 = new Button("Button 1");
    b2 = new Button("Button 2");
    add(b1);
    add(b2);
  }
  public boolean action (Event evt, Object arg) {
    if(arg.equals("Button 1"))
      getAppletContext().showStatus("Button 1");
    else if(arg.equals("Button 2"))
      getAppletContext().showStatus("Button 2");
    // Let the base class handle it:
    else
      return super.action(evt, arg);
    return true; // We've handled it here
  }
} ///:~
```

It's difficult to know exactly what the **equals( )** method is doing here, but despite that the biggest problem with this approach is that it's very easy to misspell or get the capitalization wrong, and if you change the text of the button the code will no longer work (but you won't get any compile-time or run-time error messages). You should avoid this approach if possible.

text fields

A **TextField** is a one-line area that allows the user to enter and edit text. **TextField** is inherited from **TextComponent** which lets you select text, get the selected text as a **String**, get or set the text, set whether the **TextField** is editable or not, along with other associated methods that you can find in your reference. The following example demonstrates some of the functionality of a **TextField**; you can see that the method names are fairly obvious:

```
//: TextField1.java
// Using the text field control
import java.awt.*;
import java.applet.*;

public class TextField1 extends Applet {
  Button b1, b2;
  TextField t;
  String s = new String();
  public void init() {
    b1 = new Button("Get Text");
    b2 = new Button("Set Text");
    t = new TextField("Starting text", 30);
    add(b1);
    add(b2);
    add(t);
  }
  public boolean action (Event evt, Object arg) {
    if(evt.target.equals(b1)) {
      getAppletContext().showStatus(t.getText());
      s = t.getSelectedText();
      if(s.length() == 0) s = t.getText();
      t.setEditable(true);
    }
    else if(evt.target.equals(b2)) {
      t.setText("Inserted by Button 2: " + s);
      t.setEditable(false);
    }
    // Let the base class handle it:
    else
      return super.action(evt, arg);
    return true; // We've handled it here
  }
} ///:~
```

There are several ways to construct a **TextField**; the one shown here provides an initial string and sets the size of the field in characters.

Pressing button 1 either gets the text you've selected with the mouse or it gets all the text in the field, and places the result in **String s**. It also allows the field to be edited. Pressing button 2 puts a message and **s** into the text field and prevents the field from being edited (although you can still select the text). The editability of the text is controlled by passing **setEditable( )** a **true** or **false**.

text areas

A **TextArea** is like a **TextField** except that it can have multiple lines and has significantly more functionality. In addition to what you can do with a **TextField**, you can also append text as well as insert or replace text at a given location. It seems like this functionality could be useful for **TextField** as well so it's a little confusing to try to detect how the distinction is made. You might think that if you want **TextArea** functionality everywhere you can simply use a one-line **TextArea** in places you would otherwise use a **TextField**. However, you also get scroll bars with a **TextArea** even when they're not appropriate; that is, you'll get both vertical and horizontal scroll bars for a one-line **TextArea**. The following example shows both one-line and multi-line **TextArea**s:

```
//: TextArea1.java
// Using the text area control
import java.awt.*;
import java.applet.*;
```

```
public class TextArea1 extends Applet {
  Button b1 = new Button("Text Area 1");
  Button b2 = new Button("Text Area 2");
  Button b3 = new Button("Replace Text");
  Button b4 = new Button("Insert Text");
  TextArea t1 = new TextArea("t1", 1, 30);
  TextArea t2 = new TextArea("t2", 4, 30);
  public void init() {
    add(b1);
    add(t1);
    add(b2);
    add(t2);
    add(b3);
    add(b4);
  }
  public boolean action (Event evt, Object arg) {
    if(evt.target.equals(b1))
      getAppletContext().showStatus(t1.getText());
    else if(evt.target.equals(b2)) {
      t2.setText("Inserted by Button 2");
      t2.appendText(": " + t1.getText());
      getAppletContext().showStatus(t2.getText());
    }
    else if(evt.target.equals(b3)) {
      String s = " Replacement ";
      t2.replaceText(s, 3, 3 + s.length());
    }
    else if(evt.target.equals(b4)) {
      t2.insertText(" Inserted ", 10);
    }
    // Let the base class handle it:
    else
      return super.action(evt, arg);
    return true; // We've handled it here
  }
} ///:~
```

This example also uses a different style: the controls are all created at the point of definition and the **init( )** method only adds them to the applet.

There are several different **TextArea** constructors, but the one shown here gives a starting string and the number of rows and columns. The different buttons show appending, replacing and inserting text.

labels

A **Label** does exactly what it sounds like: places a label on the form. This is particularly important for text fields and text areas which don't have labels of their own, and can also be useful if you simply want to place textual information on a form. You can, as shown in the first example in this chapter, use **drawString( )** inside **paint( )** to place text in an exact location; when you use a **Label** it allows you to (approximately) associate the text with some other component via the layout manager (which shall be discussed later in this chapter).

With the constructor you can create a blank label, a label with initial text in it (which is what you'll typically do) and a label with an alignment of **CENTER**, **LEFT** or **RIGHT** (**static final int**s defined in class **Label**). You can also change the label and its alignment with **setText( )** and **setAlignment( )**, and if you've forgotten what you've set these to you can read the values with **getText( )** and **getAlignment( )**. This example shows what you can do with labels:

```
//: Label1.java
```

```
// Using labels
import java.awt.*;
import java.applet.*;

public class Label1 extends Applet {
  TextField t1 = new TextField("t1", 10);
  Label labl1 = new Label("TextField t1");
  Label labl2 = new Label("                        ");
  Label labl3 = new Label("                        ",
    Label.RIGHT);
  Button b1 = new Button("Test 1");
  Button b2 = new Button("Test 2");
  public void init() {
    add(labl1); add(t1);
    add(b1); add(labl2);
    add(b2); add(labl3);
  }
  public boolean action (Event evt, Object arg) {
    if(evt.target.equals(b1))
      labl2.setText("Text set into Label");
    else if(evt.target.equals(b2)) {
      if(labl3.getText().trim().length() == 0)
        labl3.setText("labl3");
      if(labl3.getAlignment() == Label.LEFT)
        labl3.setAlignment(Label.CENTER);
      else if(labl3.getAlignment() == Label.CENTER)
        labl3.setAlignment(Label.RIGHT);
      else if(labl3.getAlignment() == Label.RIGHT)
        labl3.setAlignment(Label.LEFT);
    }
    else
      return super.action(evt, arg);
    return true;
  }
} ///:~
```

The first use of the label is the most typical: labeling a **TextField** or **TextArea**. In the second part of the example, a bunch of empty spaces are reserved and when you press the "Test 1" button **setText( )** is used to insert text into the field. Because a number of blank spaces do not equal the same number of characters (in a proportionally-spaced font) you'll see that the text gets truncated when inserted into the label.

The third part of the example reserves empty space, then the first time you press the "Test 2" button it sees that there are no characters in the label (since **trim( )** removes all the blank spaces at each end of a **String**) and inserts a short label, which is initially right-aligned. The rest of the times you press the button it changes the alignment so you can see the effect.

You might think that you could create an empty label and then later put text in it with **setText( )**. However, you cannot put text into an empty label – presumably because it has zero width – and so creating a label with no text seems to be a useless thing to do. In the above example, the "blank" label is filled with empty spaces so it has enough width to hold text that's later placed inside.

Similarly, **setAlignment( )** has no effect on a label that you'd typically create with text in the constructor. The label width is the width of the text, so changing the alignment doesn't do anything. However, if you start with a long label and then change it to a shorter one, you can see the effect of the alignment.

These behaviors occur because of the default *layout manager* that's used for applets, which causes things to be squished together to their smallest size. Layout managers will be covered later in this chapter, at which time you'll see that other layout managers don't have the same effect.

check boxes

A check box provides a way to make a single on-off choice; it consists of a tiny box and a label. The box typically holds a little 'x' (or some other indication that it is set) or is empty depending on whether that item was selected.

You'll normally create a **Checkbox** using a constructor that takes the label as an argument. You can get and set the state, and also get and set the label if you want to read or change it after the **Checkbox** has been created. Note that the capitalization of **Checkbox** is inconsistent with the other controls, which may catch you by surprise.

Whenever a **Checkbox** is set or cleared an event occurs, which you can capture the same way you do a button. The following example uses a **TextArea** to enumerate all the check boxes that have been checked:

```
//: CheckBox1.java
// Using check boxes
import java.awt.*;
import java.applet.*;

public class CheckBox1 extends Applet {
  TextArea t = new TextArea(6, 20);
  Checkbox cb1 = new Checkbox("Check Box 1");
  Checkbox cb2 = new Checkbox("Check Box 2");
  Checkbox cb3 = new Checkbox("Check Box 3");
  public void init() {
    add(t); add(cb1); add(cb2); add(cb3);
  }
  public boolean action (Event evt, Object arg) {
    if(evt.target.equals(cb1))
      trace("1", cb1.getState());
    else if(evt.target.equals(cb2))
      trace("2", cb2.getState());
    else if(evt.target.equals(cb3))
      trace("3", cb3.getState());
    else
      return super.action(evt, arg);
    return true;
  }
  void trace(String b, boolean state) {
    if(state)
      t.appendText("\nCheck Box " + b + " Set");
    else
      t.appendText("\nCheck Box " + b + " Cleared");
  }
} ///:~
```

The **trace( )** method sends the name of the selected **Checkbox** and its current state to the **TextArea** using **appendText( )** so you'll see a cumulative list of which checkboxes were selected and what their state is.

radio buttons

The concept of a radio button in GUI programming comes from pre-electronic car radios with mechanical buttons: when you push one in, any other button that was pressed pops out. Thus it allows you to force a single choice among many.

The AWT does not have a separate class to represent the radio button; instead it reuses the **Checkbox**. However, to put the **Checkbox** in a radio button group (and to change it's shape so it's visually different than an ordinary **Checkbox**) you must use a special constructor which takes a **CheckboxGroup** object as an argument (you can also call **setCheckboxGroup( )** after the **Checkbox** has been created).

A **CheckboxGroup** has no constructor argument; its sole reason for existence is to collect together some **Checkbox**es into a group of radio buttons. One of the **Checkbox** objects must have its state set to **true** before you try to display the group of radio buttons, otherwise you'll get an exception at run time. If you try to set more than one radio button to **true** then only the last one set will be **true**.

Here's a simple example of the use of radio buttons. Notice that you capture radio button events like all others:

```
//: RadioButton1.java
// Using radio buttons
import java.awt.*;
import java.applet.*;

public class RadioButton1 extends Applet {
  TextField t = new TextField(30);
  CheckboxGroup g = new CheckboxGroup();
  Checkbox
    cb1 = new Checkbox("one", g, false),
    cb2 = new Checkbox("two", g, true),
    cb3 = new Checkbox("three", g, false);
  public void init() {
    t.setEditable(false);
    add(t);
    add(cb1); add(cb2); add(cb3);
  }
  public boolean action (Event evt, Object arg) {
    if(evt.target.equals(cb1))
      t.setText("Radio button 1");
    else if(evt.target.equals(cb2))
      t.setText("Radio button 2");
    else if(evt.target.equals(cb3))
      t.setText("Radio button 3");
    else
      return super.action(evt, arg);
    return true;
  }
} ///:~
```

To display the state, an edit field is used. This field is set to non-editable because it's only used to display data, not to collect it.

You can have any number of **CheckboxGroup**s on a form.

drop-down lists

Like a group of radio buttons, a drop-down list is a way to force the user to select only one element from a group of possibilities. However, it's a much more compact way to accomplish this, and it's easier to change the elements of the list without surprising the user (you can change radio buttons dynamically, but that tends to be visibly jarring).

Java's **Choice** box is not like the combo box in Windows, which lets you select from a list *or* type in your own selection. With a **Choice** box you choose one and only one element from the list. In the following example, the **Choice** box starts with a certain number of entries, and then new entries are

added to the box when a button is pressed. This allows you to see some interesting behaviors in **Choice** boxes:

```
//: Choice1.java
// Using drop-down lists
import java.awt.*;
import java.applet.*;

public class Choice1 extends Applet {
  String description[] = { "Ebullient", "Obtuse",
    "Recalcitrant", "Brilliant", "Somnescent",
    "Timorous", "Florid", "Putrescent" };
  TextField t = new TextField(30);
  Choice c = new Choice();
  Button b = new Button("Add items");
  int count = 0;
  public void init() {
    t.setEditable(false);
    for(int i = 0; i < 4; i++)
      c.addItem(description[count++]);
    add(t);
    add(c);
    add(b);
  }
  public boolean action (Event evt, Object arg) {
    if(evt.target.equals(c))
      t.setText("index: " +  c.getSelectedIndex() +
                "   " + (String)arg);
    else if(evt.target.equals(b)) {
      if(count < description.length)
        c.addItem(description[count++]);
    }
    else
      return super.action(evt, arg);
    return true;
  }
} ///:~
```

The **TextField** displays the "selected index" which is the sequence number of the currently selected element, as well as the **String** representation of the second argument of **action( )**, which is in this case the string that was selected.

When you run this applet, pay attention to the way in which the size of the **Choice** box is determined: in Windows, the size is fixed from the first time you drop down the list. This means that if you drop down the list, then add more elements to the list, the elements will be there but the drop-down list won't get any longer[2] (you can scroll through the elements). However, if you add all the elements before the first time the list is dropped down, then it will be sized correctly. Of course, the user will expect to see the whole list when it's dropped down, and so this behavior puts some significant limitations on adding elements to **Choice** boxes.

list boxes

List boxes are significantly different from **Choice** boxes, and not just in appearance. While a **Choice** box drops down when you activate it, a **List** occupies some fixed number of lines on a screen all the time, and doesn't change. In addition, a **List** allows multiple selection: if you click on more than one

[2] This behavior is apparently a bug and will be fixed in a later version of Java.

item the original item stays highlighted and you can select as many as you want. If you want to see the items in a list, you simply call **getSelectedItems( )** which produces an array of **String** of the item that's been selected. To remove an item from a group you have to click it again.

A problem with a **List** is that the default action is double-clicking, not single clicking. A single click adds or removes elements from the selected group, and a double click calls **action( )**. One way around this is to re-educate your user, which is the assumption made in the following program:

```
//: List1.java
// Using lists with action()
import java.awt.*;
import java.applet.*;

public class List1 extends Applet {
  String flavors[] = { "Chocolate", "Strawberry",
    "Vanilla Fudge Swirl", "Mint Chip",
    "Mocha Almond Fudge", "Rum Raisin",
    "Praline Cream", "Mud Pie" };
  // Show 6 items, allow multiple selection:
  List lst = new List(6, true);
  TextArea t = new TextArea(flavors.length, 30);
  Button b = new Button("test");
  int count = 0;
  public void init() {
    t.setEditable(false);
    for(int i = 0; i < 4; i++)
      lst.addItem(flavors[count++]);
    add(t);
    add(lst);
    add(b);
  }
  public boolean action (Event evt, Object arg) {
    if(evt.target.equals(lst)) {
      t.setText("");
      String[] items = lst.getSelectedItems();
      for(int i = 0; i < items.length; i++)
        t.appendText(items[i] + "\n");
    }
    else if(evt.target.equals(b)) {
      if(count < flavors.length)
        lst.addItem(flavors[count++], 0);
    }
    else
      return super.action(evt, arg);
    return true;
  }
} ///:~
```

When you press the button it adds items to the *top* of the list (because of the second argument 0 to **addItem( )**). Adding elements to a **List** is more reasonable than the **Choice** box because users expect to scroll a list box (for one thing, it has a built-in scroll bar) but they don't expect to have to figure out how to get a drop-down list to scroll, as in the previous example.

However, the only way for **action( )** to be called is through a double-click. If you need to monitor other activities that the user is doing on your **List** (in particular, single clicks) you must take an alternative approach.

handleEvent( )

So far we've been using **action( )**, but there's another method that gets first crack at everything: **handleEvent( )**. Any time an event happens, it happens "over" or "to" a particular object. The

handleEvent( ) method for that object is automatically called and an **Event** object is created and passed to **handleEvent( )**. The default **handleEvent( )** (which is defined in **Component**, the base class for virtually all the "controls" in the AWT) will call either **action( )** as we've been using, or other similar methods to indicate mouse activity, keyboard activity or to indicate that the focus has moved. We'll look at those in a later section in this chapter.

What if these other methods – **action( )** in particular – don't satisfy your needs? In the case of **List**, for example, when you want to catch single mouse clicks but **action( )** only responds to double clicks? The solution is to override **handleEvent( )** for your applet, which after all is derived from **Applet** and can therefore override any non-**final** methods. When you override **handleEvent( )** for the applet you're getting all the applet events before they are routed, so you cannot just assume "it's to do with my button so I can assume it's been pressed" since that's only true for **action( )**. Inside **handleEvent( )** it's possible that the button has the focus and someone is typing to it. Whether it makes sense or not, those are events that you can detect and act upon in **handleEvent( )**.

To modify the **List** example so that it will react to single mouse-clicks, the button detection will be left in **action( )** but the code to handle the **List** will be moved into **handleEvent( )** as follows:

```
//: List2.java
// Using lists with handleEvent()
import java.awt.*;
import java.applet.*;

public class List2 extends Applet {
  String flavors[] = { "Chocolate", "Strawberry",
    "Vanilla Fudge Swirl", "Mint Chip",
    "Mocha Almond Fudge", "Rum Raisin",
    "Praline Cream", "Mud Pie" };
  // Show 6 items, allow multiple selection:
  List lst = new List(6, true);
  TextArea t = new TextArea(flavors.length, 30);
  Button b = new Button("test");
  int count = 0;
  public void init() {
    t.setEditable(false);
    for(int i = 0; i < 4; i++)
      lst.addItem(flavors[count++]);
    add(t);
    add(lst);
    add(b);
  }
  public boolean handleEvent(Event evt) {
    if(evt.id == Event.LIST_SELECT ||
       evt.id == Event.LIST_DESELECT) {
      if(evt.target.equals(lst)) {
        t.setText("");
        String[] items = lst.getSelectedItems();
        for(int i = 0; i < items.length; i++)
          t.appendText(items[i] + "\n");
      }
    }
    else
      return super.handleEvent(evt);
    return true;
  }
  public boolean action(Event evt, Object arg) {
    if(evt.target.equals(b)) {
      if(count < flavors.length)
        lst.addItem(flavors[count++], 0);
    }
    else
```

```
      return super.action(evt, arg);
    return true;
  }
} ///:~
```

The example is the same as before except for the addition of **handleEvent( )**. Inside, a check is made to see whether a list selection or deselection has occurred. Now remember **handleEvent( )** is being overridden for the applet, so that means this occurrence could be anywhere on the form, so it could be happening to another list. Thus you must also check to see what the target is (although in this case there's only one list on the applet so we could have made the assumption that all list events must be about that list – this is bad practice since it's going to be a problem as soon as another list is added). If the list matches the one we're interested in the same code as before will do the trick.

Notice that the form for **handleEvent( )** is similar to **action( )**: if you deal with a particular event you **return true**, but if you're not interested in any of the other events via **handleEvent( )** you must **return super.handleEvent(evt)**. This is very important because if you don't, none of the other event-handling code will get called. For example, try commenting out the **return super.handleEvent(evt)** in the above code. You'll discover that **action( )** never gets called, certainly not what you want. Thus, for both **action( )** and **handleEvent( )** it's important to follow the above format.

In Windows, a list box automatically allows multiple selections if you hold down the shift key. This is nice because it allows the user to choose a single or multiple selection rather than fixing it during programming. You might think to be clever and implement this yourself by checking to see if the shift key is held down when a mouse click was made, by testing for **evt.shiftDown( )**. Alas, the design of the AWT stymies you – you'd have to be able to know which item was clicked on if the shift key *wasn't* pressed so you could deselect all the rest and select only that one. However, you cannot figure that out in Java 1.0 (Java 1.1 will send all mouse, keyboard and focus events to a **List**, so you'll be able to accomplish this).

controlling layout

The way you place components on a form in Java is probably different from any other GUI system you've used. First of all, it's all code; there are no "resources" that control placement of components. Secondly, the way components are placed on a form is controlled by a "layout manager" that decides how the components lie based on the order in which you **add( )** them. The size, shape and placement of components will be remarkably different from one layout manager to another. In addition, the layout managers adapt to the dimensions of your applet or application window, so if that window dimension is changed (for example, in the HTML page's applet specification) the size, shape and placement of the components may change.

Both the **Applet** and **Frame** classes are derived from **Container**, whose job it is to contain and display **Component**s (the **Container** itself is a **Component** so it can also react to events). In **Container** there's a method called **setLayout( )** that allows you to choose a different layout manager.

In this section we'll explore the various layout managers by placing buttons into them (since that's the simplest thing to do). There won't be any capturing of button events since this is just intended to show how the buttons are laid out.

FlowLayout

So far, all the applets that have been created seem to have laid out their components using some mysterious internal logic. That's because the applet uses a default layout scheme: the **FlowLayout**. This simply "flows" the components onto the form, from left to right until the top space is full, then moves down a row and continues flowing the components on.

Here's an example that explicitly (redundantly) sets the layout manager in an applet to **FlowLayout** and then places buttons on the form. You'll notice that with **FlowLayout** the components take on their "natural" size. A **Button**, for example, will be the size of its string.

```
//: FlowLayout1.java
// Demonstrating the FlowLayout
import java.awt.*;
import java.applet.*;

public class FlowLayout1 extends Applet {
  public void init() {
    setLayout(new FlowLayout());
    for(int i = 0; i < 20; i++)
      add(new Button("Button " + i));
  }
} ///:~
```

All components will be compacted to their smallest size in a **FlowLayout**, so you may get a little bit of surprising behavior. For example, a label will be the size of its string, so if you try right-justifying it you won't see any result.

BorderLayout

This layout manager has the concept of four border regions and a center area. When you add something to a panel that's using a **BorderLayout** you must use an **add( )** method that takes a **String** object as its first argument, and that string must specify (with proper capitalization) one of: "North" (top), "South" (bottom), "East" (right), "West" (left), or "Center." If you misspell or mis-capitalize, you won't get a compile-time error, but the applet simply won't do what you expect. Fortunately, as you shall see shortly, there's a much-improved approach in Java 1.1.

Here's a simple example:

```
//: BorderLayout1.java
// Demonstrating the BorderLayout
import java.awt.*;
import java.applet.*;

public class BorderLayout1 extends Applet {
  public void init() {
    int i = 0;
    setLayout(new BorderLayout());
    add("North", new Button("Button " + i++));
    add("South", new Button("Button " + i++));
    add("East", new Button("Button " + i++));
    add("West", new Button("Button " + i++));
    add("Center", new Button("Button " + i++));
  }
} ///:~
```

For every placement but "Center," the element that you add is compressed to fit in the smallest amount of space along one dimension while it is stretched to the maximum along the other dimension. "Center," however, spreads out along both dimensions to occupy the middle.

The **BorderLayout** is the default layout manager for applications and dialogs.

GridLayout

A **GridLayout** allows you to build a table of components, and as you add them they are placed, left-to-right and top-to-bottom, in the grid. In the constructor you specify the number of rows and columns that you need and these are laid out in equal proportions.

```
//: GridLayout1.java
// Demonstrating the FlowLayout
import java.awt.*;
import java.applet.*;
```

```
public class GridLayout1 extends Applet {
  public void init() {
    setLayout(new GridLayout(7,3));
    for(int i = 0; i < 20; i++)
      add(new Button("Button " + i));
  }
} ///:~
```

In this case there are 21 slots but only 20 buttons. The last slot is left empty; no "balancing" goes on with a **GridLayout**.

CardLayout

The **CardLayout** allows you to create the rough equivalent of a "tabbed dialog," which in more sophisticated environments has actual file-folder tabs running across one edge, and all you have to do is press a tab to bring forward a different dialog. Not so in the AWT: the **CardLayout** is simply a blank space and you're responsible for bringing forward new cards.

combining layouts

This example will combine more than one layout type, which seems rather difficult at first since only one layout manager can be operating for an applet or application. This is true, but if you create more **Panel** objects, each one of those **Panel**s can have its own layout manager and then be integrated into the applet or application as simply another component, using the applet or application's layout manager. This gives you much greater flexibility as seen in the following example:

```
//: CardLayout1.java
// Demonstrating the CardLayout
import java.awt.*;
import java.applet.Applet;

class ButtonPanel extends Panel {
  ButtonPanel(String id) {
    setLayout(new BorderLayout());
    add("Center", new Button(id));
  }
}

public class CardLayout1 extends Applet {
  Button
    first = new Button("First"),
    second = new Button("Second"),
    third = new Button("Third");
  Panel cards = new Panel();
  CardLayout cl = new CardLayout();
  public void init() {
    setLayout(new BorderLayout());
    Panel p = new Panel();
    p.setLayout(new FlowLayout());
    p.add(first);
    p.add(second);
    p.add(third);
    add("North", p);
    cards.setLayout(cl);
    cards.add("First card",
      new ButtonPanel("The first one"));
    cards.add("Second card",
      new ButtonPanel("The second one"));
    cards.add("Third card",
      new ButtonPanel("The third one"));
```

```
    add("Center", cards);
  }
  public boolean action(Event evt, Object arg) {
    if (evt.target.equals(first)) {
      cl.first(cards);
    }
    else if (evt.target.equals(second)) {
      cl.first(cards);
      cl.next(cards);
    }
    else if (evt.target.equals(third)) {
      cl.last(cards);
    }
    else
      return super.action(evt, arg);
    return true;
  }
} ///:~
```

This example begins by creating a new kind of **Panel**: a **ButtonPanel**. This only contains a single button, placed at the center of a **BorderLayout**, which means it will expand to fill the entire panel. The label on the button will let you know which panel you're on in the **CardLayout**.

In the applet, both the **Panel cards** where the cards will live and the layout manager **cl** for the **CardLayout** must be members of the class because you need to have access to those handles when you want to manipulate the cards.

The applet is changed to use a **BorderLayout** instead of its default **FlowLayout**, a **Panel** is created to hold three buttons (using a **FlowLayout**) and this panel is placed at the "North" end of the applet. The **cards** panel is added to the "Center" of the applet, thus effectively occupying the rest of the real estate.

When you add the **ButtonPanel**s (or whatever other components you want) to the panel of cards, the **add( )** method's first argument is not "North," "South," etc. Instead it's a string that describes the card. Although this string doesn't show up anywhere on the card itself, you can use it if you want to flip that card using the string. This approach is not used in **action( )**; instead the **first( )**, **next( )** and **last( )** methods are used. Check your documentation for the other approach.

In Java, the **CardLayout** is quite important because (as you'll see later) in applet programming the use of pop-up dialogs is heavily discouraged. This means that, for applets, the **CardLayout** is the only viable way for the applet to have a number of different forms that "pop up" on command.

GridBagLayout

Some time ago, it was believed that all the stars, planets, the sun and the moon revolved around the earth. It seemed intuitive, from observation. But then astronomers became more sophisticated and started tracking the motion of individual objects, some of which seemed at times to go backwards in their paths. Since it was known that everything revolved around the earth, those astronomers spent large amounts of their time coming up with equations and theories to explain the motion of the stellar objects.

When trying to work with **GridBagLayout**, you can consider yourself the analog of one of those early astronomers. The basic precept (decreed, interestingly enough, by the designers at "Sun") is that everything should be done in code. The Copernican revolution (again dripping with irony, the discovery that the planets in the solar system revolve around the sun) is the use of *resources* to determine the layout and make the programmer's job easy. Until these are added to Java, you're stuck (to continue the metaphor) in the Spanish Inquisition of **GridBagLayout** and **GridBagConstraints**.

My recommendation is to avoid **GridBagLayout**. Instead, use the other layout managers and especially the technique of combining several panels using different layout managers within a single program – your applets won't look *that* different; at least not enough to justify the trouble that **GridBagLayout**

entails. For my part, it's just too painful and ridiculous to come up with an example for this (and I wouldn't want to encourage this kind of library design). Instead, I'll refer you to *Core Java* by Cornell & Horstmann (2nd ed., Prentice-Hall, 1997) to get started.

alternatives to action()

As noted previously, **action( )** isn't the only method that's automatically called by **handleEvent( )** once it sorts everything out for you. There are three other sets of methods that are called, and if you want to capture certain types of events (keyboard, mouse and focus events) all you have to do is override the provided method. These methods are defined in the base class **Component**, so they're available in virtually all the controls that you might place on a form. However, you should be very aware that this approach is deprecated in Java 1.1, so although you may see legacy code using this technique you should use the new Java 1.1 approaches (described later in this chapter) instead.

Component method	When it's called
action (Event evt, Object what)	When the "typical" event occurs for this component (for example, when a button is pushed or a drop-down list item is selected)
keyDown (Event evt, int key)	A key is pressed when this component has the focus. The second argument is the key that was pressed and is redundantly copied from **evt.key**.
keyUp(Event evt, int key)	A key is released when this component has the focus.
lostFocus(Event evt, Object what)	The focus has moved away from the target. Normally, **what** is redundantly copied from **evt.arg**.
gotFocus(Event evt, Object what)	The focus has moved into the target.
mouseDown(Event evt, int x, int y)	A mouse down has occurred over the component, at the coordinates **x, y**.
mouseUp(Event evt, int x, int y)	A mouse up has occurred over the component.
mouseMove(Event evt, int x, int y)	The mouse has moved while it's over the component.
mouseDrag(Event evt, int x, int y)	The mouse is being dragged after a **mouseDown** occurred over the component. All drag events are reported to the component where the **mouseDown** occurred until there is a **mouseUp**.
mouseEnter(Event evt, int x, int y)	The mouse wasn't over the component before, but now it is.
mouseExit(Event evt, int x, int y)	The mouse used to be over the component but now it isn't.

You can see that each method receives an **Event** object along with some information that you'll typically need when you're handling that particular situation – with a mouse event, for example, it's likely that you'll want to know the coordinates where the mouse event occurred. It's interesting to note that when **Component**'s **handleEvent( )** calls any of these methods (the typical case), the extra arguments are always redundant as they are contained within the **Event** object itself. In fact, if you look at the source code for **Component.handleEvent( )** you can see that it explicitly plucks the additional arguments out of the **Event** object (this might be considered inefficient coding in some languages, but remember that Java's focus is on safety, not necessarily speed).

To prove to yourself that these events are in fact being called and as an interesting experiment it's worth creating an applet that overrides each of the above methods (except for **action( )** which is used many other places in this chapter) and displays data about each of the events as they happen.

This example also shows you how to make your own button object, because that's what is used as the target of all the events of interest. You might first (naturally) assume that to make a new button, you'd inherit from **Button**. But this doesn't work. Instead, you inherit from **Canvas** (a much more generic component) and paint your button on that canvas by overriding the **paint( )** method. As you'll see, it's really too bad that overriding **Button** doesn't work, since there's a bit of code involved to paint the button (if you don't believe me, try exchanging **Button** for **Canvas** in this example, and remember to call the base-class constructor **super(label)**. You'll see that the button doesn't get painted and the events don't get handled).

The **myButton** class is very specific: it only works with an **AutoEvent** "parent window" (not a base class, but the window in which this button is created and lives). With this knowledge, **myButton** can reach into the parent window and manipulate its text fields, which is what's necessary to be able to write the status information into the fields of the parent. Of course this is a much more limited solution, since **myButton** can only be used in conjunction with **AutoEvent**. This kind of code is sometimes called "highly coupled." However, to make **myButton** more generic requires a lot more effort which isn't warranted for this example (and possibly for many of the applets that you will write). Again, keep in mind that the following code uses APIs that are deprecated in Java 1.1.

```
//: AutoEvent.java
// Alternatives to action()
import java.awt.*;
import java.applet.*;
import java.util.*;

class MyButton extends Canvas {
  AutoEvent parent;
  Color color;
  String label;
  MyButton(AutoEvent parent,
           Color color, String label) {
    this.label = label;
    this.parent = parent;
    this.color = color;
  }
  public void paint(Graphics  g) {
    g.setColor(color);
    int rnd = 30;
    g.fillRoundRect(0, 0, size().width,
                    size().height, rnd, rnd);
    g.setColor(Color.black);
    g.drawRoundRect(0, 0, size().width,
                    size().height, rnd, rnd);
    FontMetrics fm = g.getFontMetrics();
    int width = fm.stringWidth(label);
    int height = fm.getHeight();
    int ascent = fm.getAscent();
    int leading = fm.getLeading();
    int horizMargin = (size().width - width)/2;
    int verMargin = (size().height - height)/2;
    g.setColor(Color.white);
    g.drawString(label, horizMargin,
                 verMargin + ascent + leading);
  }
  public boolean keyDown(Event evt, int key) {
    TextField t =
      (TextField)parent.h.get("keyDown");
    t.setText(evt.toString());
```

```
    return true;
  }
  public boolean keyUp(Event evt, int key) {
    TextField t =
      (TextField)parent.h.get("keyUp");
    t.setText(evt.toString());
    return true;
  }
  public boolean lostFocus(Event evt, Object w) {
    TextField t =
      (TextField)parent.h.get("lostFocus");
    t.setText(evt.toString());
    return true;
  }
  public boolean gotFocus(Event evt, Object w) {
    TextField t =
      (TextField)parent.h.get("gotFocus");
    t.setText(evt.toString());
    return true;
  }
  public boolean mouseDown(Event evt,int x,int y){
    TextField t =
      (TextField)parent.h.get("mouseDown");
    t.setText(evt.toString());
    return true;
  }
  public boolean mouseDrag(Event evt,int x,int y){
    TextField t =
      (TextField)parent.h.get("mouseDrag");
    t.setText(evt.toString());
    return true;
  }
  public boolean mouseEnter(Event evt,int x,int y){
    TextField t =
      (TextField)parent.h.get("mouseEnter");
    t.setText(evt.toString());
    return true;
  }
  public boolean mouseExit(Event evt,int x,int y){
    TextField t =
      (TextField)parent.h.get("mouseExit");
    t.setText(evt.toString());
    return true;
  }
  public boolean mouseMove(Event evt,int x,int y){
    TextField t =
      (TextField)parent.h.get("mouseMove");
    t.setText(evt.toString());
    return true;
  }
  public boolean mouseUp(Event evt,int x,int y) {
    TextField t =
      (TextField)parent.h.get("mouseUp");
    t.setText(evt.toString());
    return true;
  }
}

public class AutoEvent extends Applet {
  Hashtable h = new Hashtable();
```

```
    String event[] = {
      "keyDown", "keyUp", "lostFocus",
      "gotFocus", "mouseDown", "mouseUp",
      "mouseMove", "mouseDrag", "mouseEnter",
      "mouseExit"
    };
    MyButton
      b1 = new MyButton(this, Color.blue, "test1"),
      b2 = new MyButton(this, Color.red, "test2");
    public void init() {
      setLayout(new GridLayout(event.length+1,2));
      for(int i = 0; i < event.length; i++) {
        TextField t = new TextField();
        t.setEditable(false);
        add(new Label(event[i], Label.CENTER));
        add(t);
        h.put(event[i], t);
      }
      add(b1);
      add(b2);
    }
  } ///:~
```

You can see the constructor uses the technique of using the same name for the argument as what it's assigned to, and differentiating between the two using **this**:

```
this.label = label;
```

The **paint( )** method starts out simple: it fills a "round rectangle" with the button's color, and then draws a black line around it. Notice the use of **size( )** to determine the width and height of the component (in pixels, of course). After this, **paint( )** seems quite complicated because there's a lot of calculation going on to figure out how to center the button's label inside the button using the "font metrics." You can get a pretty good idea of what's going on by looking at the method call, and it turns out that this is pretty stock code, so you can just cut and paste it when you want to center a label inside any component.

You can't understand exactly how the **keyDown( )**, **keyUp( )** etc. methods work until you look down at the **AutoEvent** class. This contains a **Hashtable** to hold the strings representing the type of event and the **TextField** where information about that event is held. Of course, these could have been made statically rather than putting them in a **Hashtable**, but I think you'll agree that it's a lot easier to use and change. In particular, if you need to add or remove a new type of event in **AutoEvent**, you simply add or remove a string in the **event[]** array – everything else happens automatically.

The place where you look up the strings is in the **keyDown( )**, **keyUp( )** etc. methods back in **MyButton**. Each of these methods uses the **parent** handle to reach back to the parent window. Since that parent is an **AutoEvent** it contains the **Hashtable h**, and the **get( )** method, when provided with the appropriate **String**, will produce a handle to an **Object** which we happen to know is a **TextField** – so it is cast to that. Then the **Event** object is converted to its **String** representation which is displayed in the **TextField**.

It turns out this example is rather fun to play with since you can really see what's going on with the events in your program.

applet restrictions

For safety's sake, applets are quite restricted and there are many things you can't do. You can generally answer the question of what an applet is able to do by looking at what it is *supposed* to do: extend the functionality of a Web page in a browser. Since, as a net surfer, you never really know if a Web page is

from a friendly place or not, you want any code that it runs to be very safe. So the biggest restrictions you'll notice are probably:

1) *An applet can't touch the local disk*. This means writing *or* reading, since you wouldn't want an applet to read and transmit important information about you across the Web. Writing is prevented, of course, since that would be an open invitation to a virus.

2) *An applet can't have menus.* This is probably less oriented towards safety and more towards reducing confusion. You may have noticed that an applet looks like it blends right in as part of a Web page; you often don't see the boundaries of the applet. There's no frame or title bar to hang the menu from, other than the one belonging to the Web browser. Perhaps the design could be changed to allow you to merge your applet menu with the browser menu, but that would not only be complicated but would probably get a bit too close to the edge of safety by allowing the applet to affect it's environment.

3) *Dialog boxes are "untrusted."* In Java, dialog boxes present a bit of a quandary. First of all, they're not exactly disallowed in applets but they're heavily discouraged: if you pop up a dialog box from within an applet you'll get an "untrusted applet" message attached to that dialog. This is because, in theory, it would be possible to fool the user into thinking that they're dealing with a regular native application and to get them to type in their credit card number which then goes across the Web (personally, I never type my credit card number into my computer. It does my bookkeeping, and more than that it doesn't need to know). After seeing the kinds of GUIs that the AWT produces you may have a hard time believing *anybody* could be fooled that way. But an applet is always attached to a Web page and visible within your Web browser, while a dialog box is detached so in theory it could be possible. As a result it will be very rare to see an applet that uses a dialog box.

Many applet restrictions may be relaxed for trusted applets (those signed by a trusted source) in newer browsers.

There are other issues when thinking about applet development, as well:

- Applets take longer to download since you must download the whole thing every time, including a separate server hit for each different class. Your browser may cache the applet, but there are no guarantees. One improvement in Java 1.1 is the JAR (Java ARchive) file that allows packaging all the applet components (including other **.class** files as well as images and sounds) together into a single compressed file that can be downloaded in a single server transaction. "Digital signing" (the ability to verify the creator of a class) is available for each individual entry in the JAR file.

- Because of security issues you must work harder to do certain things such as accessing databases and sending email. In addition, the security restrictions make accessing multiple hosts difficult, since everything has to be routed through the server which then becomes a performance bottleneck and a single failure point that can stop the entire process.

- An applet within the browser doesn't have the same kind of control that a native application does. For example, you can't have a modal dialog box within an applet, since the user can always switch the page. When the user *does* change from a Web page or even exit the browser, the results can be catastrophic for your applet – there's no way to save the state so if you're in the middle of a transaction or other operation the information can be lost. In addition, different browsers do different things to your applet when you leave a Web page so the results are essentially undefined.

applet advantages

If you can live within the restrictions, applets have definite advantages, especially when building client/server or other networked applications:

- There is no installation issue. An applet has true platform independence (including the ability to easily play audio files, etc.) so you don't need to make any changes in your code for different platforms nor does anyone have to perform any "tweaking" upon installation. In fact, installation is automatic every time the user loads the Web page along

with the applets, so updates happen silently and automatically. In traditional client/server systems, building and installing a new version of the client software is often a nightmare.

- Because of the security built into the core Java language and the applet structure, you don't have to worry about bad code causing damage to someone's system. This, along with the previous point, makes Java (as well as alternative client-side Web programming tools like JavaScript and VBscript) popular for so-called *Intranet* client/server applications that live only within the company and don't move out onto the Internet.
- Because applets are automatically integrated with HTML, you have a built-in platform-independent documentation system to support the applet. It's an interesting twist, since we're used to having the documentation part of the program rather than vice versa.

windowed applications

It's possible to see that for safety's sake you can only have very limited behavior within an applet. In a very real sense, the applet is a temporary extension to the Web browser so its functionality must be limited along with its knowledge and control. There are times, however, when you'd like to make a windowed program do something else than sit on a Web page, and perhaps you'd like it to do some of the things a "regular" application can do and yet have the vaunted instant portability provided by Java. In previous chapters in this book we've been making command-line applications, but in some operating environments (the Macintosh, for example) there isn't a command line. So for any number of reasons you'd like to build a windowed, non-applet program using Java. This is certainly a reasonable desire.

Think hard, though, before committing to Java at this point. As this section points out, a Java windowed application can have menus and dialog boxes (impossible or difficult with an applet), and yet you still sacrifice the native operating environment's look-and-feel when you use Java[3], and this may not be acceptable to your users. Someday, perhaps even someday soon, Java will have hooks and libraries that will allow you to make an application that preserves the look and feel of the underlying operating environment – even at the cost of introducing some non-portability to your code – and at that point you will be able to deliver applications that won't confound your users. But at this point, you'll have to make a decision about what's more important: cross-platform compatibility or native look-and-feel.

menus

It's impossible to put a menu on an applet, so they're only for regular applications. Go ahead, try it if you don't believe me and you're sure that it would make sense to have menus on applets. There's no **setMenuBar( )** method in **Applet** and that's the way a menu is attached.

There are four different types of **MenuComponent**, all derived from that abstract class: **MenuBar** (you can only have one **MenuBar** for a particular **Frame**), **Menu** to hold one individual drop-down menu or submenu, **MenuItem** to represent one single element on a menu, and **CheckboxMenuItem** which is derived from **MenuItem** and produces a checkmark to indicate whether that menu item is selected or not.

Unlike a system that uses resources, with Java and the AWT you must hand-assemble all the menus in source code. Here are the ice-cream flavors again, used to create menus:

```
//: Menu1.java
// Menus only work with applications.
// Shows submenus, checkbox menu items
// and swapping menus.
```

[3] This is changing somewhat with Java 1.1 and with various add-ons to the AWT like Microsoft's AFC and Javasoft's "Swing".

```
import java.awt.*;

public class Menu1 extends Frame {
  String flavors[] = { "Chocolate", "Strawberry",
    "Vanilla Fudge Swirl", "Mint Chip",
    "Mocha Almond Fudge", "Rum Raisin",
    "Praline Cream", "Mud Pie" };
  TextField t = new TextField("No flavor", 30);
  MenuBar mb1 = new MenuBar();
  Menu f = new Menu("File");
  Menu m = new Menu("Flavors");
  Menu s = new Menu("Safety");
  // Alternative approach:
  CheckboxMenuItem safety[] = {
    new CheckboxMenuItem("Guard"),
    new CheckboxMenuItem("Hide")
  };
  MenuItem file[] = {
    new MenuItem("Open"),
    new MenuItem("Exit")
  };
  // A second menu bar to swap to:
  MenuBar mb2 = new MenuBar();
  Menu fooBar = new Menu("fooBar");
  MenuItem other[] = {
    new MenuItem("Foo"),
    new MenuItem("Bar"),
    new MenuItem("Baz"),
  };
  Button b = new Button("Swap Menus");
  public Menu1() {
    for(int i = 0; i < flavors.length; i++) {
      m.add(new MenuItem(flavors[i]));
      // Add separators at intervals:
      if((i+1) % 3 == 0)
        m.addSeparator();
    }
    for(int i = 0; i < safety.length; i++)
      s.add(safety[i]);
    f.add(s);
    for(int i = 0; i < file.length; i++)
      f.add(file[i]);
    mb1.add(f);
    mb1.add(m);
    setMenuBar(mb1);
    t.setEditable(false);
    add("Center", t);
    // Set up the system for swapping menus:
    add("North", b);
    for(int i = 0; i < other.length; i++)
      fooBar.add(other[i]);
    mb2.add(fooBar);
  }
  public boolean handleEvent(Event evt) {
    if(evt.id == Event.WINDOW_DESTROY)
      System.exit(0);
    else
      return super.handleEvent(evt);
    return true;
  }
```

```
  public boolean action(Event evt, Object arg) {
    if(evt.target.equals(b)) {
      MenuBar m = getMenuBar();
      if(m == mb1) setMenuBar(mb2);
      else if (m == mb2) setMenuBar(mb1);
    }
    else if(evt.target instanceof MenuItem) {
      if(arg.equals("Open")) {
        String s = t.getText();
        boolean chosen = false;
        for(int i = 0; i < flavors.length; i++)
          if(s.equals(flavors[i])) chosen = true;
        if(!chosen)
          t.setText("Choose a flavor first!");
        else
          t.setText("Opening "+ s +". Mmm, mm!");
      }
      else if(arg.equals("Guard"))
        t.setText("Guard the Ice Cream! " +
          "Guarding is " + safety[0].getState());
      else if(arg.equals("Hide"))
        t.setText("Hide the Ice Cream! " +
          "Is it cold? " + safety[1].getState());
      // You can also use the other way of
      // matching to a string:
      else if(evt.target.equals(file[1]))
        System.exit(0);
      else
        t.setText(arg.toString());
    }
    else
      return super.action(evt, arg);
    return true;
  }
  public static void main(String args[]) {
    Menu1 f = new Menu1();
    f.resize(300,200);
    f.show();
  }
} ///:~
```

In this program I avoided the typical long lists of **add( )** calls for each menu, because that seemed like a lot of unnecessary typing. Instead I placed the menu items into arrays, and then simply stepped through each array calling **add( )** in a **for** loop. This also means that adding or subtracting a menu item is less tedious.

As an alternative approach (which I find less desirable since it requires more typing) the **CheckboxMenuItem**s are created in an array of handles **safety**; this is true for the arrays **file** and **other** as well.

This program creates not one but two **MenuBar**s to demonstrate that menu bars can be actively swapped while the program is running. You can see how a **MenuBar** is made up of **Menu**s, and each **Menu** is itself made up of **MenuItem**s, **CheckboxMenuItem**s, or even other **Menu**s (which produce submenus). When a **MenuBar** is assembled it can be installed into the current program with the **setMenuBar( )** method. Note that when the button is pressed, it checks to see which menu is currently installed using **getMenuBar( )**, then puts the other menu bar in it's place.

The rest of **action( )** deals with the various menu items, testing for "Open," "Guard" or "Hide" (note spelling and capitalization is critical and it is not checked, so this is a clear source of programming errors), and otherwise sending the string to the **TextField**. The checking and un-checking of the menu items is taken care of automatically, and the **getState( )** method reveals the state.

You might think that one menu could reasonably reside on more than one menu bar. This does seem to make sense because all you're passing to the **MenuBar add( )** method is a handle. However, if you try this the behavior will be strange, and not what you expect (it's difficult to know if this is a bug or if they intended it to work this way).

This example also shows what you need to do to create an application instead of an applet (again, because an application can support menus and an applet cannot). Instead of inheriting from **Applet**, you inherit from **Frame**. Instead of **init( )** to set things up, you make a constructor for your class. Finally, you create a **main( )** and in that you build an object of your new type, resize it and then call **show( )**. It's only different from an applet in a few small places, but it's now a standalone windowed application and you've got menus.

dialog boxes

A dialog box is a window that pops up out of another window. It's purpose is to deal with some specific issue without cluttering the original window with those details. Dialog boxes are heavily used in windowed programming environments, but as mentioned previously, rarely used in applets.

To create a dialog box, you inherit from **Dialog**, which is just another kind of **Window**, like a **Frame**. Unlike a **Frame**, a **Dialog** cannot have a menu bar or change the cursor, but other than that they're quite similar. A dialog has a layout manager (which defaults to **BorderLayout**) and you override **action( )** etc., or **handleEvent( )** to deal with events. One significant difference you'll want to pay attention to in **handleEvent( )**: when the **WINDOW_DESTROY** event occurs, you don't want to shut down the application! Instead, you release the resources used by the dialog's window by calling **dispose( )**.

In the following example, the dialog box is made up of a grid (using **GridLayout**) of a special kind of button which is defined here as class **ToeButton**. This button draws a frame around itself and, depending on its state, a blank, an "x" or an "o" in the middle. It starts out blank, and then depending on whose turn it is, changes to an "x" or an "o." However, it will also flip back and forth between "x" and "o" when you click on the button (this makes the tic-tac-toe concept only slightly more annoying than it already is). In addition, the dialog box can be set up for any number of rows and columns by changing numbers in the main application window.

```
//: ToeTest.java
// Demonstration of dialog boxes
// and creating your own components
import java.awt.*;

class ToeButton extends Canvas {
  int state = 0;
  ToeDialog parent;
  ToeButton(ToeDialog parent) {
    this.parent = parent;
  }
  public void paint(Graphics  g) {
    int x1 = 0;
    int y1 = 0;
    int x2 = size().width - 1;
    int y2 = size().height - 1;
    g.drawRect(x1, y1, x2, y2);
    x1 = x2/4;
    y1 = y2/4;
    int wide = x2/2;
    int high = y2/2;
    if(state == 1) {
      g.drawLine(x1, y1, x1 + wide, y1 + high);
      g.drawLine(x1, y1 + high, x1 + wide, y1);
    }
    if(state == 2) {
      g.drawOval(x1, y1, x1 + wide/2, y1 + high/2);
```

```
    }
  }
  public boolean mouseDown(Event evt, int x, int y) {
    if(state == 0) {
      state = parent.turn;
      parent.turn = (parent.turn == 1 ? 2 : 1);
    }
    else
      state = (state == 1 ? 2 : 1);
    repaint();
    return true;
  }
}

class ToeDialog extends Dialog {
  // w = number of cells wide
  // h = number of cells high
  static final int XX = 1;
  static final int OO = 2;
  int turn = XX; // Start with x's turn
  public ToeDialog(Frame parent, int w, int h) {
    super(parent, "The game itself", false);
    setLayout(new GridLayout(w, h));
    for(int i = 0; i < w * h; i++)
      add(new ToeButton(this));
    resize(w * 50, h * 50);
  }
  public boolean handleEvent(Event evt) {
    if(evt.id == Event.WINDOW_DESTROY)
      dispose();
    else
      return super.handleEvent(evt);
    return true;
  }
}

public class ToeTest extends Frame {
  TextField rows = new TextField("3");
  TextField cols = new TextField("3");
  public ToeTest() {
    setTitle("Toe Test");
    Panel p = new Panel();
    p.setLayout(new GridLayout(2,2));
    p.add(new Label("Rows", Label.CENTER));
    p.add(rows);
    p.add(new Label("Columns", Label.CENTER));
    p.add(cols);
    add("North", p);
    add("South", new Button("go"));
  }
  public boolean handleEvent(Event evt) {
    if(evt.id == Event.WINDOW_DESTROY)
      System.exit(0);
    else
      return super.handleEvent(evt);
    return true;
  }
  public boolean action(Event evt, Object arg) {
    if(arg.equals("go")) {
      Dialog d = new ToeDialog(
```

```
          this,
          Integer.parseInt(rows.getText()),
          Integer.parseInt(cols.getText()));
        d.show();
      }
      else
        return super.action(evt, arg);
      return true;
    }
    public static void main(String args[]) {
      Frame f = new ToeTest();
      f.resize(200,100);
      f.show();
    }
  } ///:~
```

The **ToeButton** class keeps a handle to its parent, which must be of type **ToeDialog**. As before, this introduces high coupling because a **ToeButton** can only be used with a **ToeDialog** but it solves a number of problems, and in truth it doesn't seem like such a bad solution because there's no other kind of dialog that's keeping track of whose turn it is. Of course, you can take another approach which is to make **ToeDialog.turn** a **static** member of **ToeButton**, which eliminates the coupling, but prevents you from having more than one **ToeDialog** at a time (more than one that works properly, anyway).

The **paint( )** method is concerned with the graphics: drawing the square around the button and drawing the "x" or the "o." This is full of tedious calculations, but it's straightforward.

A mouse click is captured by the overridden **mouseDown( )** method, which first checks to see if the button has anything written on it. If not, the parent window is queried to find out whose turn it is and that is used to establish the state of the button. Note that the button then reaches back into the parent and changes the turn. If the button is already displaying an "x" or an "o" then that is flopped. You can see in these calculations the convenient use of the ternary if-else described in Chapter 3. After a button state change, the button is repainted.

The constructor for **ToeDialog** is quite simple: it adds into a **GridLayout** as many buttons as you request, then resizes it for 50 pixels on a side for each button (if you don't resize a **Window**, it won't show up!). Notice that **handleEvent( )** just calls **dispose( )** for a **WINDOW_DESTROY** so the whole application doesn't go away.

ToeTest sets up the whole application by creating the **TextField**s (for inputting the rows and columns of the button grid) and the "go" button. You'll see in **action( )** that this program uses the less-desirable "string match" technique for detecting the button press (make sure you get spelling and capitalization right!). When the button is pressed, the data in the **TextField**s must be fetched and, since they are in **String** form, turned into **int**s using the **static Integer.parseInt( )** method. Once the **Dialog** is created, the **show( )** method must be called to display and activate it.

You'll note that the **ToeDialog** object is assigned to a **Dialog** handle **d**. This is an example of upcasting, although it really doesn't make much difference here since all that's happening is the **show( )** method is called. However, if you wanted to call some method that only existed in **ToeDialog** you would want to assign to a **ToeDialog** handle and not lose the information in an upcast.

file dialogs

Some operating systems have a number of special built-in dialog boxes to handle the selection of things like fonts, colors, printers and the like. Virtually all graphical operating systems support the opening and saving of files, however, and so Java's **FileDialog** encapsulates these for easy use. This, of course, makes no sense at all to use from an applet since an applet can neither read nor write files on the local disk (this will change for trusted applets in newer browsers).

The following application exercises the two forms of file dialogs, one for opening and one for saving. Most of the code should by now be familiar, and all the interesting activities happen in **action( )** for the two different button clicks:

```
//: FileDialogTest.java
// Demonstration of File dialog boxes
import java.awt.*;

public class FileDialogTest extends Frame {
  TextField filename = new TextField();
  TextField directory = new TextField();
  Button open = new Button("Open");
  Button save = new Button("Save");
  public FileDialogTest() {
    setTitle("File Dialog Test");
    Panel p = new Panel();
    p.setLayout(new FlowLayout());
    p.add(open);
    p.add(save);
    add("South", p);
    directory.setEditable(false);
    filename.setEditable(false);
    p = new Panel();
    p.setLayout(new GridLayout(2,1));
    p.add(filename);
    p.add(directory);
    add("North", p);
  }
  public boolean handleEvent(Event evt) {
    if(evt.id == Event.WINDOW_DESTROY)
      System.exit(0);
    else
      return super.handleEvent(evt);
    return true;
  }
  public boolean action(Event evt, Object arg) {
    if(evt.target.equals(open)) {
      // Two arguments, defaults to open file:
      FileDialog d = new FileDialog(this,
        "What file do you want to open?");
      d.setFile("*.java");
      d.setDirectory("."); // Current directory
      d.show();
      if(d != null) {
        filename.setText(d.getFile());
        directory.setText(d.getDirectory());
      }
    }
    else if(evt.target.equals(save)) {
      FileDialog d = new FileDialog(this,
        "What file do you want to save?",
        FileDialog.SAVE);
      d.setFile("*.java");
      d.setDirectory(".");
      d.show();
      if(d != null) {
        filename.setText(d.getFile());
        directory.setText(d.getDirectory());
      }
    }
    else
      return super.action(evt, arg);
    return true;
  }
```

```
  public static void main(String args[]) {
    Frame f = new FileDialogTest();
    f.resize(250,110);
    f.show();
  }
} ///:~
```

For an "open file" dialog, you use the constructor that takes two arguments; the first is the parent window handle and the second is the title for the title bar of the **FileDialog**. The method **setFile( )** provides an initial file name – presumably the native OS supports wildcards, so in this example all the **.java** files will initially be displayed. The **setDirectory( )** method chooses the directory in which the file selection will begin (generally the OS allows the user to change directories).

Now the behavior departs from a normal dialog, which is gone when it closes (the **show( )** command doesn't return until the dialog is closed). By some magic, when a **FileDialog** is closed by the user its data still exists, so you can read it – as long as the user didn't "cancel" out of the dialog, in which case the handle becomes **null**. You can see the test for **null** followed by the calls to **getFile( )** and **getDirectory( )**, the results of which are displayed in the **TextField**s.

The button for saving works the same way, except that it uses a different constructor for the **FileDialog**. This constructor takes three arguments, and the third argument must be either **FileDialog.SAVE** or **FileDialog.OPEN**.

the new AWT

In Java 1.1 a dramatic change has been accomplished in the creation of the new AWT. Most of this change revolves around the new event model used in Java 1.1: as bad and awkward and non-object-oriented as the old event model was, the new event model is possibly the most elegant I have seen, and it's difficult to understand how such a bad design (the old AWT) and such a good one (the new event model) could come out of the same group. This new way of thinking about events seems to drop so easily into your mind that the issue no longer becomes an impediment; instead it's a tool that helps you design the system. It's also essential for JavaBeans, which is described later in the chapter.

Instead of the non-object-oriented cascaded **if** statements in the old AWT, the new approach designates objects as "sources" and "listeners" of events. As you shall see, the use of inner classes is integral to the object-oriented nature of the new event model. In addition, events are now represented in a class hierarchy rather than a single class, and you can create your own event types.

You'll also find, if you've programmed with the old AWT, that Java 1.1 has made a number of what may seem like gratuitous name changes, for example **setSize( )** replaces **resize( )**. This will make sense when you learn about JavaBeans in a later chapter, because Beans use a particular naming convention so the names had to be modified to make the standard AWT components into Beans.

Java 1.1 continues to support the old AWT to ensure backward compatibility with existing programs. Without fully admitting disaster, the online documents for Java 1.1 list all the problems involved with programming the old AWT and describe how those problems are addressed in the new AWT.

Clipboard operations are supported in 1.1, although drag-and-drop "will be supported in a future release." You can access the desktop color scheme so your Java program can fit in with the rest of the desktop. Popup menus are available, and there are some improvements for graphics and images. Mouseless operation is supported. There is a simple API for printing and simplified support for scrolling.

the new event model

In the new event model, a component may initiate ("fire") an event. Each type of event is represented by a distinct class. When an event is fired, it is received by one or more "listeners," which act on that event. Thus the source of an event and the place where the event is handled may be separate.

Each event listener is an object of a class that implements a particular type of listener **interface**. So as a programmer, all you do is create a listener object and register it with the component that's firing the

event. This registration is performed by calling a **addXXXListener( )** method in the event-firing component, where **XXX** represents the type of event listened for. This means you can easily know what types of events can be handled by noticing the names of the addListener methods, and if you try to listen for the wrong events you'll find out your mistake at compile time. JavaBeans also utilizes the names of the addListener methods to determine what a Bean can do.

All of your event logic, then, will go inside a listener class. When you create a listener class, the only restriction is that it must implement the appropriate interface. You can create a global listener class, but this is a situation where inner classes tend to be quite useful, not only because they provide a logical grouping of your listener classes inside the UI or business logic classes they are serving, but because (as you shall see later) the fact that an inner class object keeps a handle to its parent object provides a very nice way to call across class and subsystem boundaries.

A simple example will make this clear. Consider the **Button2.java** example from earlier in this chapter.

```
//: Button2New.java
// Capturing button presses
import java.awt.*;
import java.awt.event.*; // Must add this
import java.applet.*;

public class Button2New extends Applet {
  Button b1, b2;
  public void init() {
    b1 = new Button("Button 1");
    b2 = new Button("Button 2");
    b1.addActionListener(new B1());
    b2.addActionListener(new B2());
    add(b1);
    add(b2);
  }
  class B1 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      getAppletContext().showStatus("Button 1");
    }
  }
  class B2 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      getAppletContext().showStatus("Button 2");
    }
  }
  /* The old way:
  public boolean action(Event evt, Object arg) {
    if(evt.target.equals(b1))
      getAppletContext().showStatus("Button 1");
    else if(evt.target.equals(b2))
      getAppletContext().showStatus("Button 2");
    // Let the base class handle it:
    else
      return super.action(evt, arg);
    return true; // We've handled it here
  }
  */
} ///:~
```

So you can compare the two approaches, the old code is left in as a comment. In **init( )**, the only change is the addition of the two lines:

```
b1.addActionListener(new B1());
b2.addActionListener(new B2());
```

addActionListerner( ) tells a button which object to activate when the button is pressed. The classes **B1** and **B2** are inner classes that implement the **interface ActionListener**. This interface contains a single method **actionPerformed( )** (meaning "this is the action that will be performed when the event is fired"). Notice that **actionPerformed( )** does not take a generic event, but rather a specific type of event, **ActionEvent**, so you don't need to bother testing and downcasting the argument if you want to extract specific **ActionEvent** information.

One of the nicest things about **actionPerformed( )** is how simple it is. It's just a method that gets called. Compare it to the old **action( )** method, where you must figure out what actually happened and act appropriately, and also worry about calling the base class version of **action( )** and return a value to indicate whether it's been handled or not. With the new event model you know that all the event-detection logic is taken care of so you don't have to figure that out; you just say what happens and you're done. If you're not already in love with this approach over the old one, you will be soon.

event and listener types

All the AWT components have been changed to include **addXXXListener( )** and **removeXXXListener( )** methods so that the appropriate types of listeners can be added and removed from each component. You'll notice that the "**XXX**" in each case also represents the argument for the method, for example **addFooListener(FooListener fl)**. The following table includes the associated events, listeners, methods and the components that support those particular events by providing the **addXXXListener( )** and **removeXXXListener( )** methods.

Event, listener interface and add- and remove-methods	Components supporting this event
ActionEvent **ActionListener** **addActionListener( )** **removeActionListener( )**	**Button**, **List**, **TextField,** **MenuItem** and its derivatives including **CheckboxMenuItem**, **Menu** and **PopupMenu**
AdjustmentEvent **AdjustmentListener** **addAdjustmentListener( )** **removeAdjustmentListener( )**	**Scrollbar** Anything you create that implements **Adjustable**
ComponentEvent **ComponentListener** **addComponentListener( )** **removeComponentListener( )**	**Component** and its derivatives, including **Button**, **Canvas**, **Checkbox**, **Choice**, **Container**, **Panel**, **Applet**, **ScrollPane**, **Window**, **Dialog**, **FileDialog,** **Frame**, **Label**, **List**, **Scrollbar**, **TextArea** and **TextField**
ContainerEvent **ContainerListener** **addContainerListener( )** **removeContainerListener( )**	**Container** and its derivatives, including **Panel**, **Applet**, **ScrollPane**, **Window**, **Dialog**, **FileDialog** and **Frame**
FocusEvent **FocusListener** **addFocusListener( )** **removeFocusListener( )**	**Component** and its derivatives, including **Button**, **Canvas**, **Checkbox**, **Choice**, **Container**, **Panel**, **Applet**, **ScrollPane**, **Window**, **Dialog**, **FileDialog,** **Frame Label**, **List**, **Scrollbar**, **TextArea** and **TextField**
KeyEvent **KeyListener** **addKeyListener( )**	**Component** and its derivatives, including **Button**, **Canvas**, **Checkbox**, **Choice**, **Container**, **Panel**, **Applet**, **ScrollPane**,

Event, listener interface and add- and remove-methods	Components supporting this event
removeKeyListener( )	**Window**, **Dialog**, **FileDialog,** **Frame**, **Label**, **List**, **Scrollbar**, **TextArea** and **TextField**
MouseEvent (for both clicks and motion) **MouseListener** **addMouseListener( )** **removeMouseListener( )**	**Component** and its derivatives, including **Button**, **Canvas**, **Checkbox**, **Choice**, **Container**, **Panel**, **Applet**, **ScrollPane**, **Window**, **Dialog**, **FileDialog,** **Frame**, **Label**, **List**, **Scrollbar**, **TextArea** and **TextField**
MouseEvent (for both clicks and motion) **MouseMotionListener** **addMouseMotionListener( )** **removeMouseMotionListener( )**	**Component** and its derivatives, including **Button**, **Canvas**, **Checkbox**, **Choice**, **Container**, **Panel**, **Applet**, **ScrollPane**, **Window**, **Dialog**, **FileDialog,** **Frame**, **Label**, **List**, **Scrollbar**, **TextArea** and **TextField**
WindowEvent **WindowListener** **addWindowListener( )** **removeWindowListener( )**	**Window** and its derivatives, including **Dialog**, **FileDialog** and **Frame**
ItemEvent **ItemListener** **addItemListener( )** **removeItemListener( )**	**Checkbox**, **CheckboxMenuItem**, **Choice**, **List** and anything that implements **ItemSelectable**
TextEvent **TextListener** **addTextListener( )** **removeTextListener( )**	Anything derived from **TextComponent**, including **TextArea** and **TextField**

You can see that each type of component only supports certain types of events. It's helpful to see the events supported by each component, as shown in the following table:

Component type	Events supported by this component
Adjustable	**AdjustmentEvent**
Applet	**ContainerEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent**
Button	**ActionEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent**
Canvas	**FocusEvent, KeyEvent, MouseEvent, ComponentEvent**
Checkbox	**ItemEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent**
CheckboxMenuItem	**ActionEvent, ItemEvent**
Choice	**ItemEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent**

Component type	Events supported by this component
Component	**FocusEvent, KeyEvent, MouseEvent, ComponentEvent**
Container	**ContainerEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent**
Dialog	**ContainerEvent, WindowEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent**
FileDialog	**ContainerEvent, WindowEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent**
Frame	**ContainerEvent, WindowEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent**
Label	**FocusEvent, KeyEvent, MouseEvent, ComponentEvent**
List	**ActionEvent, FocusEvent, KeyEvent, MouseEvent, ItemEvent, ComponentEvent**
Menu	**ActionEvent**
MenuItem	**ActionEvent**
Panel	**ContainerEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent**
PopupMenu	**ActionEvent**
Scrollbar	**AdjustmentEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent**
ScrollPane	**ContainerEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent**
TextArea	**TextEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent**
TextComponent	**TextEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent**
TextField	**ActionEvent, TextEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent**
Window	**ContainerEvent, WindowEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent**

Once you know what events a particular component supports, you don't need to look anything up to react to that event. You simply:

1. Take the name of the event class and remove the word "**Event**." Add the word "**Listener**" to what remains. This is the listener interface you need to implement in your inner class.

2. Implement the above interface, and write out the methods for the events you want to capture. For example, you may be looking for mouse movements, so you write code for the **mouseMoved( )** method of the **MouseMotionListener** interface (you'll have to implement the other methods, of course, but there's a shortcut for that which you'll see soon).

3. Create an object of the listener class in step 2. Register it with your component with the method produced by prepending "**add**" to your listener name. For example, **addMouseMotionListener( )**.

To finish what you need to know, here are the listener interfaces:

Listener interface w/ adapter	Methods in interface
ActionListener	**actionPerformed(ActionEvent)**
AdjustmentListener	**adjustmentValueChanged(AdjustmentEvent)**
ComponentListener **ComponentAdapter**	**componentHidden(ComponentEvent)** **componentShown(ComponentEvent)** **componentMoved(ComponentEvent)** **componentResized(ComponentEvent)**
ContainerListener **ContainerAdapter**	**componentAdded(ContainerEvent)** **componentRemoved(ContainerEvent)**
FocusListener **FocusAdapter**	**focusGained(FocusEvent)** **focusLost(FocusEvent)**
KeyListener **KeyAdapter**	**keyPressed(KeyEvent)** **keyReleased(KeyEvent)** **keyTyped(KeyEvent)**
MouseListener **MouseAdapter**	**mouseClicked(MouseEvent)** **mouseEntered(MouseEvent)** **mouseExited(MouseEvent)** **mousePressed(MouseEvent)** **mouseReleased(MouseEvent)**
MouseMotionListener **MouseMotionAdapter**	**mouseDragged(MouseEvent)** **mouseMoved(MouseEvent)**
WindowListener **WindowAdapter**	**windowOpened(WindowEvent)** **windowClosing(WindowEvent)** **windowClosed(WindowEvent)** **windowActivated(WindowEvent)** **windowDeactivated(WindowEvent)** **windowIconified(WindowEvent)** **windowDeiconified(WindowEvent)**
ItemListener	**itemStateChanged(ItemEvent)**
TextListener	**textValueChanged(TextEvent)**

using listener adapters for simplicity

In the above table, you can see that some listener interfaces have only one method. These are trivial to implement, since you'll only implement them when you want to write that particular method. However, the listener interfaces that have multiple methods could be less pleasant to use. For example, something you must always do when creating an application is provide a **WindowListener** to

the **Frame** so that when you get the **windowClosing( )** event you can call **System.exit(0)** to exit the application. But since **WindowListener** is an **interface**, you must implement all the other methods even if they don't do anything. This can be painful and annoying.

To solve the problem, each of the listener interfaces that have more than one method are provided with *adapters*, the names of which you can see in the above table. Each adapter provides default methods for each of the interface methods (alas, **WindowAdapter** does *not* have a default **windowClosing( )** that calls **System.exit(0)**). Then all you need to do is inherit from the adapter and override only the methods you need to change. For example, the typical **WindowListener** you'll use looks like this:

```
class MyWindowListener extends WindowAdapter {
  public void windowClosing(WindowEvent e) {
    System.exit(0);
  }
}
```

The whole point of the adapters is to make the creation of listener classes very easy.

There is a downside to adapters, however, in the form of a pitfall. Suppose you write a **WindowAdapter** like the above one:

```
class MyWindowListener extends WindowAdapter {
  public void WindowClosing(WindowEvent e) {
    System.exit(0);
  }
}
```

This doesn't work, but it will drive you crazy trying to figure out why, since everything will compile and run just fine – except that closing the window won't exit the program. Can you see the problem? It's in the name of the method: **WindowClosing( )** instead of **windowClosing( )**. A simple slip in capitalization results in the addition of a completely new method. However, this is not the method that's called when the window is closing, so you don't get the desired results.

making windows and applets with the Java 1.1 AWT

Often you'll want to be able to create a class that can be invoked as both a window and an applet. To accomplish this, you simply add a **main( )** to your applet that builds an instance of the applet inside a **Frame**. As a simple example, let's look at **Button2New.java** modified to work as both an application and an applet:

```
//: Button2NewB.java
// An application and an applet
import java.awt.*;
import java.awt.event.*; // Must add this
import java.applet.*;

public class Button2NewB extends Applet {
  Button b1, b2;
  TextField t = new TextField(20);
  public void init() {
    b1 = new Button("Button 1");
    b2 = new Button("Button 2");
    b1.addActionListener(new B1());
    b2.addActionListener(new B2());
    add(b1);
    add(b2);
    add(t);
  }
```

```
  class B1 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t.setText("Button 1");
    }
  }
  class B2 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t.setText("Button 2");
    }
  }
  // To close the application:
  static class WL extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
      System.exit(0);
    }
  }
  // A main() for the application:
  public static void main(String args[]) {
    Button2NewB applet = new Button2NewB();
    Frame aFrame = new Frame("Button2NewB");
    aFrame.addWindowListener(new WL());
    aFrame.add(applet, BorderLayout.CENTER);
    aFrame.setSize(300,200);
    applet.init();
    applet.start();
    aFrame.setVisible(true);
  }
} ///:~
```

The inner class **WL** and the **main( )** are the only two elements added to the applet, and the rest of the applet is untouched. In fact, you can generally copy and paste the **WL** class and **main( )** into your own applets with very little modification. The **WL** class is **static** so it can be easily created in **main( )** (remember that an inner class normally needs an outer class handle when it's created. Making it **static** eliminates this need). You can see that in **main( )**, the applet is explicitly initialized and started since in this case the browser isn't available to do it for you. Of course, this doesn't provide the full behavior of the browser, which also calls **stop( )** and **destroy( )**, but for most situations it's acceptable. If it's a problem, you can:

1. Make the handle **applet** a **static** member of the class (rather than a local variable of **main( )**).
2. Call **applet.stop( )** and **applet.destroy( )** inside **WindowAdapter.windowClosing( )** before you call **System.exit( )**.

Notice the last line:

```
aFrame.setVisible(true);
```

This is one of the changes in the Java 1.1 AWT: the **show( )** method is deprecated, and **setVisible(true)** replaces it. These sorts of seemingly capricious changes will make more sense when you learn about JavaBeans later in the chapter.

This example is also modified to use a **TextField** rather than printing to the console or to the browser status line. One restriction to making a program that's both an applet and an application is that you must choose input and output forms that work for both situations.

There's another small new feature of the Java 1.1 AWT shown here. You no longer need to use the error-prone approach of specifying **BorderLayout** positions using a **String**. When adding an element to a **BorderLayout** in Java 1.1, you can say:

```
aFrame.add(applet, BorderLayout.CENTER);
```

You name the location with one of the **BorderLayout** constants, which can then be checked at compile-time (rather than just quietly doing the wrong thing, as with the old form). This is a definite improvement, and shall be used throughout the rest of the book.

revisiting the earlier examples

So you can see a number of examples using the new event model and so you can study the way a program can be converted from the old to the new event model, the following examples revisit many of the issues demonstrated in the first part of this chapter using the old event model. In addition, each program is now both an applet and an application so you can run it with or without a browser.

text fields

This is similar to **TextField1.java**, but it adds significant extra behavior:

```
//: TextNew.java
// Text fields with Java 1.1 events
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class TextNew extends Applet {
  Button
    b1 = new Button("Get Text"),
    b2 = new Button("Set Text");
  TextField
    t1 = new TextField(30),
    t2 = new TextField(30),
    t3 = new TextField(30);
  String s = new String();
  public void init() {
    b1.addActionListener(new B1());
    b2.addActionListener(new B2());
    t1.addTextListener(new T1());
    t1.addActionListener(new T1A());
    t1.addKeyListener(new T1K());
    add(b1);
    add(b2);
    add(t1);
    add(t2);
    add(t3);
  }
  class T1 implements TextListener {
    public void textValueChanged(TextEvent e) {
      t2.setText(t1.getText());
    }
  }
  class T1A implements ActionListener {
    private int count = 0;
    public void actionPerformed(ActionEvent e) {
      t3.setText("t1 Action Event " + count++);
    }
  }
  class T1K extends KeyAdapter {
    public void keyTyped(KeyEvent e) {
      String ts = t1.getText();
      if(e.getKeyChar() ==
          KeyEvent.VK_BACK_SPACE) {
        ts = ts.substring(0, ts.length() - 1);
        t1.setText(ts);
      }
```

```
      else
        t1.setText(
          t1.getText() +
            Character.toUpperCase(
              e.getKeyChar()));
      t1.setCaretPosition(
        t1.getText().length());
      // Stop regular character from appearing:
      e.consume();
    }
  }
  class B1 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      s = t1.getSelectedText();
      if(s.length() == 0) s = t1.getText();
      t1.setEditable(true);
    }
  }
  class B2 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t1.setText("Inserted by Button 2: " + s);
      t1.setEditable(false);
    }
  }
  static class WL extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
      System.exit(0);
    }
  }
  public static void main(String args[]) {
    TextNew applet = new TextNew();
    Frame aFrame = new Frame("TextNew");
    aFrame.addWindowListener(new WL());
    aFrame.add(applet, BorderLayout.CENTER);
    aFrame.setSize(300,200);
    applet.init();
    applet.start();
    aFrame.setVisible(true);
  }
} ///:~
```

The **TextField t3** is included as a place to report when the action listener for the **TextField t1** is fired. You'll see that the action listener for a **TextField** is only fired when you press the "enter" key.

The **TextField t1** has several listeners attached to it. The **T1** listener copies all text from **t1** into **t2**, and the **T1K** listener forces all characters to upper case. You'll notice that the two work together, and if you add the **T1K** listener *after* you add the **T1** listener, it doesn't matter: all characters will still be forced to upper case in both text fields. So it would seem that keyboard events are always fired before **TextComponent** events, and if you want the characters in **t2** to retain the original case that was typed in, you'll have to do some extra work.

T1K has some other activities of interest. You must detect a backspace (since you're controlling everything now) and perform the deletion. The caret must be explicitly set to the end of the field, otherwise it won't behave as you expect. And finally, to prevent the original character from being handled by the default mechanism, the event must be "consumed" using the **consume( )** method that exists for event objects. This tells the system to stop firing the rest of the event handlers for this particular event.

This example also quietly demonstrates one of the benefits of the design of inner classes. Notice that in the inner class:

```
  class T1 implements TextListener {
    public void textValueChanged(TextEvent e) {
      t2.setText(t1.getText());
    }
  }
```

t1 and **t2** are *not* members of **T1**, and yet they're accessible without any special qualification. This is because an object of an inner class automatically captures a handle to the outer object that created it, so you can treat members and methods of the enclosing class object as if they're yours. As you can see, this is quite convenient.[4]

check boxes & radio buttons

As noted previously, check boxes and radio buttons are both created with the same class, **Checkbox**, but radio buttons are **Checkbox**es placed into a **CheckboxGroup**. In either case, the interesting event is **ItemEvent**, for which you create an **ItemListener**.

When dealing with a group of check boxes or radio buttons, you have a choice: either create a new inner class to handle the event for each different **Checkbox**, or create one inner class that determines which **Checkbox** was clicked and register a single object of that inner class with each **Checkbox** object. The following example shows both approaches:

```
//: RadioCheckNew.java
// Radio buttons and Check Boxes in Java 1.1
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class RadioCheckNew extends Applet {
  TextField t = new TextField(30);
  Checkbox cb[] = {
    new Checkbox("Check Box 1"),
    new Checkbox("Check Box 2"),
    new Checkbox("Check Box 3") };
  CheckboxGroup g = new CheckboxGroup();
  Checkbox
    cb4 = new Checkbox("four", g, false),
    cb5 = new Checkbox("five", g, true),
    cb6 = new Checkbox("six", g, false);
  public void init() {
    t.setEditable(false);
    add(t);
    ILCheck il = new ILCheck();
    for(int i = 0; i < cb.length; i++) {
      cb[i].addItemListener(il);
      add(cb[i]);
    }
    cb4.addItemListener(new IL4());
    cb5.addItemListener(new IL5());
    cb6.addItemListener(new IL6());
    add(cb4); add(cb5); add(cb6);
  }
  // Checking the source:
  class ILCheck implements ItemListener {
    public void itemStateChanged(ItemEvent e) {
      for(int i = 0; i < cb.length; i++) {
```

[4] And it solves the problem of "callbacks" without adding any awkward "method pointer" feature to Java.

```
        if(e.getSource().equals(cb[i])) {
          t.setText("Check box " + (i + 1));
          return;
        }
      }
    }
  }
  // Vs. individual class for each item:
  class IL4 implements ItemListener {
    public void itemStateChanged(ItemEvent e) {
      t.setText("Radio button four");
    }
  }
  class IL5 implements ItemListener {
    public void itemStateChanged(ItemEvent e) {
      t.setText("Radio button five");
    }
  }
  class IL6 implements ItemListener {
    public void itemStateChanged(ItemEvent e) {
      t.setText("Radio button six");
    }
  }
  static class WL extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
      System.exit(0);
    }
  }
  public static void main(String args[]) {
    RadioCheckNew applet = new RadioCheckNew();
    Frame aFrame = new Frame("RadioCheckNew");
    aFrame.addWindowListener(new WL());
    aFrame.add(applet, BorderLayout.CENTER);
    aFrame.setSize(300,200);
    applet.init();
    applet.start();
    aFrame.setVisible(true);
  }
} ///:~
```

ILCheck has the advantage that it automatically adapts when you add or subtract **Checkbox**es. Of course, you can use this with radio buttons as well. It should only be used, however, when your logic is general enough to support this approach, otherwise you'll end up with a cascaded **if** statement, a sure sign you should revert to using independent listener classes.

drop-down lists

Drop-down lists (**Choice**) in Java 1.1 also use **ItemListener**s to notify you when a choice has changed:

```
//: ChoiceNew.java
// Drop-down lists with Java 1.1
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class ChoiceNew extends Applet {
  String description[] = { "Ebullient", "Obtuse",
    "Recalcitrant", "Brilliant", "Somnescent",
    "Timorous", "Florid", "Putrescent" };
  TextField t = new TextField(100);
  Choice c = new Choice();
```

```
  Button b = new Button("Add items");
  int count = 0;
  public void init() {
    t.setEditable(false);
    for(int i = 0; i < 4; i++)
      c.addItem(description[count++]);
    add(t);
    add(c);
    add(b);
    c.addItemListener(new CL());
    b.addActionListener(new BL());
  }
  class CL implements ItemListener {
    public void itemStateChanged(ItemEvent e) {
      t.setText("index: " +  c.getSelectedIndex()
        + "   " + e.toString());
    }
  }
  class BL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if(count < description.length)
        c.addItem(description[count++]);
    }
  }
  static class WL extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
      System.exit(0);
    }
  }
  public static void main(String args[]) {
    ChoiceNew applet = new ChoiceNew();
    Frame aFrame = new Frame("ChoiceNew");
    aFrame.addWindowListener(new WL());
    aFrame.add(applet, BorderLayout.CENTER);
    aFrame.setSize(750,100);
    applet.init();
    applet.start();
    aFrame.setVisible(true);
  }
} ///:~
```

Nothing else here is particularly new (except that Java 1.1 has significantly fewer bugs in the UI classes).

lists

You'll recall that one of the problems with the Java 1.0 **List** design is that it took extra work to make it do what you'd expect: react to a single click on one of the list elements. Java 1.1 has solved this problem:

```
//: ListNew.java
// Java 1.1 Lists are easier to use
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class ListNew extends Applet {
  String flavors[] = { "Chocolate", "Strawberry",
    "Vanilla Fudge Swirl", "Mint Chip",
    "Mocha Almond Fudge", "Rum Raisin",
    "Praline Cream", "Mud Pie" };
```

```
    // Show 6 items, allow multiple selection:
    List lst = new List(6, true);
    TextArea t = new TextArea(flavors.length, 30);
    Button b = new Button("test");
    int count = 0;
    public void init() {
      t.setEditable(false);
      for(int i = 0; i < 4; i++)
        lst.addItem(flavors[count++]);
      add(t);
      add(lst);
      add(b);
      lst.addItemListener(new LL());
      b.addActionListener(new BL());
    }
    class LL implements ItemListener {
      public void itemStateChanged(ItemEvent e) {
        t.setText("");
        String[] items = lst.getSelectedItems();
        for(int i = 0; i < items.length; i++)
          t.append(items[i] + "\n");
      }
    }
    class BL implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        if(count < flavors.length)
          lst.addItem(flavors[count++], 0);
      }
    }
    static class WL extends WindowAdapter {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    }
    public static void main(String args[]) {
      ListNew applet = new ListNew();
      Frame aFrame = new Frame("ListNew");
      aFrame.addWindowListener(new WL());
      aFrame.add(applet, BorderLayout.CENTER);
      aFrame.setSize(300,200);
      applet.init();
      applet.start();
      aFrame.setVisible(true);
    }
  } ///:~
```

You can see that no extra logic is required to support a single click on a list item. You just attach a listener like you do everywhere else.

menus

The event handling for menus does seem to benefit from the new Java 1.1 event model, but Java's approach to menus is still very messy and requires a lot of hand-coding. The right medium for a menu seems to be a resource rather than lots of code. Keep in mind that program-building tools will generally handle the creation of menus for you, so that will reduce the pain somewhat (as long as they will also handle the maintenance!).

In addition, you'll find the events for menus are inconsistent and can lead to confusion: **MenuItem**s use **ActionListener**s, but **CheckboxMenuItem**s use **ItemListener**s. The **Menu** objects themselves can also support **ActionListener**s, but that's not usually helpful. Generally you'll just attach listeners to each **MenuItem** or **CheckboxMenuItem**, but the following example (revised from the earlier version) also

shows ways to combine the capture of multiple menu components into a single listener class. As you'll see, it's probably not worth the hassle to do this.

```
//: MenuNew.java
// Menus in Java 1.1
import java.awt.*;
import java.awt.event.*;

public class MenuNew extends Frame {
  String flavors[] = { "Chocolate", "Strawberry",
    "Vanilla Fudge Swirl", "Mint Chip",
    "Mocha Almond Fudge", "Rum Raisin",
    "Praline Cream", "Mud Pie" };
  TextField t = new TextField("No flavor", 30);
  MenuBar mb1 = new MenuBar();
  Menu f = new Menu("File");
  Menu m = new Menu("Flavors");
  Menu s = new Menu("Safety");
  // Alternative approach:
  CheckboxMenuItem safety[] = {
    new CheckboxMenuItem("Guard"),
    new CheckboxMenuItem("Hide")
  };
  MenuItem file[] = {
    new MenuItem("Open"),
    new MenuItem("Exit")
  };
  // A second menu bar to swap to:
  MenuBar mb2 = new MenuBar();
  Menu fooBar = new Menu("fooBar");
  MenuItem other[] = {
    new MenuItem("Foo"),
    new MenuItem("Bar"),
    new MenuItem("Baz"),
  };
  // Initialization code:
  {
    ML ml = new ML();
    CMI cmi = new CMI();
    safety[0].setActionCommand("Guard");
    safety[0].addItemListener(cmi);
    safety[1].setActionCommand("Hide");
    safety[1].addItemListener(cmi);
    file[0].setActionCommand("Open");
    file[0].addActionListener(ml);
    file[1].setActionCommand("Exit");
    file[1].addActionListener(ml);
    other[0].addActionListener(new FooL());
    other[1].addActionListener(new BarL());
    other[2].addActionListener(new BazL());
  }
  Button b = new Button("Swap Menus");
  public MenuNew() {
    FL fl = new FL();
    for(int i = 0; i < flavors.length; i++) {
      MenuItem mi = new MenuItem(flavors[i]);
      mi.addActionListener(fl);
      m.add(mi);
      // Add separators at intervals:
      if((i+1) % 3 == 0)
```

```
        m.addSeparator();
      }
      for(int i = 0; i < safety.length; i++)
        s.add(safety[i]);
      f.add(s);
      for(int i = 0; i < file.length; i++)
        f.add(file[i]);
      mb1.add(f);
      mb1.add(m);
      setMenuBar(mb1);
      t.setEditable(false);
      add(t, BorderLayout.CENTER);
      // Set up the system for swapping menus:
      b.addActionListener(new BL());
      add(b, BorderLayout.NORTH);
      for(int i = 0; i < other.length; i++)
        fooBar.add(other[i]);
      mb2.add(fooBar);
    }
    class BL implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        MenuBar m = getMenuBar();
        if(m == mb1) setMenuBar(mb2);
        else if (m == mb2) setMenuBar(mb1);
      }
    }
    class ML implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        MenuItem target = (MenuItem)e.getSource();
        String actionCommand =
          target.getActionCommand();
        if(actionCommand.equals("Open")) {
          String s = t.getText();
          boolean chosen = false;
          for(int i = 0; i < flavors.length; i++)
            if(s.equals(flavors[i])) chosen = true;
          if(!chosen)
            t.setText("Choose a flavor first!");
          else
            t.setText("Opening "+ s +". Mmm, mm!");
        } else if(actionCommand.equals("Exit")) {
          dispatchEvent(
            new WindowEvent(MenuNew.this,
              WindowEvent.WINDOW_CLOSING));
        }
      }
    }
    class FL implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        MenuItem target = (MenuItem)e.getSource();
        t.setText(target.getLabel());
      }
    }
    // Alternatively, you can create a different
    // class for each different MenuItem. Then you
    // Don't have to figure out which one it is:
    class FooL implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        t.setText("Foo selected");
      }
```

```
  }
  class BarL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t.setText("Bar selected");
    }
  }
  class BazL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t.setText("Baz selected");
    }
  }
  class CMI implements ItemListener {
    public void itemStateChanged(ItemEvent e) {
      CheckboxMenuItem target =
        (CheckboxMenuItem)e.getSource();
      String actionCommand =
        target.getActionCommand();
      if(actionCommand.equals("Guard"))
        t.setText("Guard the Ice Cream! " +
          "Guarding is " + target.getState());
      else if(actionCommand.equals("Hide"))
        t.setText("Hide the Ice Cream! " +
          "Is it cold? " + target.getState());
    }
  }
  static class WL extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
      System.exit(0);
    }
  }
  public static void main(String args[]) {
    MenuNew f = new MenuNew();
    f.setSize(300,200);
    f.setVisible(true);
    f.addWindowListener(new WL());
  }
} ///:~
```

This code is essentially the same as the previous (Java 1.0) version, until you get to the initialization section (marked by the opening brace right after the comment "Initialization code:"). Here you can see the **ItemListener**s and **ActionListener**s attached to the various menu components.

You can also see the use of **setActionCommand( )**. This seems a bit strange because in each case the "action command" is exactly the same as the label on the menu component. Why not just use the label, instead of this alternative string? The problem is internationalization. If you retarget this program to another language, you only want to change the label in the menu, and not go through the code changing all the logic which will no doubt introduce new errors. So to make this easy for code that checks the text string associated with a menu component, the "action command" can be immutable while the menu label can change. All the code works with the "action command," so it's unaffected by changes to the menu labels. Notice that in this program, not all the menu components are examined for their action commands, so those that aren't don't have their action command set.

Much of the constructor is the same as before, with the exception of a couple of calls to add listeners. The bulk of the work happens in the listeners themselves. In **BL**, the **MenuBar** swapping happens as in the previous example. In **ML**, the "figure out who rang" approach is taken by getting the source of the **ActionEvent** and casting it to a **MenuItem**, then getting the action command string to pass it through a cascaded **if** statement. Much of this is the same as before, but notice that if "Exit" is chosen, a new **WindowEvent** is created, passing in the handle of the enclosing class object (**MenuNew.this**) and creating a **WINDOW_CLOSING** event. This is handed to the **dispatchEvent( )** method of the enclosing class object, which then ends up calling **windowClosing( )** inside **WL**, just as if the message had been

generated the "normal" way. Through this mechanism, you can dispatch any message you want in any circumstances, so it's quite powerful.

The **FL** listener is simple even though it's handling all the different flavors in the flavor menu. This approach is useful if you have enough simplicity in your logic, but in general you'll usually want to take the approach used with **FooL**, **BarL** and **BazL**, where they are each only attached to a single menu component, so no extra detection logic is necessary and you know exactly who called the listener. Even with the profusion of classes generated this way, the code inside tends to be smaller and the process is more foolproof.

dialog boxes

This is a direct rewrite of the earlier **ToeTest.java.** In this version, however, everything is placed inside an inner class. Although this completely eliminates the need to keep track of the object that spawned any class, as was the case in **ToeTest.java**, it may be taking the concept of inner classes a bit too far. At one point, the inner classes are nested four deep! This is the kind of design where you need to decide whether the benefit of inner classes is worth the increased complexity. In addition, when you create an inner class you're tying that class to its surrounding class. A standalone class can more easily be reused.

```
//: ToeTestNew.java
// Demonstration of dialog boxes
// and creating your own components
import java.awt.*;
import java.awt.event.*;

public class ToeTestNew extends Frame {
  TextField rows = new TextField("3");
  TextField cols = new TextField("3");
  public ToeTestNew() {
    setTitle("Toe Test");
    Panel p = new Panel();
    p.setLayout(new GridLayout(2,2));
    p.add(new Label("Rows", Label.CENTER));
    p.add(rows);
    p.add(new Label("Columns", Label.CENTER));
    p.add(cols);
    add(p, BorderLayout.NORTH);
    Button b = new Button("go");
    b.addActionListener(new BL());
    add(b, BorderLayout.SOUTH);
  }
  static final int XX = 1;
  static final int OO = 2;
  class ToeDialog extends Dialog {
    // w = number of cells wide
    // h = number of cells high
    int turn = XX; // Start with x's turn
    public ToeDialog(int w, int h) {
      super(ToeTestNew.this,
        "The game itself", false);
      setLayout(new GridLayout(w, h));
      for(int i = 0; i < w * h; i++)
        add(new ToeButton());
      setSize(w * 50, h * 50);
      addWindowListener(new WLD());
    }
    class ToeButton extends Canvas {
      int state = 0;
      ToeButton() {
        addMouseListener(new ML());
```

```
      }
      public void paint(Graphics  g) {
        int x1 = 0;
        int y1 = 0;
        int x2 = getSize().width - 1;
        int y2 = getSize().height - 1;
        g.drawRect(x1, y1, x2, y2);
        x1 = x2/4;
        y1 = y2/4;
        int wide = x2/2;
        int high = y2/2;
        if(state == 1) {
          g.drawLine(x1, y1,
            x1 + wide, y1 + high);
          g.drawLine(x1, y1 + high,
            x1 + wide, y1);
        }
        if(state == 2) {
          g.drawOval(x1, y1,
            x1 + wide/2, y1 + high/2);
        }
      }
      class ML extends MouseAdapter {
        public void mousePressed(MouseEvent e) {
          if(state == 0) {
            state = turn;
            turn = (turn == 1 ? 2 : 1);
          }
          else
            state = (state == 1 ? 2 : 1);
          repaint();
        }
      }
    }
    class WLD extends WindowAdapter {
      public void windowClosing(WindowEvent e) {
        dispose();
      }
    }
  }
  class BL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      Dialog d = new ToeDialog(
        Integer.parseInt(rows.getText()),
        Integer.parseInt(cols.getText()));
      d.show();
    }
  }
  static class WL extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
      System.exit(0);
    }
  }
  public static void main(String args[]) {
    Frame f = new ToeTestNew();
    f.setSize(200,100);
    f.setVisible(true);
    f.addWindowListener(new WL());
  }
} ///:~
```

There are some restrictions when using inner classes here. In particular, **static**s can only be at the outer level of the class, so inner classes cannot have **static** data or inner classes.

file dialogs

Converting from **FileDialogTest.java** to the new event model is very straightforward:

```
//: FileDialogNew.java
// Demonstration of File dialog boxes
import java.awt.*;
import java.awt.event.*;

public class FileDialogNew extends Frame {
  TextField filename = new TextField();
  TextField directory = new TextField();
  Button open = new Button("Open");
  Button save = new Button("Save");
  public FileDialogNew() {
    setTitle("File Dialog Test");
    Panel p = new Panel();
    p.setLayout(new FlowLayout());
    open.addActionListener(new OpenL());
    p.add(open);
    save.addActionListener(new SaveL());
    p.add(save);
    add(p, BorderLayout.SOUTH);
    directory.setEditable(false);
    filename.setEditable(false);
    p = new Panel();
    p.setLayout(new GridLayout(2,1));
    p.add(filename);
    p.add(directory);
    add(p, BorderLayout.NORTH);
  }
  class OpenL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      // Two arguments, defaults to open file:
      FileDialog d = new FileDialog(
        FileDialogNew.this,
        "What file do you want to open?");
      d.setFile("*.java");
      d.setDirectory("."); // Current directory
      d.show();
      if(d != null) {
        filename.setText(d.getFile());
        directory.setText(d.getDirectory());
      }
    }
  }
  class SaveL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      FileDialog d = new FileDialog(
        FileDialogNew.this,
        "What file do you want to save?",
        FileDialog.SAVE);
      d.setFile("*.java");
      d.setDirectory(".");
      d.show();
      if(d != null) {
        filename.setText(d.getFile());
        directory.setText(d.getDirectory());
```

```
      }
    }
  }
  static class WL extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
      System.exit(0);
    }
  }
  public static void main(String args[]) {
    Frame f = new FileDialogNew();
    f.setSize(250,110);
    f.setVisible(true);
    f.addWindowListener(new WL());
  }
} ///:~
```

It would be nice if all the conversions were this easy, but in general they're easy enough, and your code benefits from the improved readability.

binding events dynamically

One of the benefits of the new AWT event model is flexibility. In the old model you were forced to hard-code the behavior of your program, but with the new model you can add and remove event behavior with single method calls. The following example demonstrates this:

```
//: DynamicEvents.java
// The new Java 1.1 event model allows you to
// change event behavior dynamically. Also
// demonstrates multiple actions for an event.
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class DynamicEvents extends Frame {
  Vector v = new Vector();
  int i = 0;
  Button
    b1 = new Button("Button 1"),
    b2 = new Button("Button 2");
  public DynamicEvents() {
    addWindowListener(new BWL());
    setLayout(new FlowLayout());
    b1.addActionListener(new B());
    b1.addActionListener(new B1());
    b2.addActionListener(new B());
    b2.addActionListener(new B2());
    add(b1);
    add(b2);
  }
  class B implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      System.out.println("A button was pressed");
    }
  }
  class CountListener implements ActionListener {
    int index;
    public CountListener(int i) { index = i; }
    public void actionPerformed(ActionEvent e) {
      System.out.println(
        "Counted Listener " + index);
```

```
    }
  }
  class B1 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      System.out.println("Button 1 pressed");
      ActionListener a = new CountListener(i++);
      v.addElement(a);
      b2.addActionListener(a);
    }
  }
  class B2 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      System.out.println("Button 2 pressed");
      int end = v.size() -1;
      if(end >= 0) {
        b2.removeActionListener(
          (ActionListener)v.elementAt(end));
        v.removeElementAt(end);
      }
    }
  }
  class BWL extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
      System.out.println("Window Closing");
      System.exit(0);
    }
  }
  public static void main(String args[]) {
    Frame f = new DynamicEvents();
    f.setSize(300,200);
    f.show();
  }
} ///:~
```

The new twists in this example are:

1. There is more than one listener attached to each **Button**. Usually, components handle events as *multicast*, meaning you can register many listeners for a single event. In the special components where an event is handled as *unicast*, you'll get a **TooManyListenersException**.

2. During the execution of the program, listeners are dynamically added and removed from the **Button b2**. Adding is accomplished in the way you've seen before, but each component also has a **removeXXXListener( )** method to remove each type of listener.

This kind of flexibility provides much greater power in your programming.

separating business logic from UI logic

In general you'll want to design your classes so that each one does "only one thing." This is particularly important where user-interface code is concerned, since it's very easy to wrap up "what you're doing" with "how you're displaying it." This kind of coupling prevents code reuse. It's much more desirable to separate your "business logic" from the GUI. This way, you can not only reuse the business logic more easily, it's also easier to reuse the GUI.

Another issue is *multi-tiered* systems, where the "business objects" reside on a completely separate machine. This central location of the business rules allows changes to be instantly effective for all new transactions, and is thus a very compelling way to set up a system. However, these business objects may be used in many different applications and thus should not be tied to any particular mode of display. They should just perform the business operations, nothing more.

The following example shows how easy it is to separate the business logic from the GUI code:

```
//: Separation.java
// Separating GUI logic and business objects
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

class BusinessLogic {
  private int modifier;
  BusinessLogic(int mod) {
    modifier = mod;
  }
  public void setModifier(int mod) {
    modifier = mod;
  }
  public int getModifier() {
    return modifier;
  }
  // Some business operations:
  public int calculation1(int arg) {
    return arg * modifier;
  }
  public int calculation2(int arg) {
    return arg + modifier;
  }
}

public class Separation extends Applet {
  TextField
    t = new TextField(20),
    mod = new TextField(20);
  BusinessLogic bl = new BusinessLogic(2);
  Button
    calc1 = new Button("Calculation 1"),
    calc2 = new Button("Calculation 2");
  public void init() {
    add(t);
    calc1.addActionListener(new Calc1L());
    calc2.addActionListener(new Calc2L());
    add(calc1); add(calc2);
    mod.addTextListener(new ModL());
    add(new Label("Modifier:"));
    add(mod);
  }
  static int getValue(TextField tf) {
    try {
      return Integer.parseInt(tf.getText());
    } catch(NumberFormatException e) {
      return 0;
    }
  }
  class Calc1L implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t.setText(Integer.toString(
        bl.calculation1(getValue(t))));
    }
  }
  class Calc2L implements ActionListener {
    public void actionPerformed(ActionEvent e) {
```

```
        t.setText(Integer.toString(
          bl.calculation2(getValue(t))));
      }
    }
    class ModL implements TextListener {
      public void textValueChanged(TextEvent e) {
        bl.setModifier(getValue(mod));
      }
    }
    static class WL extends WindowAdapter {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    }
    public static void main(String args[]) {
      Separation applet = new Separation();
      Frame aFrame = new Frame("Separation");
      aFrame.addWindowListener(new WL());
      aFrame.add(applet, BorderLayout.CENTER);
      aFrame.setSize(200,200);
      applet.init();
      applet.start();
      aFrame.setVisible(true);
    }
  } ///:~
```

You can see that **BusinessLogic** is a very straightforward class that performs its operations without even a hint that it may be used in a GUI environment. It just does its job.

Separation keeps track of all the UI details, and it only talks to **BusinessLogic** through its **public** interface. All the operations are centered around getting information back and forth through the UI and the **BusinessLogic** object. So **Separation**, in turn, just does its job. Since it's only talking to a **BusinessLogic** object and doesn't know much else about it, this could be massaged into talking to other types of objects without much trouble.

Thinking in terms of separating UI from business logic also makes life easier when you're adapting legacy code to work with Java.

recommended coding approaches

Inner classes, the new event model, and the fact that the old event model is still supported along with *new* support for old-style programming has added a new element of confusion. Now there are even more different ways for people to write unpleasant code. Unfortunately, this kind of code is showing up in books and article examples, and even in documentation and examples distributed from Javasoft! In this section we'll look at some misunderstandings about what you should and shouldn't do with the new AWT, and end by showing that except in very extenuating circumstances you can always use listener classes (written as inner classes) to solve your event-handling needs. Since this is also the simplest and clearest approach, it should be a relief for you to learn this.

Before looking at anything else, you should know that although Java 1.1 is backward-compatible with Java 1.0 (that is, you can compile and run 1.0 programs with 1.1), you cannot mix the event models within the same program. That is, you cannot use the old-style **action( )** method in the same program where you employ listeners. This can be a problem in a larger program when you're trying to integrate old code with a new program, since you must decide whether to use the old, hard-to-maintain approach with the new program or to update the old code. This shouldn't be too much of a battle since the new approach is so superior to the old.

baseline: the good way to do it

So you have something to compare with, here's an example showing the recommended approach. By now it should be reasonably familiar and comfortable:

```
//: GoodIdea.java
// The best way to design classes using the new
// Java 1.1 event model: use an inner class for
// each different event. This maximizes
// flexibility and modularity.
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class GoodIdea extends Frame {
  Button
    b1 = new Button("Button 1"),
    b2 = new Button("Button 2");
  public GoodIdea() {
    setLayout(new FlowLayout());
    addWindowListener(new WL());
    b1.addActionListener(new B1());
    b2.addActionListener(new B2());
    add(b1);
    add(b2);
  }
  public class B1 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      System.out.println("Button 1 pressed");
    }
  }
  public class B2 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      System.out.println("Button 2 pressed");
    }
  }
  class WL extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
      System.out.println("Window Closing");
      System.exit(0);
    }
  }
  public static void main(String args[]) {
    Frame f = new GoodIdea();
    f.setSize(300,200);
    f.setVisible(true);
  }
} ///:~
```

This is fairly trivial: each button has its own listener which prints something out to the console. But notice there isn't an **if** statement in the entire program, or any statement which says "I wonder what caused this event." Each piece of code is concerned with *doing*, not type-checking. This is the best way to write your code – not only is it easier to conceptualize, but much easier to read and maintain. Cutting-and-pasting to create new programs is also much easier.

implementing the main class as a listener

The first bad idea is a very common and recommended approach. This makes the main class (typically **Applet** or **Frame**, but it could be any class) implement the various listeners. Here's an example:

```
//: BadIdea1.java
// Some literature recommends this approach,
// but it's missing the point of the new event
// model in Java 1.1
import java.awt.*;
import java.awt.event.*;
```

```
import java.util.*;

public class BadIdea1 extends Frame
    implements ActionListener, WindowListener {
  Button
    b1 = new Button("Button 1"),
    b2 = new Button("Button 2");
  public BadIdea1() {
    setLayout(new FlowLayout());
    addWindowListener(this);
    b1.addActionListener(this);
    b2.addActionListener(this);
    add(b1);
    add(b2);
  }
  public void actionPerformed(ActionEvent e) {
    Object source = e.getSource();
    if(source == b1)
      System.out.println("Button 1 pressed");
    else if(source == b2)
      System.out.println("Button 2 pressed");
    else
      System.out.println("Something else");
  }
  public void windowClosing(WindowEvent e) {
    System.out.println("Window Closing");
    System.exit(0);
  }
  public void windowClosed(WindowEvent e) {}
  public void windowDeiconified(WindowEvent e) {}
  public void windowIconified(WindowEvent e) {}
  public void windowActivated(WindowEvent e) {}
  public void windowDeactivated(WindowEvent e) {}
  public void windowOpened(WindowEvent e) {}

  public static void main(String args[]) {
    Frame f = new BadIdea1();
    f.setSize(300,200);
    f.setVisible(true);
  }
} ///:~
```

The use of this shows up in the three lines:

```
    addWindowListener(this);
    b1.addActionListener(this);
    b2.addActionListener(this);
```

Since **BadIdea1** implements **ActionListener** and **WindowListener** these lines are certainly acceptable, and if you're still stuck in the mode of trying to make fewer classes to reduce server hits during applet loading, it seems to be a good idea. However:

1. Java 1.1 supports JAR files so all your files can be placed in a single compressed JAR archive which only requires one server hit. You no longer need to reduce class count for Internet efficiency.

2. The above code is much less modular so it's harder to grab and paste. Notice that you must not only implement the various interfaces for your main class, but in **actionPerformed( )** you've got to detect which action was performed using a cascaded **if** statement. Not only is this going backwards, *away* from the listener model, but you can't easily reuse the **actionPerformed( )** method since it's very specific to this particular application. Contrast

this with **GoodIdea.java**, where you can just grab one listener class and paste it in anywhere else with minimal fuss. Plus you can register multiple listener classes with a single event, allowing even more modularity in what each listener class does.

mixing the approaches

The second bad idea is to mix the two approaches: use inner listener classes, but also implement one or more listener interfaces as part of the main class. This approach has appeared without explanation in books and documentation, and I can only assume that the authors thought they must use the different approaches for different purposes. But you don't – in your programming you can probably use inner listener classes exclusively.

```
//: BadIdea2.java
// An improvement over BadIdea1.java, since it
// uses the WindowAdapter as an inner class
// instead of implementing all the methods of
// WindowListener, but still misses the
// valuable modularity of inner classes
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class BadIdea2 extends Frame
    implements ActionListener {
  Button
    b1 = new Button("Button 1"),
    b2 = new Button("Button 2");
  public BadIdea2() {
    setLayout(new FlowLayout());
    addWindowListener(new WL());
    b1.addActionListener(this);
    b2.addActionListener(this);
    add(b1);
    add(b2);
  }
  public void actionPerformed(ActionEvent e) {
    Object source = e.getSource();
    if(source == b1)
      System.out.println("Button 1 pressed");
    else if(source == b2)
      System.out.println("Button 2 pressed");
    else
      System.out.println("Something else");
  }
  class WL extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
      System.out.println("Window Closing");
      System.exit(0);
    }
  }

  public static void main(String args[]) {
    Frame f = new BadIdea2();
    f.setSize(300,200);
    f.setVisible(true);
  }
} ///:~
```

Since **actionPerformed( )** is still tightly coupled to the main class, it's hard to reuse that code. It's also messier and less pleasant to read than the inner class approach.

There's no reason you have to use any of the old thinking for events in Java 1.1 – so why do it?

inheriting a component

Another place where you'll often see variations on the old way of doing things is when creating a new type of component. Here's a simple example showing that here, too, the new way works:

```
//: GoodTechnique.java
// Your first choice when overriding components
// should be to install listeners. The code is
// much safer, more modular and maintainable.
import java.awt.*;
import java.awt.event.*;

class Display {
  public static final int
    event = 0, component = 1,
    mouse = 2, mouseMove = 3,
    focus = 4, key = 5, action = 6,
    last = 7;
  public String evnt[];
  Display() {
    evnt = new String[last];
    for(int i = 0; i < last; i++)
      evnt[i] = new String();
  }
  public void show(Graphics g) {
    for(int i = 0; i < last; i++)
      g.drawString(evnt[i], 0, 10 * i + 10);
  }
}

class EnabledPanel extends Panel {
  Color c;
  int id;
  Display display = new Display();
  public EnabledPanel(int i, Color mc) {
    id = i;
    c = mc;
    setLayout(new BorderLayout());
    add(new MyButton(), BorderLayout.SOUTH);
    addComponentListener(new CL());
    addFocusListener(new FL());
    addKeyListener(new KL());
    addMouseListener(new ML());
    addMouseMotionListener(new MML());
  }
  // To eliminate flicker:
  public void update(Graphics g) {
    paint(g);
  }
  public void paint(Graphics  g) {
    g.setColor(c);
    Dimension s = getSize();
    g.fillRect(0, 0, s.width, s.height);
    g.setColor(Color.black);
    display.show(g);
  }
  // Don't need to enable anything for this:
  public void
      processEvent(AWTEvent e) {
```

```
    display.evnt[Display.event]= e.toString();
    repaint();
    super.processEvent(e);
  }
  class CL implements ComponentListener {
    public void componentMoved(ComponentEvent e) {
      display.evnt[Display.component] =
        "component moved";
      repaint();
    }
    public void componentResized(ComponentEvent e) {
      display.evnt[Display.component] =
        "component resized";
      repaint();
    }
    public void componentHidden(ComponentEvent e) {
      display.evnt[Display.component] =
        "component hidden";
      repaint();
    }
    public void componentShown(ComponentEvent e) {
      display.evnt[Display.component] =
        "component shown";
      repaint();
    }
  }
  class FL implements FocusListener {
    public void focusGained(FocusEvent e) {
      display.evnt[Display.focus] =
        "focus gained";
      repaint();
    }
    public void focusLost(FocusEvent e) {
      display.evnt[Display.focus] =
        "focus lost";
      repaint();
    }
  }
  class KL implements KeyListener {
    public void keyPressed(KeyEvent e) {
      display.evnt[Display.key] =
        "key pressed: ";
      showCode(e);
    }
    public void keyReleased(KeyEvent e) {
      display.evnt[Display.key] =
        "key released: ";
      showCode(e);
    }
    public void keyTyped(KeyEvent e) {
      display.evnt[Display.key] =
        "key typed: ";
      showCode(e);
    }
    void showCode(KeyEvent e) {
      int code = e.getKeyCode();
      display.evnt[Display.key] +=
        KeyEvent.getKeyText(code);
      repaint();
    }
```

```
  }
  class ML implements MouseListener {
    public void mouseClicked(MouseEvent e) {
      requestFocus(); // Get focus on click
      display.evnt[Display.mouse] =
        "mouse clicked";
      showMouse(e);
    }
    public void mousePressed(MouseEvent e) {
      display.evnt[Display.mouse] =
        "mouse pressed";
      showMouse(e);
    }
    public void mouseReleased(MouseEvent e) {
      display.evnt[Display.mouse] =
        "mouse released";
      showMouse(e);
    }
    public void mouseEntered(MouseEvent e) {
      display.evnt[Display.mouse] =
        "mouse entered";
      showMouse(e);
    }
    public void mouseExited(MouseEvent e) {
      display.evnt[Display.mouse] =
        "mouse exited";
      showMouse(e);
    }
    void showMouse(MouseEvent e) {
      display.evnt[Display.mouse] +=
        ", x = " + e.getX() +
        ", y = " + e.getY();
      repaint();
    }
  }
  class MML implements MouseMotionListener {
    public void mouseDragged(MouseEvent e) {
      display.evnt[Display.mouseMove] =
        "mouse dragged";
      showMouse(e);
    }
    public void mouseMoved(MouseEvent e) {
      display.evnt[Display.mouseMove] =
        "mouse moved";
      showMouse(e);
    }
    void showMouse(MouseEvent e) {
      display.evnt[Display.mouseMove] +=
        ", x = " + e.getX() +
        ", y = " + e.getY();
      repaint();
    }
  }
}

class MyButton extends Button {
  int clickCounter;
  String label = "";
  public MyButton() {
    addActionListener(new AL());
```

```
  }
  public void paint(Graphics g) {
    g.setColor(Color.green);
    Dimension s = getSize();
    g.fillRect(0, 0, s.width, s.height);
    g.setColor(Color.black);
    g.drawRect(0, 0, s.width - 1, s.height - 1);
    drawLabel(g);
  }
  private void drawLabel(Graphics g) {
    FontMetrics fm = g.getFontMetrics();
    int width = fm.stringWidth(label);
    int height = fm.getHeight();
    int ascent = fm.getAscent();
    int leading = fm.getLeading();
    int horizMargin =
      (getSize().width - width)/2;
    int verMargin =
      (getSize().height - height)/2;
    g.setColor(Color.red);
    g.drawString(label, horizMargin,
                 verMargin + ascent + leading);
  }
  class AL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      clickCounter++;
      label = "click #" + clickCounter +
        " " + e.toString();
      repaint();
    }
  }
}

public class GoodTechnique extends Frame {
  GoodTechnique() {
    setLayout(new GridLayout(2,2));
    add(new EnabledPanel(1, Color.cyan));
    add(new EnabledPanel(2, Color.lightGray));
    add(new EnabledPanel(3, Color.yellow));
    addWindowListener(new WL());
  }
  class WL extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
      System.out.println(e);
      System.out.println("Window Closing");
      System.exit(0);
    }
  }
  public static void main(String args[]) {
    Frame f = new GoodTechnique();
    f.setTitle("Good Technique");
    f.setSize(700,700);
    f.setVisible(true);
  }
} ///:~
```

This example also demonstrates the various events that occur, and displays the information about them. The class **Display** is a way to centralize that information display. There's an array of **String**s to hold information about each type of event, and the method **show( )** takes a handle to whatever

Graphics object you have and writes directly on that surface. The scheme is intended to be somewhat reusable.

EnabledPanel represents the new type of component. It's a colored panel with a button at the bottom, and it captures all the events that happen over it. The **MyButton** object also captures the events that happen over it. Both components capture all events by using inner listener classes for every single event *except* where **EnabledPanel** overrides **processEvent( )** in the old style (notice it must also call **super.processEvent( )**). The only reason for using this method is that it captures every event that happens, so you can view everything that goes on. **processEvent( )** does nothing more than show the string representation of each event, otherwise it would have to use a cascade of **if** statements to figure out what event it was. On the other hand, the inner listener classes already know precisely what event occurred (assuming you only register them to components where you don't need any control logic, which should be your goal). Thus they don't have to check anything out; they just do their stuff.

Each listener modifies the **Display** string associated with its particular event and calls **repaint( )** so the strings get displayed. You can also see a trick that will usually eliminate flicker:

```
public void update(Graphics g) {
  paint(g);
}
```

You don't always need to override **update( )**, but if you write something that flickers, try it.

You can see that there are lots of listeners – however, type checking occurs for the listeners, and you can't listen for something that the component doesn't support (unlike **BadTechnique.java**, which you will see momentarily).

Experimenting with this program is quite educational since you learn a lot about the way events are occurring in Java.

ugly component inheritance

The alternative, which you will see put forward in many published works, is to call **enableEvents( )** and pass it the masks corresponding to the events you want to handle. This causes those events to be sent to the old-style methods (although they're new to Java 1.1) with names like **processFocusEvent( )**. You must also remember to call the base-class version. Here's what it looks like:

```
//: BadTechnique.java
// It's possible to override components this way,
// but the listener approach is much better, so
// why would you?
import java.awt.*;
import java.awt.event.*;

class Display {
  public static final int
    event = 0, component = 1,
    mouse = 2, mouseMove = 3,
    focus = 4, key = 5, action = 6,
    last = 7;
  public String evnt[];
  Display() {
    evnt = new String[last];
    for(int i = 0; i < last; i++)
      evnt[i] = new String();
  }
  public void show(Graphics g) {
    for(int i = 0; i < last; i++)
      g.drawString(evnt[i], 0, 10 * i + 10);
  }
```

```
}

class EnabledPanel extends Panel {
  Color c;
  int id;
  Display display = new Display();
  public EnabledPanel(int i, Color mc) {
    id = i;
    c = mc;
    setLayout(new BorderLayout());
    add(new MyButton(), BorderLayout.SOUTH);
    // Type checking is lost. You can enable and
    // process events that the component doesn't
    // capture:
    enableEvents(
      // Panel doesn't handle these:
      AWTEvent.ACTION_EVENT_MASK |
      AWTEvent.COMPONENT_EVENT_MASK |
      AWTEvent.ADJUSTMENT_EVENT_MASK |
      AWTEvent.ITEM_EVENT_MASK |
      AWTEvent.TEXT_EVENT_MASK |
      AWTEvent.WINDOW_EVENT_MASK |
      // Panel can handle these:
      AWTEvent.FOCUS_EVENT_MASK |
      AWTEvent.KEY_EVENT_MASK |
      AWTEvent.MOUSE_EVENT_MASK |
      AWTEvent.MOUSE_MOTION_EVENT_MASK |
      AWTEvent.CONTAINER_EVENT_MASK);
      // You can enable an event without
      // overriding its process method.
  }
  // To eliminate flicker:
  public void update(Graphics g) {
    paint(g);
  }
  public void paint(Graphics  g) {
    g.setColor(c);
    Dimension s = getSize();
    g.fillRect(0, 0, s.width, s.height);
    g.setColor(Color.black);
    display.show(g);
  }
  public void
      processEvent(AWTEvent e) {
    display.evnt[Display.event]= e.toString();
    repaint();
    super.processEvent(e);
  }
  public void
      processComponentEvent(ComponentEvent e) {
    switch(e.getID()) {
      case ComponentEvent.COMPONENT_MOVED:
        display.evnt[Display.component] =
          "component moved";
        break;
      case ComponentEvent.COMPONENT_RESIZED:
        display.evnt[Display.component] =
          "component resized";
        break;
      case ComponentEvent.COMPONENT_HIDDEN:
```

```
        display.evnt[Display.component] =
          "component hidden";
        break;
      case ComponentEvent.COMPONENT_SHOWN:
        display.evnt[Display.component] =
          "component shown";
        break;
      default:
    }
    repaint();
    // Must always remember to call the "super"
    // version of whatever you override:
    super.processComponentEvent(e);
  }
  public void
      processFocusEvent(FocusEvent e) {
    switch(e.getID()) {
      case FocusEvent.FOCUS_GAINED:
        display.evnt[Display.focus] =
          "focus gained";
        break;
      case FocusEvent.FOCUS_LOST:
        display.evnt[Display.focus] =
          "focus lost";
        break;
      default:
    }
    repaint();
    super.processFocusEvent(e);
  }
  public void
      processKeyEvent(KeyEvent e) {
    switch(e.getID()) {
      case KeyEvent.KEY_PRESSED:
        display.evnt[Display.key] =
          "key pressed: ";
        break;
      case KeyEvent.KEY_RELEASED:
        display.evnt[Display.key] =
          "key released: ";
        break;
      case KeyEvent.KEY_TYPED:
        display.evnt[Display.key] =
          "key typed: ";
        break;
      default:
    }
    int code = e.getKeyCode();
    display.evnt[Display.key] +=
      KeyEvent.getKeyText(code);
    repaint();
    super.processKeyEvent(e);
  }
  public void
      processMouseEvent(MouseEvent e) {
    switch(e.getID()) {
      case MouseEvent.MOUSE_CLICKED:
        requestFocus(); // Get focus on click
        display.evnt[Display.mouse] =
          "mouse clicked";
```

```
        break;
      case MouseEvent.MOUSE_PRESSED:
        display.evnt[Display.mouse] =
          "mouse pressed";
        break;
      case MouseEvent.MOUSE_RELEASED:
        display.evnt[Display.mouse] =
          "mouse released";
        break;
      case MouseEvent.MOUSE_ENTERED:
        display.evnt[Display.mouse] =
          "mouse entered";
        break;
      case MouseEvent.MOUSE_EXITED:
        display.evnt[Display.mouse] =
          "mouse exited";
        break;
      default:
    }
    display.evnt[Display.mouse] +=
      ", x = " + e.getX() +
      ", y = " + e.getY();
    repaint();
    super.processMouseEvent(e);
  }
  public void
      processMouseMotionEvent(MouseEvent e) {
    switch(e.getID()) {
      case MouseEvent.MOUSE_DRAGGED:
        display.evnt[Display.mouseMove] =
          "mouse dragged";
        break;
      case MouseEvent.MOUSE_MOVED:
        display.evnt[Display.mouseMove] =
          "mouse moved";
        break;
      default:
    }
    display.evnt[Display.mouseMove] +=
      ", x = " + e.getX() +
      ", y = " + e.getY();
    repaint();
    super.processMouseMotionEvent(e);
  }
}

class MyButton extends Button {
  int clickCounter;
  String label = "";
  public MyButton() {
    enableEvents(AWTEvent.ACTION_EVENT_MASK);
  }
  public void paint(Graphics g) {
    g.setColor(Color.green);
    Dimension s = getSize();
    g.fillRect(0, 0, s.width, s.height);
    g.setColor(Color.black);
    g.drawRect(0, 0, s.width - 1, s.height - 1);
    drawLabel(g);
  }
```

```
  private void drawLabel(Graphics g) {
    FontMetrics fm = g.getFontMetrics();
    int width = fm.stringWidth(label);
    int height = fm.getHeight();
    int ascent = fm.getAscent();
    int leading = fm.getLeading();
    int horizMargin =
      (getSize().width - width)/2;
    int verMargin =
      (getSize().height - height)/2;
    g.setColor(Color.red);
    g.drawString(label, horizMargin,
                 verMargin + ascent + leading);
  }
  public void
      processActionEvent(ActionEvent e) {
    clickCounter++;
    label = "click #" + clickCounter +
      " " + e.toString();
    repaint();
    super.processActionEvent(e);
  }
}

public class BadTechnique extends Frame {
  BadTechnique() {
    setLayout(new GridLayout(2,2));
    add(new EnabledPanel(1, Color.cyan));
    add(new EnabledPanel(2, Color.lightGray));
    add(new EnabledPanel(3, Color.yellow));
    // You can also do it for Windows:
    enableEvents(AWTEvent.WINDOW_EVENT_MASK);
  }
  public void processWindowEvent(WindowEvent e) {
    System.out.println(e);
    if(e.getID() == WindowEvent.WINDOW_CLOSING) {
      System.out.println("Window Closing");
      System.exit(0);
    }
  }
  public static void main(String args[]) {
    Frame f = new BadTechnique();
    f.setTitle("Bad Technique");
    f.setSize(700,700);
    f.setVisible(true);
  }
} ///:~
```

Sure, it works. But it's ugly and hard to write, read, debug, maintain and reuse. So why bother when you can use inner listener classes?

new Java 1.1 UI APIs

Java 1.1 has also added some important new functionality, including focus traversal, desktop color access, printing "inside the sandbox" and the beginnings of clipboard support.

Focus traversal is quite easy, since it's transparently present in the AWT library components and you don't have to do anything to make it work. If you make your own components and want them to

handle focus traversal, you override **isFocusTraversable( )** to return **true**. If you want to capture the keyboard focus on a mouse click, you catch the mouse down event and call **requestFocus( )**.

desktop colors

The desktop colors provide a way for you to know what the various color choices are on the current user's desktop. This way, you can use those colors in your program, if you desire. The colors are automatically initialized and placed in **static** members of class **SystemColor**, so all you need to do is read the member you're interested in. The names are intentionally self-explanatory: **desktop**, **activeCaption**, **activeCaptionText**, **activeCaptionBorder**, **inactiveCaption**, **inactiveCaptionText**, **inactiveCaptionBorder**, **window**, **windowBorder**, **windowText**, **menu**, **menuText**, **text**, **textText**, **textHighlight**, **textHighlightText**, **textInactiveText**, **control**, **controlText**, **controlHighlight**, **controlLtHighlight**, **controlShadow**, **controlDkShadow**, **scrollbar**, **info** (for help), **infoText** (for help text).

printing

There's some confusion involved with Java 1.1 printing support. Some of the publicity seemed to claim that you'd be able to print from within an applet. However, to print anything you must get a **PrintJob** object through a **Toolkit** object's **getPrintJob( )** method, which only takes a **Frame** object, and not an **Applet**. Thus it is only possible to print from within an application, not an applet.

Unfortunately, there isn't much that's automatic with printing; instead you must go through a number of mechanical, non-OO steps in order to print. Printing a component graphically can be slightly more automatic: by default, the **print( )** method calls **paint( )** to do it's work. There are times when this is satisfactory, but if you want to do anything more specialized you must know that you're printing so you can in particular find out the page dimensions.

The following example demonstrates the printing of both text and graphics, and the different approaches you can use for printing graphics. In addition, it tests the printing support (printing in Java 1.1 is a bit "fragile" and the example explores this):

```
//: PrintDemo.java
// Printing with Java 1.1
import java.awt.*;
import java.awt.event.*;

public class PrintDemo extends Frame {
  Button
    printText = new Button("Print Text"),
    printGraphics = new Button("Print Graphics");
  TextField ringNum = new TextField(3);
  Choice faces = new Choice();
  Graphics g = null;
  Plot plot = new Plot3(); // Try different plots
  Toolkit tk = Toolkit.getDefaultToolkit();
  public PrintDemo() {
    ringNum.setText("3");
    ringNum.addTextListener(new RingL());
    Panel p = new Panel();
    p.setLayout(new FlowLayout());
    printText.addActionListener(new TBL());
    p.add(printText);
    p.add(new Label("Font:"));
    p.add(faces);
    printGraphics.addActionListener(new GBL());
    p.add(printGraphics);
    p.add(new Label("Rings:"));
    p.add(ringNum);
    setLayout(new BorderLayout());
```

```
    add(p, BorderLayout.NORTH);
    add(plot, BorderLayout.CENTER);
    String fontList[] = tk.getFontList();
    for(int i = 0; i < fontList.length; i++)
      faces.add(fontList[i]);
    faces.select("Serif");
  }
  class PrintData {
    public PrintJob pj;
    public int pageWidth, pageHeight;
    PrintData(String jobName) {
      pj = getToolkit().getPrintJob(
        PrintDemo.this, jobName, null);
      if(pj != null) {
        pageWidth = pj.getPageDimension().width;
        pageHeight = pj.getPageDimension().height;
        g = pj.getGraphics();
      }
    }
    void end() { pj.end(); }
  }
  class ChangeFont {
    private int stringHeight;
    ChangeFont(String face, int style, int point) {
      if(g != null) {
        g.setFont(new Font(face, style, point));
        stringHeight =
          g.getFontMetrics().getHeight();
      }
    }
    int stringWidth(String s) {
      return g.getFontMetrics().stringWidth(s);
    }
    int stringHeight() { return stringHeight; }
  }
  class TBL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      PrintData pd =
        new PrintData("Print Text Test");
      // Null means print job canceled:
      if(pd == null) return;
      String s = "PrintDemo";
      ChangeFont cf = new ChangeFont(
        faces.getSelectedItem(), Font.ITALIC,72);
      g.drawString(s,
        (pd.pageWidth - cf.stringWidth(s)) / 2,
        (pd.pageHeight - cf.stringHeight()) / 3);

      s = "A smaller point size";
      cf = new ChangeFont(
        faces.getSelectedItem(), Font.BOLD, 48);
      g.drawString(s,
        (pd.pageWidth - cf.stringWidth(s)) / 2,
        (int)((pd.pageHeight -
           cf.stringHeight())/1.5));
      g.dispose();
      pd.end();
    }
  }
  class GBL implements ActionListener {
```

```
    public void actionPerformed(ActionEvent e) {
      PrintData pd =
        new PrintData("Print Graphics Test");
      if(pd == null) return;
      plot.print(g);
      g.dispose();
      pd.end();
    }
  }
  class RingL implements TextListener {
    public void textValueChanged(TextEvent e) {
      int i = 1;
      try {
        i = Integer.parseInt(ringNum.getText());
      } catch(NumberFormatException ex) {
        i = 1;
      }
      plot.rings = i;
      plot.repaint();
    }
  }
  static class WL extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
      System.exit(0);
    }
  }
  public static void main(String args[]) {
    Frame pdemo = new PrintDemo();
    pdemo.setTitle("Print Demo");
    pdemo.addWindowListener(new WL());
    pdemo.setSize(500, 500);
    pdemo.setVisible(true);
  }
}

class Plot extends Canvas {
  public int rings = 3;
}

class Plot1 extends Plot {
  // Default print() calls paint():
  public void paint(Graphics g) {
    int w = getSize().width;
    int h = getSize().height;
    int xc = w / 2;
    int yc = w / 2;
    int x = 0, y = 0;
    for(int i = 0; i < rings; i++) {
      if(x < xc && y < yc) {
        g.drawOval(x, y, w, h);
        x += 10; y += 10;
        w -= 20; h -= 20;
      }
    }
  }
}

class Plot2 extends Plot {
  // To fit the picture to the page, you must
  // know whether you're printing or painting:
```

```
  public void paint(Graphics g) {
    int w, h;
    if(g instanceof PrintGraphics) {
      PrintJob pj =
        ((PrintGraphics)g).getPrintJob();
      w = pj.getPageDimension().width;
      h = pj.getPageDimension().height;
    }
    else {
      w = getSize().width;
      h = getSize().height;
    }
    int xc = w / 2;
    int yc = w / 2;
    int x = 0, y = 0;
    for(int i = 0; i < rings; i++) {
      if(x < xc && y < yc) {
        g.drawOval(x, y, w, h);
        x += 10; y += 10;
        w -= 20; h -= 20;
      }
    }
  }
}

class Plot3 extends Plot {
  // Somewhat better. Separate
  // printing from painting:
  public void print(Graphics g) {
    // Assume it's a PrintGraphics object:
    PrintJob pj =
      ((PrintGraphics)g).getPrintJob();
    int w = pj.getPageDimension().width;
    int h = pj.getPageDimension().height;
    doGraphics(g, w, h);
  }
  public void paint(Graphics g) {
    int w = getSize().width;
    int h = getSize().height;
    doGraphics(g, w, h);
  }
  private void doGraphics(
      Graphics g, int w, int h) {
    int xc = w / 2;
    int yc = w / 2;
    int x = 0, y = 0;
    for(int i = 0; i < rings; i++) {
      if(x < xc && y < yc) {
        g.drawOval(x, y, w, h);
        x += 10; y += 10;
        w -= 20; h -= 20;
      }
    }
  }
} ///:~
```

The program allows you to select fonts from a **Choice** list (and you'll see that the number of fonts available in Java 1.1 is still extremely limited, and has nothing to do with any extra fonts you install on your machine). It uses these to print out text in bold and italic and in different sizes. In addition, a new type of component called a **Plot** is created to demonstrate graphics. A **Plot** has rings that it will display

on the screen and print onto paper, and the three derived classes **Plot1**, **Plot2** and **Plot3** perform these tasks in different ways so you can see your alternatives when printing graphics. Also, you can change the number of rings in a plot – this is interesting because it shows the printing fragility in Java 1.1, since on my system the printer gave error messages and didn't print correctly when the ring count got "too high" (whatever that means), but worked fine when the count was "low enough." You will notice, too, that the page dimensions produced when printing do not seem to correspond to the actual dimensions of the page. This may be fixed in a future release of Java, and you can use this program to test it.

This program encapsulates functionality inside inner classes whenever possible, to facilitate reuse. For example, whenever you want to begin a print job (whether for graphics or text), you must create a **PrintJob** object, which has its own **Graphics** object along with the width and height of the page. The creation of a **PrintJob** and extraction of page dimensions is encapsulated in the **PrintData** class.

printing text

Conceptually, printing text is straightforward: you choose a typeface and size, decide where the string should go on the page, and draw it with **Graphics.drawString( )**. This means, however, that you must perform all the calculations of exactly where each line will go on the page to make sure it doesn't run off the end of the page or collide with other lines. If you want to make a word processor, your work is cut out for you.

ChangeFont encapsulates a little of the process of changing from one font to another by automatically creating a new **Font** object with your desired typeface, style (**Font.BOLD** or **Font.ITALIC** – there's no support for underline, strikethrough, etc.) and point size. It also simplifies the calculation of the width and height of a string.

When you press the "Print text" button, the **TBL** listener is activated. You can see that it goes through two iterations of creating a **ChangeFont** object and calling **drawString( )** to print out the string in a calculated position, centered and 1/3 and 2/3 down the page, respectively. Notice whether these calculations actually produce the expected results (they didn't with the version I used).

printing graphics

When you press the "Print graphics" button the **GBL** listener is activated. The creation of a **PrintData** object initializes **g**, and then you simply call **print( )** for the component you want to print. To force printing you must call **dispose( )** for the **Graphics** object and **end( )** for the **PrintData** object (which turns around and calls **end( )** for the **PrintJob**).

The actual work is going on inside the **Plot** object. You can see that the base-class **Plot** is very simple – it extends **Canvas** and contains an **int** called **rings** to indicate how many concentric rings to draw on this particular **Canvas**. The three derived classes show different approaches to accomplishing the same goal: drawing on both the screen and on the printed page.

Plot1 takes the simplest approach to coding: ignore the fact that there are differences in painting and printing, and just override **paint( )**. The reason this works is that the default **print( )** method simply turns around and calls **paint( )**. However, you'll notice that the size of the output depends on the size of the on-screen canvas, which makes sense since the **width** and **height** are determined by calling **Canvas.getSize( )**. The other situation where this is acceptable is if your image is always a fixed size.

When the size of the drawing surface is important, then you must discover the dimensions. Unfortunately, this turns out to be awkward, as you can see in **Plot2**. For some possibly very good reason that I don't know, you cannot simply ask the **Graphics** object the dimensions of its drawing surface. This would have made the whole process quite elegant. Instead, to see if you're printing rather than painting, you must detect the **PrintGraphics** using the RTTI **instanceof** keyword (described in Chapter 11), then downcast and call the sole **PrintGraphics** method: **getPrintJob( )**. Now you have a handle to the **PrintJob** and you can find out the width and height of the paper. This is a very hacky approach, but perhaps there is some rational reason for it (on the other hand, you've seen some of the other library designs by now so you may have the impression that the designers were in fact just hacking around…).

You can see that **paint( )** in **Plot2** goes through both possibilities of printing or painting. But since the **print( )** method should be called when printing, why not use that? This approach is used in **Plot3**, and

it eliminates the need to use **instanceof** since inside **print( )** you can just assume that you can cast to a **PrintGraphics** object. This is a little better. The situation is improved by placing the common drawing code (once the dimensions have been detected) inside a separate method **doGraphics( )**.

the clipboard

Java 1.1 supports limited operations with the system clipboard. You can copy **String** objects to the clipboard as text, and you can paste text from the clipboard into **String** objects. Of course, the clipboard is designed to hold any type of data, but how this data is represented on the clipboard is up to the program doing the cutting and pasting. Although it currently only supports string data, the Java clipboard API provides for extensibility through the concept of a "flavor." When data comes off the clipboard, it has an associated set of flavors that it can be converted to (for example, a graph might be represented as a string of numbers or as an image) and you can see if that particular clipboard data supports the flavor you're interested in.

The following program is a simple demonstration of cut, copy and paste with **String** data in a **TextArea**. One thing you'll notice is that the keyboard sequences you normally use for cutting, copying and pasting also work. But if you look at any **TextField** or **TextArea** in any other program you'll find they also automatically support the clipboard key sequences. This example simply adds programmatic control of the clipboard, and you could use these techniques if you want to capture clipboard text into some non-**TextComponent**.

```
//: CutAndPaste.java
// Using the clipboard from Java 1.1
import java.awt.*;
import java.awt.event.*;
import java.awt.datatransfer.*;

public class CutAndPaste extends Frame {
  MenuBar mb = new MenuBar();
  Menu edit = new Menu("Edit");
  MenuItem
    cut = new MenuItem("Cut"),
    copy = new MenuItem("Copy"),
    paste = new MenuItem("Paste");
  TextArea text = new TextArea(20,20);
  Clipboard clipbd =
    getToolkit().getSystemClipboard();
  public CutAndPaste() {
    cut.addActionListener(new CutL());
    copy.addActionListener(new CopyL());
    paste.addActionListener(new PasteL());
    edit.add(cut);
    edit.add(copy);
    edit.add(paste);
    mb.add(edit);
    setMenuBar(mb);
    add(text, BorderLayout.CENTER);
  }
  class CopyL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      String selection = text.getSelectedText();
      StringSelection clipString =
        new StringSelection(selection);
      clipbd.setContents(clipString, clipString);
    }
  }
  class CutL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      String selection = text.getSelectedText();
```

```
        StringSelection clipString =
          new StringSelection(selection);
        clipbd.setContents(clipString, clipString);
        text.replaceRange("",
          text.getSelectionStart(),
          text.getSelectionEnd());
      }
    }
    class PasteL implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        Transferable clipData =
          clipbd.getContents(CutAndPaste.this);
        try {
          String clipString =
            (String)clipData.
              getTransferData(
                DataFlavor.stringFlavor);
          text.replaceRange(clipString,
            text.getSelectionStart(),
            text.getSelectionEnd());
        } catch(Exception ex) {
          System.out.println("not String flavor");
        }
      }
    }
    static class WL extends WindowAdapter {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    }
    public static void main(String args[]) {
      CutAndPaste cp = new CutAndPaste();
      cp.setSize(300,200);
      cp.setVisible(true);
      cp.addWindowListener(new WL());
    }
  } ///:~
```

The creation and addition of the menu and **TextArea** should by now seem a pedestrian activity. What's different is the creation of the **Clipboard** field **clipbd**, which is done through the **Toolkit**.

All the action takes place in the listeners. The **CopyL** and **CutL** listeners are the same except for the last line of **CutL**, which erases the line that's been copied. The special two lines are the creation of a **StringSelection** object from the **String**, and the call to **setContents( )** with this **StringSelection**. That's all there is to putting a **String** on the clipboard.

In **PasteL** data is pulled off the clipboard using **getContents( )**. What comes back is a fairly anonymous **Transferable** object, and you don't really know what it contains. One way to find out is to call **getTransferDataFlavors( )**, which returns an array of **DataFlavor** objects indicating which flavors are supported by this particular object. You can also ask it directly with **isDataFlavorSupported( )**, passing in the flavor you're interested in. Here, however, the bold approach is taken: **getTransferData( )** is called assuming that the contents supports the **String** flavor, and if it doesn't the problem is sorted out in the exception handler.

In the future you can expect more data flavors to be supported.

visual programming & Beans

So far in this book you've seen how valuable Java is for creating reusable pieces of code. The "most reusable" unit of code has been the class, since it comprises a cohesive unit of characteristics (fields) and behaviors (methods) that can be reused either directly via composition, or through inheritance.

Inheritance and polymorphism are essential parts of object-oriented programming, but in the majority of cases when you're putting together an application, what you really want is components that do exactly what you need. You'd like to drop these parts into your design like the electronic engineer puts together chips on a circuit board (or even, in the case of Java, onto a Web page). It seems, too, that there should be some way to accelerate this "module assembly" style of programming.

"Visual programming" first became successful – *very* successful – with Microsoft's Visual Basic (VB), followed by a second-generation design in Borland's Delphi (the primary inspiration for the JavaBeans design). With these programming tools the components are represented visually, which makes sense since they usually display some kind of visual component like a button or a text field. The visual representation, in fact, is often exactly the way the component will look in the running program. So part of the process of visual programming involves dragging a component from a pallet and dropping it onto your form – the application builder tool writes code as you do this, and that code will cause the component to be created in the running program.

Simply dropping the component onto a form is normally not enough to complete the program. Usually you'll need to change the characteristics of a component, such as what color it is or what text is on it or what database it's connected to, etc. Characteristics that can be modified at design time are referred to as *properties*. You can manipulate the properties of your component inside the application builder tool, and when you create the program this configuration data is saved so that it can be rejuvenated when the program is started.

By now you're probably used to the idea that an object is more than characteristics; it's also a set of behaviors. At design-time, the behaviors of a visual component are represented by *events*, meaning "here's something that can happen to the component." Normally, you decide what you want to happen when an event occurs by tying code to that event.

Here's the critical part: the application builder tool is able to dynamically interrogate the component to find out what properties and events the component supports. Once it knows what they are, it can display the properties and allow you to change those (saving the state when you build the program), and also display the events. Generally you do something like double clicking on an event and the application builder tool creates a code body and ties it to that particular event. All you have to do at that point is write the code that executes when the event occurs.

All this adds up to a lot of work that's done for you by the application builder tool. As a result you can focus on what the program looks like and what it is supposed to do, and rely on the application builder tool to manage the connection details for you. The reason that visual programming tools have been so successful is that they dramatically speed up the process of building an application – certainly the user interface, but often other portions of the application as well.

what is a Bean?

After the dust settles, then, a component is really just a block of code, typically embodied in a class. The key issue is the ability for the application builder tool to discover the properties and events for that component. To create a VB component, the programmer had to write a fairly complicated piece of code, following certain conventions to expose the properties and events. Delphi was a second-generation visual programming tool and the language was actively designed around visual programming so it is much easier to create a visual component. However, Java has brought the creation of visual components to its most advanced state with JavaBeans, because a Bean is just a class. You don't have to write any extra code or use special language extensions in order to make something a Bean. The only thing you need to do, in fact, is slightly modify the way you name your methods. It is the method name that tells the application builder tool whether this is a property, an event or just an ordinary method.

In the Java documentation, this naming convention is mistakenly termed a "design pattern." This is unfortunate since design patterns (see Chapter 16) are challenging enough without this sort of confusion. It's not a design pattern, it's just a naming convention and it's fairly simple:

1. For a property named **xxx**, you typically create two methods: **getXxx( )** and **setXxx( )**. Note that the first letter after get or set is automatically decapitalized to produce the property name. The type produced by the "get" function is the same as the type of the argument to the "set" function. The name of the property and the type for the "get" and "set" are not related.

2. For a boolean property, you can use the "get" and "set" approach above, but you may also use "is" instead of "get."

3. Ordinary methods of the Bean don't conform to the above naming convention, but they're **public**.

4. For events, you use the "listener" approach. It's exactly the same as you've been seeing: **addFooBarListener(FooBarListener)** and **removeFooBarListener(FooBarListener)** to handle a **FooBarEvent**. Most of the time the built-in events and listeners will satisfy your needs, but you can also create your own events and listener interfaces.

Point 1 above answers a question about something you may have noticed in the change from Java 1.0 to Java 1.1: a number of method names have had small, apparently meaningless name changes. Now you can notice that most of those changes had to do with adapting to the "get" and "set" naming convention, in order to make that particular component into a Bean.

We can use these guidelines to create a very simple Bean:

```
//: Frog.java
// A trivial Java bean
import java.awt.*;
import java.awt.event.*;

class Spots {}

public class Frog {
  private int jumps;
  private Color color;
  private Spots spots;
  private boolean jmpr;
  public int getJumps() { return jumps; }
  public void setJumps(int newJumps) {
    jumps = newJumps;
  }
  public Color getColor() { return color; }
  public void setColor(Color newColor) {
    color = newColor;
  }
  public Spots getSpots() { return spots; }
  public void setSpots(Spots newSpots) {
    spots = newSpots;
  }
  public boolean isJumper() { return jmpr; }
  public void setJumper(boolean j) { jmpr = j; }
  public void addActionListener(
      ActionListener l) {
    //...
  }
  public void removeActionListener(
      ActionListener l) {
    // ...
```

```
  }
  public void addKeyListener(KeyListener l) {
    // ...
  }
  public void removeKeyListener(KeyListener l) {
    // ...
  }
  // An "ordinary" public method:
  public void croak() {
    System.out.println("Ribbet!");
  }
} ///:~
```

First, you can see that it's just a class. Normally, all your fields will be **private**, and only accessible through methods. Following the naming convention, the properties are **jumps**, **color**, **spots** and **jumper** (note the change in case of the first letter in the property name). Although the name of the internal identifier is the same as the name of the property in the first three cases, in **jumper** you can see that the property name does not force you to use any particular name for internal variables (or indeed, to even *have* any internal variable for that property).

The events this Bean handles are **ActionEvent** and **KeyEvent**, based on the naming of the "add" and "remove" methods for the associated listener. Finally you can see the ordinary method **croak( )** is still part of the Bean simply because it's a **public** method, not because it conforms to any naming scheme.

extracting BeanInfo with the Introspector

One of the most critical parts of the Bean scheme occurs when you drag a Bean off a palette and plop it down on a form. The application builder tool must be able to create the Bean (which it can do if there's a default constructor) and then, without access to the Bean's source code, extract all the necessary information to create the property sheet and event handlers.

Part of the solution is already evident from the end of Chapter 11: Java 1.1 *reflection* allows all the methods of an anonymous class to be discovered. This is perfect for solving the Bean problem without requiring you to use any extra language keywords like those required in other visual programming languages. In fact, one of the prime reasons reflection was added to Java 1.1 was to support Beans (as well as object serialization and remote method invocation). So you might expect that the creator of the application builder tool would have to reflect each Bean and hunt through its methods to find the properties and events for that Bean.

This is certainly possible, but the Java designers wanted to provide a standard interface for everyone to use, not only to make Beans simpler to use but also to provide a standard gateway to the creation of more complex Beans. This interface is the **Introspector** class, and the most important method in this class is the **static getBeanInfo( )**. You pass a **Class** handle to this method, and it fully interrogates that class and returns a **BeanInfo** object that you can then dissect to find properties, methods and events.

Normally you won't care about any of this – you'll probably get most of your Beans off the shelf from vendors and you don't need to know all the magic that's going on underneath. You'll simply drag your beans onto your form, then configure their properties and write handlers for the events you're interested in. However, it's an interesting and educational exercise to use the **Introspector** to display information about a Bean, so here's a tool that does it:

```
//: BeanDumper.java
// A method to introspect a bean
import java.beans.*;
import java.lang.reflect.*;

public class BeanDumper {
  public static void dump(Class bean){
    BeanInfo bi = null;
    try {
      bi = Introspector.getBeanInfo(
```

```
      bean, java.lang.Object.class);
  } catch(IntrospectionException ex) {
    System.out.println("Couldn't introspect " +
      bean.getName());
    System.exit(1);
  }
  PropertyDescriptor properties[] =
    bi.getPropertyDescriptors();
  for(int i = 0; i < properties.length; i++) {
    Class p = properties[i].getPropertyType();
    System.out.println(
      "Property type:\n  " + p.getName());
    System.out.println(
      "Property name:\n  " +
      properties[i].getName());
    Method readMethod =
      properties[i].getReadMethod();
    if(readMethod != null)
      System.out.println(
        "Read method:\n  " +
        readMethod.toString());
    Method writeMethod =
      properties[i].getWriteMethod();
    if(writeMethod != null)
      System.out.println(
        "Write method:\n  " +
        writeMethod.toString());
    System.out.println("====================");
  }
  System.out.println("Public methods:");
  MethodDescriptor methods[] =
    bi.getMethodDescriptors();
  for(int i = 0; i < methods.length; i++)
    System.out.println(
      methods[i].getMethod().toString());
  System.out.println("======================");
  System.out.println("Event support:");
  EventSetDescriptor events[] =
    bi.getEventSetDescriptors();
  for(int i = 0; i < events.length; i++) {
    System.out.println("Listener type:\n  " +
      events[i].getListenerType().getName());
    Method lm[] =
      events[i].getListenerMethods();
    for(int j = 0; j < lm.length; j++)
      System.out.println(
        "Listener method:\n  " +
        lm[j].getName());
    MethodDescriptor lmd[] =
      events[i].getListenerMethodDescriptors();
    for(int j = 0; j < lmd.length; j++)
      System.out.println(
        "Method descriptor:\n  " +
        lmd[j].getMethod().toString());
    Method addListener =
      events[i].getAddListenerMethod();
    System.out.println(
        "Add Listener Method:\n  " +
      addListener.toString());
    Method removeListener =
```

```
        events[i].getRemoveListenerMethod();
      System.out.println(
        "Remove Listener Method:\n  " +
        removeListener.toString());
      System.out.println("====================");
    }
  }
  // Dump the class of your choice:
  public static void main(String args[]) {
    if(args.length < 1) {
      System.err.println("usage: \n" +
        "BeanDumper fully.qualified.class");
      System.exit(0);
    }
    Class c = null;
    try {
      c = Class.forName(args[0]);
    } catch(ClassNotFoundException ex) {
      System.err.println(
        "Couldn't find " + args[0]);
      System.exit(0);
    }
    dump(c);
  }
} ///:~
```

BeanDumper.dump( ) is the method that does all the work. First it tries to create a **BeanInfo** object, and if successful calls the methods of **BeanInfo** that produce information about properties, methods and events. In **Introspector.getBeanInfo( )**, you'll see there is a second argument. This tells the **Introspector** where to stop in the inheritance hierarchy. Here, it stops before it parses all the methods from **Object**, since we're not interested in seeing those.

For properties, **getPropertyDescriptors( )** returns an array of **PropertyDescriptor**s. For each **PropertyDescriptor** you can call **getPropertyType( )** to find the class of object that is passed in and out via the property methods. Then for each property you can get its pseudonym (extracted from the method names) with **getName( )**, the method for reading with **getReadMethod( )** and the method for writing with **getWriteMethod( )**. These last two methods return a **Method** object which can actually be used to invoke the corresponding method on the object (this is part of reflection).

For the public methods (including the property methods), **getMethodDescriptors( )** returns an array of **MethodDescriptor**s. For each one you can get the associated **Method** object and print out its name.

For the events, **getEventSetDescriptors( )** returns an array of (what else) **EventSetDescriptor**s. Each of these can be queried to find out the class of the listener, the methods of that listener class, and the add- and remove-listener methods. The **BeanDumper** program prints out all of this information.

If you invoke **BeanDumper** on the **Frog** class like this:

```
java BeanDumper Frog
```

the output, after removing extra details that are unnecessary here, is:

```
class name: Frog
Property type:
  Color
Property name:
  color
Read method:
  public Color getColor()
Write method:
  public void setColor(Color)
====================
```

```
Property type:
  Spots
Property name:
  spots
Read method:
  public Spots getSpots()
Write method:
  public void setSpots(Spots)
====================
Property type:
  boolean
Property name:
  jumper
Read method:
  public boolean isJumper()
Write method:
  public void setJumper(boolean)
====================
Property type:
  int
Property name:
  jumps
Read method:
  public int getJumps()
Write method:
  public void setJumps(int)
====================
Public methods:
public void setJumps(int)
public void croak()
public void removeActionListener(ActionListener)
public void addActionListener(ActionListener)
public int getJumps()
public void setColor(Color)
public void setSpots(Spots)
public void setJumper(boolean)
public boolean isJumper()
public void addKeyListener(KeyListener)
public Color getColor()
public void removeKeyListener(KeyListener)
public Spots getSpots()
======================
Event support:
Listener type:
  KeyListener
Listener method:
  keyTyped
Listener method:
  keyPressed
Listener method:
  keyReleased
Method descriptor:
  public void keyTyped(KeyEvent)
Method descriptor:
  public void keyPressed(KeyEvent)
Method descriptor:
  public void keyReleased(KeyEvent)
Add Listener Method:
  public void addKeyListener(KeyListener)
Remove Listener Method:
```

```
  public void removeKeyListener(KeyListener)
====================
Listener type:
  ActionListener
Listener method:
  actionPerformed
Method descriptor:
  public void actionPerformed(ActionEvent)
Add Listener Method:
  public void addActionListener(ActionListener)
Remove Listener Method:
  public void removeActionListener(ActionListener)
====================
```

This reveals most of what the **Introspector** sees as it produces a **BeanInfo** object from your Bean. You can see that the type of the property and its name are independent. Notice the decapitalization of the property name (the only time this doesn't occur is when the property name begins with more than one capital letter in a row). And remember that the method names you're seeing here (such as the read and write methods) are actually produced from a **Method** object that can be used to invoke the associated method on the object.

The public method list includes the methods that are not associated with a property or event, such as **croak( )**, as well as those that are. These are all the methods that you can call programmatically for a bean, and the application builder tool can choose to list all these while you're making method calls, to ease your task.

Finally, you can see that the events are fully parsed out into the listener, its methods and the add- and remove-listener methods. Basically, once you have the **BeanInfo**, you can find out everything of importance for the Bean. You can also call the methods for that Bean, even though you don't have any other information except the object itself (again, a feature of reflection).

a more sophisticated Bean

This next example is slightly more sophisticated, albeit frivolous. It's a canvas that draws a little circle around the mouse whenever it moves. When you press the mouse, the word "Bang!" appears in the middle of the screen, and an action listener is fired.

The properties you can change are the size of the circle as well as the color, size and text of the word that is displayed when you press the mouse. A **BangBean** also has its own **addActionListener( )** and **removeActionListener( )** so you can attach your own listener that will be fired when the user clicks on the **BangBean**. You should be able to recognize the property and event support:

```
//: TestBangBean.java
// A graphical Bean
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;

class BangBean extends Canvas
    implements Serializable {
  private int xm, ym;
  private int cSize = 20; // Circle size
  private String text = "Bang!";
  private int fontSize = 48;
  private Color tColor = Color.red;
  private ActionListener actionListener;
  public BangBean() {
    addMouseListener(new ML());
    addMouseMotionListener(new MM());
  }
```

```
  public int getCircleSize() { return cSize; }
  public void setCircleSize(int newSize) {
    cSize = newSize;
  }
  public String getBangText() { return text; }
  public void setBangText(String newText) {
    text = newText;
  }
  public int getFontSize() { return fontSize; }
  public void setFontSize(int newSize) {
    fontSize = newSize;
  }
  public Color getTextColor() { return tColor; }
  public void setTextColor(Color newColor) {
    tColor = newColor;
  }
  public void paint(Graphics g) {
    g.setColor(Color.black);
    g.drawOval(xm - cSize/2, ym - cSize/2,
      cSize, cSize);
  }
  // This is a unicast listener, which is
  // the simplest form of listener management:
  public void addActionListener (
      ActionListener l)
        throws TooManyListenersException {
    if(actionListener != null)
      throw new TooManyListenersException();
    actionListener = l;
  }
  public void removeActionListener(
      ActionListener l) {
    actionListener = null;
  }
  class ML extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
      Graphics g = getGraphics();
      g.setColor(tColor);
      g.setFont(
        new Font(
          "TimesRoman", Font.BOLD, fontSize));
      int width =
        g.getFontMetrics().stringWidth(text);
      g.drawString(text,
        (getSize().width - width) /2,
        getSize().height/2);
      g.dispose();
      // Call the listener's method:
      if(actionListener != null)
        actionListener.actionPerformed(
          new ActionEvent(BangBean.this,
            ActionEvent.ACTION_PERFORMED, null));
    }
  }
  class MM extends MouseMotionAdapter {
    public void mouseMoved(MouseEvent e) {
      xm = e.getX();
      ym = e.getY();
      repaint();
    }
```

```
    }
  }

  public class TestBangBean extends Frame {
    BangBean bb = new BangBean();
    public TestBangBean() {
      setTitle("BangBean Test");
      addWindowListener(new WL());
      setSize(300,300);
      add(bb, BorderLayout.CENTER);
      try {
        bb.addActionListener(new BBL());
      } catch(TooManyListenersException e) {}
    }
    class WL extends WindowAdapter {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    }
    class BBL implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        System.out.println("BangBean action");
      }
    }
    public static void main(String args[]) {
      Frame f = new TestBangBean();
      f.setVisible(true);
    }
  } ///:~
```

The first thing you'll notice is that **BangBean** implements the **Serializable** interface. This means that the application builder tool can "pickle" all the information for the **BangBean** using serialization after the program designer has adjusted the values of the properties. When the Bean is created as part of the running application, these "pickled" properties are restored so you get exactly what you designed.

You can see that all the fields are **private**, which is what you'll normally do with a Bean – allow access only through methods, usually using the "property" scheme.

When you look at the signature for **addActionListener( )**, you'll see that it may throw a **TooManyListenersException**. This indicates that it is *unicast*, which means it only notifies one listener when the event occurs. Normally, you'll use *multicast* events, which means that many listeners may be notified of an event. However, that runs into issues that you won't be ready for until the next chapter, so it will be revisited there (under the heading "JavaBeans revisited"). A unicast event sidesteps the problem.

When you press the mouse, the text is put in the middle of the **BangBean**, and if the **actionListener** field is not **null**, its **actionPerformed( )** is called. Whenever the mouse is moved, its new coordinates are captured and the canvas is repainted (erasing, you'll see, any text that's on the canvas).

TestBangBean creates a **Frame** and places a **BangBean** within it, attaching a simple **ActionListener** to the **BangBean**. Normally, of course, the application builder tool would create most of the code that uses the Bean.

When you run the **BangBean** through **BeanDumper**, you'll notice there are many more properties and actions than are evident from the above code. That's because **BangBean** is inherited from **Canvas**, and **Canvas** is itself a Bean, so you're seeing its properties and events as well.

more complex Bean support

You can see how remarkably simple it is to make a Bean. But you aren't limited to what you've seen here. The JavaBean design provides a simple point of entry but can also scale to more complex

situations. These situations are beyond the scope of this book but they will be briefly introduced here. You can find more details at http://www.java.sun.com/beans.

One place where you can add sophistication is with properties. The above examples have only shown single properties, but it's also possible to represent multiple properties in an array. This is called an *indexed property*. You simply provide the appropriate methods (again following a naming convention for the method names) and the **Introspector** recognizes an indexed property so your application builder tool can respond appropriately.

Properties may be *bound*, which means they will notify other objects via a **PropertyChangeEvent**. The other objects can then choose to change themselves based on the change to the Bean.

Properties may be *constrained*, which means other objects can veto a change to that property if it is unacceptable. The other objects are notified using a **PropertyChangeEvent**, and they may throw a **ProptertyVetoException** to prevent the change from happening and to restore the old values.

You can also change the way your Bean is represented at design time:

1. You can provide a custom property sheet for your particular Bean. The ordinary property sheet will be used for all other Beans, but yours is automatically invoked when your Bean is selected.

2. You can create a custom editor for a particular property, so the ordinary property sheet is used but when your special property is being edited, your editor will automatically be invoked.

3. You can provide a custom **BeanInfo** class for your Bean that produces different information than the default created by the **Introspector**.

4. It's also possible to turn "expert" mode on and off in all **FeatureDescriptor**s to distinguish between basic features and more complicated ones.

more to Beans

There's another issue that couldn't be addressed here. Whenever you create a Bean, you should expect that it will be run in a multithreaded environment. This means that you must understand the issues of threading, which will be introduced in the next chapter. You'll find a section there called "JavaBeans revisited" which will look at the problem and its solution.

summary

When you're building an applet for placement on a Web page, it's much easier to think about working within the restrictions of the AWT and applets, because what you're getting in return is a completely portable program that instantly runs on all platforms. That's a big advantage – as long as you must live within a Web page.

Once you step outside the confines of the Web and start making regular applications, the AWT can become a disadvantage. Not only does it allow you to "create a GUI that's equally mediocre on all platforms" but it's also very awkward and unpleasant to use compared with native application development tools on a particular platform. A good example of this is the problem of getting information out of dialog boxes into the application that spawned the dialog box. This problem has been solved in several ways with other class libraries, but with the AWT you're back to square one. (On the upside, JavaBeans can be created to provide more appealing interface components).

Web page applets are certainly a useful application of the Java language, but for it to be a truly successful language in the future Java will need to evolve into a tool that can make good native operating-system applications, and for this the AWT is not satisfactory. The new AWT and Beans in Java 1.1 take a step towards satisfying this, but they also move towards Sun's "universal GUI" vision, where all computer interfaces will look the same. This is admittedly an interesting (though potentially

frightening) experiment, but it doesn't solve the more immediate needs of programming on a particular platform.

It's highly likely that the AWT will persist in some form within future Java distributions to support Web applet programming, but it's not the right solution for native platform applications. As much as some would like to believe that the future of all programming lies in abandoning specific operating systems in favor of completely generic programming, the only way this might happen is if all windowed operating systems conformed to some set of standards so that a future windowing toolkit could be created that takes advantage of all those features. But in the meantime generic programs will only satisfy a portion of the application needs, and it's a bit far-fetched to think that programmers will abandon the ability to use the specific features of some operating system that might serve many tens of millions.

Various groups are working on improved libraries and even improvements to Java itself in order to support the solution of more general programming problems. How well these problems are solved may be crucial in determining Java's place in the future of programming.

exercises

1. Create an applet with a text field and 3 buttons. When you press each button, make some different text appear in the text field.

2. Add a check box to the applet created in exercise one, capture the event and insert different text into the text field.

3. Create an applet and add all the components that cause **action( )** to be called, then capture their events and display an appropriate message for each inside a text field.

4. Add to exercise three the components that can only be used with events detected by **handleEvent( )**. Override **handleEvent( )** and display appropriate messages for each inside a text field.

5. Create an applet with a **Button** and a **TextField**. Write a **handleEvent( )** so that if the button has the focus, characters typed into it will appear in the **TextField**.

6. Create an application and add to the main frame all the components described in this chapter, including menus and a dialog box.

7. Modify **TextNew.java** so that the characters in **t2** retain the original case that they were typed in, instead of automatically being forced to upper case.

8. Modify **CardLayout1.java** so that it uses the new Java 1.1 event model.

14

14: multiple threads

Objects provide a way to divide a program up into independent sections. Often you also need to turn a program into separate, independently-running processes.

Each of these independent processes is called a *thread*, and you program as if they run all by themselves, and have the CPU all to themselves. Some underlying mechanism is actually dividing up the CPU time for you, but in general you don't have to think about it, which makes programming with multiple threads a much easier task.

There are many possible uses for multithreading, but in general some part of your program is tied to a particular event or resource, and you don't want to hang up the rest of your program because of that. So you create a thread associated with that event or resource, and let it run independently of the main program. A good example is a "quit" button – you don't want to be forced to poll the quit button in every piece of code you write in your program, and yet you want the quit button to be responsive, as if you *were* checking it regularly. In fact, one of the most immediately compelling reasons for multithreading is to produce a responsive user interface.

responsive user interfaces

As a starting point, consider a program that performs some CPU-intensive operation and thus ends up ignoring user input and being unresponsive. This one, a combined applet/application, will simply display the result of a running counter:

```
//: Counter1.java
// A non-responsive user interface
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class Counter1 extends Applet {
  int count = 0;
  Button
    onOff = new Button("Toggle"),
    start = new Button("Start");
  TextField t = new TextField(10);
  boolean runFlag = true;
  public void init() {
    add(t);
    start.addActionListener(new StartL());
    add(start);
    onOff.addActionListener(new OnOffL());
    add(onOff);
  }
  public void go() {
    while (true) {
      try {
        Thread.currentThread().sleep(100);
      } catch (InterruptedException e){}
      if(runFlag)
        t.setText(Integer.toString(count++));
    }
  }
  class StartL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      go();
    }
  }
  class OnOffL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      runFlag = !runFlag;
    }
  }
  static class WL extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
      System.exit(0);
    }
  }
  public static void main(String args[]) {
    Counter1 applet = new Counter1();
    Frame aFrame = new Frame("Counter1");
    aFrame.addWindowListener(new WL());
    aFrame.add(applet, BorderLayout.CENTER);
    aFrame.setSize(300,200);
    applet.init();
    applet.start();
    aFrame.setVisible(true);
```

```
  }
} ///:~
```

At this point the AWT and applet code should be reasonably familiar from Chapter 13. The **go( )** method is where the program stays busy: it puts the current value of **count** into the **TextField t**, then increments **count**.

Part of the infinite loop inside **go( )** is to call **sleep( )**. **sleep( )** must be associated with a **Thread** object, and it turns out that every application has *some* thread associated with it (indeed, Java is based on threads and there are always some running along with your application). So regardless of whether you're actively using threads or not you can produce the current thread used by your program with **Thread.currentThread()** (a static method of the **Thread** class) and then call **sleep( )** for that thread.

Note that **sleep( )** may throw **InterruptedException**, although throwing such an exception is considered a hostile way to break from a thread and should be discouraged (once again, exceptions are for exceptional conditions, not normal flow of control). Interrupting a sleeping thread is included to support a future language feature.

When the **start** button is pressed, **go( )** is invoked. And upon examining **go( )**, you may naively think (as I did) that it should allow multithreading because it goes to sleep. That is, while the method is asleep it seems like the CPU could be busy monitoring other button presses. But it turns out the real problem is that **go( )** never returns, since it's in an infinite loop, and this means that **actionPerformed( )** never returns. Since you're stuck inside **actionPerformed( )** for the first keypress, the program can't handle any other events (to get out, you must somehow kill the process; the easiest way to do this is to press Control-C in the console window).

The basic problem here is that **go( )** needs to continue performing its operations, and at the same time it needs to return so **actionPerformed( )** can complete and the user interface can continue responding to the user. But in a conventional method like **go( )** it cannot continue *and* at the same time return control to the rest of the program. This sounds like an impossible thing to accomplish, as if the CPU must be in two places at once, but this is precisely what threading allows. It is a way to have more than one independent process running at the same time, and the CPU will pop around and give each process some of its time. Each process has the consciousness of constantly having the CPU all to itself, but the CPU's time is actually sliced between all the processes.

Of course, if you have more than one CPU then the operating system can dedicate each CPU to a set of threads or even a single thread, and then the whole program can run much faster. Multitasking and multithreading tend to be the most reasonable ways to utilize multiprocessor systems.

inheriting from Thread

The simplest way to create a thread is to inherit from class **Thread**, which has all the wiring necessary to create and run threads. The most important method for **Thread** is **run( )**, which you must override to make the thread do your bidding. Thus, **run( )** is the code that will be executed "simultaneously" with the other threads in a program.

The following example creates any number of threads which it keeps track of by assigning each thread a unique number, generated with a **static** variable. The **Thread**'s **run( )** method is overridden to count down each time it passes through its loop, and to finish when the count is zero (at the point when **run( )** returns, the thread is terminated).

```
//: SimpleThread.java
// Very simple Threading example

public class SimpleThread extends Thread {
  private int countDown = 5;
  private int threadNumber;
  private static int threadCount = 0;
  public SimpleThread() {
    threadNumber = ++threadCount;
    System.out.println("Making " + threadNumber);
```

```
  }
  public void run() {
    while(true) {
      System.out.println("Thread " +
        threadNumber + "(" + countDown + ")");
      if(--countDown == 0) return;
    }
  }
  public static void main(String args[]) {
    for(int i = 0; i < 5; i++)
      new SimpleThread().start();
    System.out.println("All Threads Started");
  }
} ///:~
```

A **run( )** method virtually always has some kind of loop that continues until the thread is no longer necessary, so you'll have to establish the condition on which to break out of this loop (or, in the above case, simply **return** from **run( )**). Very often, **run( )** is cast in the form of an infinite loop, which means that, barring some external call to **stop( )** or **destroy( )** for that thread, it will run forever (until the program completes).

In **main( )** you can see a number of threads being created and run. The special method that comes with the **Thread** class is **start( )**, which performs special initialization for the thread and then calls **run( )**. So the steps are: the constructor is called to build the object, then **start( )** configures the thread and calls **run( )**. If you don't call **start( )** (which you can do in the constructor, if that's appropriate) the thread will never be started.

The output for this program is:

```
Making 1
Making 2
Making 3
Making 4
Making 5
Thread 1(5)
Thread 1(4)
Thread 1(3)
Thread 1(2)
Thread 2(5)
Thread 2(4)
Thread 2(3)
Thread 2(2)
Thread 2(1)
Thread 1(1)
All Threads Started
Thread 3(5)
Thread 4(5)
Thread 4(4)
Thread 4(3)
Thread 4(2)
Thread 4(1)
Thread 5(5)
Thread 5(4)
Thread 5(3)
Thread 5(2)
Thread 5(1)
Thread 3(4)
Thread 3(3)
Thread 3(2)
Thread 3(1)
```

You'll note that nowhere in this example is **sleep( )** called, and yet the output indicates that each thread gets a portion of the CPU's time in which to execute. This shows that **sleep( )**, while it relies on the existence of a thread in which to execute, is not involved with either enabling or disabling threading. It's simply another method.

You can also see that the threads are not run in the order that they're created. In fact, the order in which an existing set of threads is attended to by the CPU is indeterminate, unless you go in and adjust the priorities using **Thread**'s **setPriority( )** method.

threading for a responsive interface

Now it's possible to solve the problem in **Counter1.java** with a thread. The trick is to place the "process" – that is, the loop that's inside **go( )** – inside the **run( )** method of a thread. When the user presses the **start** button, the thread is started, but then the *creation* of the thread completes and so even though the thread is running, the main process of the program (watching for and responding to user-interface events) can continue. Here's the solution:

```
//: Counter2.java
// A responsive user interface with threads
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

class SeparateProcess extends Thread {
  int count = 0;
  Counter2 c2;
  boolean runFlag = true;
  SeparateProcess(Counter2 c2) {
    this.c2 = c2;
    start();
  }
  public void run() {
    while (true) {
     try {
      sleep(100);
     } catch (InterruptedException e){}
     if(runFlag)
       c2.t.setText(Integer.toString(count++));
    }
  }
}

public class Counter2 extends Applet {
  SeparateProcess sp = null;
  Button
    onOff = new Button("Toggle"),
    start = new Button("Start");
  TextField t = new TextField(10);
  public void init() {
    add(t);
    start.addActionListener(new StartL());
    add(start);
    onOff.addActionListener(new OnOffL());
    add(onOff);
  }
  class StartL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if(sp == null)
        sp = new SeparateProcess(Counter2.this);
    }
```

```
  }
  class OnOffL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if(sp != null)
        sp.runFlag = !sp.runFlag;
    }
  }
  static class WL extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
      System.exit(0);
    }
  }
  public static void main(String args[]) {
    Counter2 applet = new Counter2();
    Frame aFrame = new Frame("Counter2");
    aFrame.addWindowListener(new WL());
    aFrame.add(applet, BorderLayout.CENTER);
    aFrame.setSize(300,200);
    applet.init();
    applet.start();
    aFrame.setVisible(true);
  }
} ///:~
```

Counter2 is now a very straightforward program, whose only job is to set up and maintain the user interface. But now, when the user presses the **start** button, a method is not called. Instead a thread of class **SeparateProcess** is created (the constructor starts it, in this case), and then the **Counter2** event loop continues. Note that the handle to the **SeparateProcess** is stored so that when you press the **onOff** button it can toggle the **runFlag** inside the **SeparateProcess** object, and that process (when it looks at the flag) can then start and stop itself (this could also have been accomplished by making **SeparateProcess** an inner class).

The class **SeparateProcess** is a simple extension of **Thread** with a constructor (that stores the **Counter2** handle and then runs the thread by calling **start( )**) and a **run( )** that essentially contains the code from inside **go( )** in **Counter1.java**. Because **SeparateProcess** knows that it holds a handle to a **Counter2**, it can reach in and access **Counter2**'s **TextField** when it needs to.

When you press the **onOff** button, you'll see a virtually instant response. Of course, the response isn't really instant, not like that of a system that's driven by interrupts. The counter only stops when the thread has the CPU and notices that the flag has changed.

combining the thread with the main class

In the above example you can see that the thread class is separate from the program's main class. This makes a lot of sense and is relatively easy to understand. There is, however, an alternate form that you will often see used which is not so clear but is usually more concise (which probably accounts for its popularity). This form combines the main program class with the thread class by making the main program class a thread. Since for a GUI program the main program class must be inherited from either **Frame** or **Applet**, this means that an interface must be used to paste on the additional functionality. This interface is called **Runnable**, and it contains the same basic method that **Thread** does. In fact, **Thread** also implements **Runnable**, which only specifies that there be a **run( )** method.

The use of the combined program/thread is not quite so obvious. When you start the program, you create an object that's **Runnable** in the process, but you don't start the thread. This must be done explicitly. You can see this in the following program, which reproduces the functionality of **Counter2**:

```
//: Counter3.java
// Using the Runnable interface to turn the
// main class into a thread.
import java.awt.*;
import java.awt.event.*;
```

```
import java.applet.*;

public class Counter3
    extends Applet implements Runnable {
  int count = 0;
  boolean runFlag = true;
  Thread selfThread = null;
  Button
    onOff = new Button("Toggle"),
    start = new Button("Start");
  TextField t = new TextField(10);
  public void init() {
    add(t);
    start.addActionListener(new StartL());
    add(start);
    onOff.addActionListener(new OnOffL());
    add(onOff);
  }
  public void run() {
    while (true) {
      try {
        Thread.currentThread().sleep(100);
      } catch (InterruptedException e){}
      if(runFlag)
        t.setText(Integer.toString(count++));
    }
  }
  class StartL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if(selfThread == null) {
        selfThread = new Thread(Counter3.this);
        selfThread.start();
      }
    }
  }
  class OnOffL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      runFlag = !runFlag;
    }
  }
  static class WL extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
      System.exit(0);
    }
  }
  public static void main(String args[]) {
    Counter3 applet = new Counter3();
    Frame aFrame = new Frame("Counter3");
    aFrame.addWindowListener(new WL());
    aFrame.add(applet, BorderLayout.CENTER);
    aFrame.setSize(300,200);
    applet.init();
    applet.start();
    aFrame.setVisible(true);
  }
} ///:~
```

Now the **run( )** is inside the class, but it's still dormant after **init( )** completes. When you press the **start** button, the thread is created (if it doesn't already exist) in the somewhat obscure expression:

```
new Thread(Counter3.this);
```

When something has a **Runnable** interface, it simply means it has a **run( )** method, but there's nothing special about that – it doesn't produce any innate threading abilities, like a class inherited from **Thread** has. So to produce a thread from a **Runnable** object, you must actually create a thread separately and hand it the **Runnable** object; there's a special constructor for this that takes a **Runnable** as its argument. You can then call **start( )** for that thread:

```
selfThread.start();
```

This performs the usual initialization and then calls **run( )**.

The convenient thing about the **Runnable interface** is that everything belongs to the same class. If you need to access something, you simply do it without going through a separate object. The penalty for this convenience is strict, though – you can only have a single thread running for that particular object (although you can create more objects of that type, or create other threads in different classes).

Note that the **Runnable** interface is not what imposes this restriction. It's the combination of **Runnable** and your main class that does it, since you can only have one object of your main class per application.

making many threads

Consider the creation of many different threads. You can't do this with the previous example, so you'll have to go back to having separate classes inherited from **Thread** to encapsulate the **run( )**. But this is a more general solution and easier to understand, so while the previous example shows a coding style you'll often see, I can't recommend it because it's just a little bit more confusing and less flexible.

The following example repeats the form of the above examples with counters and toggle buttons. But now all the information, including the button and text field, for a particular counter is inside its own object which is inherited from **Thread**:

```
//: Counter4.java
// If you separate your thread from the main
// class, you can have as many threads as you
// want.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

class Ticker extends Thread {
  Button b = new Button("Toggle");
  TextField t = new TextField(10);
  int count = 0;
  boolean runFlag = true;
  Ticker() {
    b.addActionListener(new ToggleL());
  }
  class ToggleL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      runFlag = !runFlag;
    }
  }
  public void run() {
    while (true) {
      if(runFlag)
        t.setText(Integer.toString(count++));
       try {
        sleep(100);
      } catch (InterruptedException e){}
    }
```

```
    }
  }

  public class Counter4 extends Applet {
    Button start = new Button("Start");
    boolean started = false;
    Ticker s[];
    boolean isApplet = true;
    int size;
    public void init() {
      // Get parameter "size" from Web page:
      if(isApplet)
        size =
          Integer.parseInt(getParameter("size"));
      s = new Ticker[size];
      for(int i = 0; i < s.length; i++) {
        s[i] = new Ticker();
        Panel p = new Panel();
        p.add(s[i].t);
        p.add(s[i].b);
        add(p);
      }
      start.addActionListener(new StartL());
      add(start);
    }
    class StartL implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        if(!started) {
          started = true;
          for(int i = 0; i < s.length; i++)
            s[i].start();
        }
      }
    }
    static class WL extends WindowAdapter {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    }
    public static void main(String args[]) {
      Counter4 applet = new Counter4();
      // This isn't an applet, so set the flag and
      // produce the parameter values from args[]:
      applet.isApplet = false;
      applet.size =
        (args.length == 0 ? 5 :
          Integer.parseInt(args[0]));
      Frame aFrame = new Frame("Counter4");
      aFrame.addWindowListener(new WL());
      aFrame.add(applet, BorderLayout.CENTER);
      aFrame.setSize(200, applet.size * 50);
      applet.init();
      applet.start();
      aFrame.setVisible(true);
    }
  } ///:~
```

This way, **Ticker** contains not only its threading equipment but also the way to control and display the thread. Thus, you can create as many threads as you want without explicitly creating the windowing components.

In **Counter4** there's an array of **Ticker** objects called **s**. For maximum flexibility, the size of this array is initialized by reaching out into the Web page itself, using applet parameters. Here's what the size parameter looks like on the page, embedded inside the applet description:

```
<applet code=Counter4 width=600 height=600>
<param name=size value="20">
</applet>
```

The **param**, **name** and **value** are all Web-page keywords. **name** is what you'll be referring to in your program, and **value** can be any string, not just something that resolves to a number.

You'll notice that the determination of the size of the array **s** is done inside **init( )**, and not as part of an inline definition of **s**. That is, you *cannot* say as part of the class definition (outside of any methods):

```
int size = Integer.parseInt(getParameter("size"));
Ticker s[] = new Ticker[size];
```

Actually, you can compile this but you'll get a strange null-pointer exception at run time. It works fine if you move the **getParameter( )** initialization inside of **init( )**. The applet framework performs the necessary startup to grab the parameters before entering **init( )**.

In addition, this code is set up to be either an applet or an application. When it's an application the **size** argument is extracted from the command line (or a default value is provided).

Once the size of the array is established, new **Ticker** objects are created and the button and text field for each one is added to the applet.

Pressing the **start** button means looping through the whole array of **ticker**s and calling **start( )** for each one. Remember, **start( )** performs necessary thread initialization and then calls **run( )** for that thread.

The **ToggleL** listener simply inverts the flag in **Ticker**, and when the associated process next takes note it can react accordingly.

One value of this example is that it allows you to easily create large sets of independent processes and to monitor their behavior. In this case, you'll see that as the number of processes gets larger, your machine will probably show more and more divergence in the displayed numbers because of the way that the threads are served.

You can also experiment to discover how important the **sleep(100)** is inside **Ticker.run( )**. If you remove the **sleep( )**, things will work fine until you press a toggle button. Then that particular process has a false **runFlag** and the **run( )** is just tied up in a very tight infinite loop, which appears very difficult to break during multithreading, so the responsiveness and speed of the program really bogs down.

daemon threads

A "daemon" thread is one that is supposed to provide a general service in the background as long as the program is running, but is not part of the essence of the program. Thus, when all the non-daemon threads complete then the program is terminated.

You can find out if a thread is a daemon by calling **isDaemon( )**, and you can turn the daemonhood of a thread on and off with **setDaemon( )**. If a thread is a daemon, then any threads it creates will automatically be daemons.

The following example demonstrates daemon threads:

```
//: Daemons.java
// Daemonic behavior

class Daemon extends Thread {
  int j;
  static final int size = 10;
  Thread t[] = new Thread[size];
```

```
  Daemon() {
    setDaemon(true);
    start();
  }
  public void run() {
    for(int i = 0; i < size; i++)
      t[i] = new DaemonSpawn(i);
    for(int i = 0; i < size; i++)
      System.out.println(
        "t[" + i + "].isDaemon() = "
        + t[i].isDaemon());
    while(true) j++;
  }
}

class DaemonSpawn extends Thread {
  int i;
  DaemonSpawn(int i) {
    System.out.println(
      "DaemonSpawn " + i + " started");
    start();
  }
  public void run() {
    while(true) i++;
  }
}

public class Daemons {
  public static void main(String args[]) {
    Thread d = new Daemon();
    System.out.println(
      "d.isDaemon() = " + d.isDaemon());
  }
} ///:~
```

The **Daemon** thread sets its daemon flag to "true" and then spawns a bunch of other threads to show that they are also daemons. Then it goes into an infinite loop and counts, which is what the **DaemonSpawn** threads do as well.

When all the daemon threads are created, **Daemons.main( )** has nothing else to do. Since there are nothing but daemon threads running, the program should terminate (note that this program did not produce the desired behavior on all JVM's when this was written).

sharing limited resources

You can think of a single-threaded program as one lonely entity moving around through your problem space and doing one thing at a time. Because there's only one entity, you never have to think about the problem of two entities trying to use the same resource at the same time, like two people trying to park in the same space, walk through a door at the same time, or even talk at the same time.

With multithreading, things aren't lonely anymore but you now have the possibility of two or more threads trying to use the same limited resource at once. Colliding over a resource must be prevented, or else you'll have two threads trying to access the same bank account at the same time, or print to the same printer, or adjust the same valve, etc.

improperly accessing resources

Consider a variation on the counters that have been used so far in this chapter. In the following example, each thread contains two counters which are incremented and displayed inside **run( )**. In

addition, there's another thread of class **Watcher** that is watching the counters to see if they're always equivalent. This seems like a needless activity, since looking at the code it appears obvious that the counters will always be the same. But that's where the surprise comes in. Here's the first version of the program:

```
//: Sharing1.java
// Problems with resource sharing while threading
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

class TwoCounter extends Thread {
  boolean started = false;
  TextField
    t1 = new TextField(5),
    t2 = new TextField(5);
  Label l = new Label("count1 == count2");
  int count1 = 0, count2 = 0;
  public void run() {
    while (true) {
      t1.setText(Integer.toString(count1++));
      t2.setText(Integer.toString(count2++));
      try {
        sleep(500);
      } catch (InterruptedException e){}
    }
  }
  void synchTest() {
    Sharing1.incrementAccess();
    if(count1 != count2)
      l.setText("Unsynched");
  }
}

class Watcher extends Thread {
  Sharing1 p;
  Watcher(Sharing1 p) {
    this.p = p;
    start();
  }
  public void run() {
    while(true) {
      for(int i = 0; i < p.s.length; i++)
        p.s[i].synchTest();
      try {
        sleep(500);
      } catch (InterruptedException e){}
    }
  }
}

public class Sharing1 extends Applet {
  static int accessCount = 0;
  static TextField aCount = new TextField("0", 10);
  static void incrementAccess() {
    accessCount++;
    aCount.setText(Integer.toString(accessCount));
  }
  Button
    start = new Button("Start"),
```

```
  observer = new Button("Observe");
TwoCounter s[];
boolean isApplet = true;
int numCounters = 0;
int numObservers = 0;
public void init() {
  if(isApplet) {
    numCounters =
      Integer.parseInt(getParameter("size"));
    numObservers =
      Integer.parseInt(
        getParameter("observers"));
  }
  s = new TwoCounter[numCounters];
  for(int i = 0; i < s.length; i++) {
    s[i] = new TwoCounter();
    Panel p = new Panel();
    p.add(s[i].t1);
    p.add(s[i].t2);
    p.add(s[i].l);
    add(p);
  }
  Panel p = new Panel();
  start.addActionListener(new StartL());
  p.add(start);
  observer.addActionListener(new ObserverL());
  p.add(observer);
  p.add(new Label("Access Count"));
  p.add(aCount);
  add(p);
}
class StartL implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    for(int i = 0; i < s.length; i++) {
      if(!s[i].started) {
        s[i].started = true;
        s[i].start();
      }
    }
  }
}
class ObserverL implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    for(int i = 0; i < numObservers; i++)
      new Watcher(Sharing1.this);
  }
}
static class WL extends WindowAdapter {
  public void windowClosing(WindowEvent e) {
    System.exit(0);
  }
}
public static void main(String args[]) {
  Sharing1 applet = new Sharing1();
  // This isn't an applet, so set the flag and
  // produce the parameter values from args[]:
  applet.isApplet = false;
  applet.numCounters =
    (args.length == 0 ? 5 :
      Integer.parseInt(args[0]));
```

```
    applet.numObservers =
      (args.length < 2 ? 5 :
        Integer.parseInt(args[1]));
    Frame aFrame = new Frame("Sharing1");
    aFrame.addWindowListener(new WL());
    aFrame.add(applet, BorderLayout.CENTER);
    aFrame.setSize(350, applet.numCounters*100);
    applet.init();
    applet.start();
    aFrame.setVisible(true);
  }
} ///:~
```

As before, each counter contains its own display components: two text fields and a label that initially indicates the counts are equivalent. In run, **count1** and **count2** are incremented and displayed in a manner that would seem to keep them exactly identical. Then **sleep( )** is called; without this call the program becomes very balky because it becomes hard for the CPU to swap tasks.

The **synchTest( )** method performs the apparently useless activity of checking to see if **count1** is equivalent to **count2**; if they are not equivalent it sets the label to "Unsynched" to indicate this. But first, it calls a static member of the class **Sharing1** that increments and displays an access counter to show how many times this check has successfully occurred (the reason for this will become apparent in future variations of this example).

The **Watcher** class is a thread whose job is to call **synchTest( )** for all the **TwoCounter** objects that are active. It does this by stepping through the array that's kept in the **Sharing1** object. You can think of the **Watcher** as constantly peeking over the shoulders of the **TwoCounter** objects.

Sharing1 contains an array of **TwoCounter** objects that it initializes in **init( )** and starts as threads when you press the "start" button. Later, when you press the "Observe" button, one or more observers are created and freed upon the unsuspecting **TwoCounter** threads.

Note that to run this as an applet in a browser, your Web page will need to contain the lines:

```
<applet code=Sharing1 width=650 height=500>
<param name=size value="20">
<param name=observers value="1">
</applet>
```

You can change the width, height and parameters to suit your experimental tastes. By changing the **size** and **observers** you'll change the behavior of the program. You can also see that this program is set up to run as a stand-alone application, by pulling the arguments from the command line (or providing defaults).

OK, here's the surprising part. In **TwoCounter.run( )**, the infinite loop is just passing over and over the adjacent lines:

```
t1.setText(Integer.toString(count1++));
t2.setText(Integer.toString(count2++));
```

(as well as sleeping, but that's not important here). When you run the program, however, you'll discover that **count1** and **count2** will be observed (by the **Watcher**) to be unequal at times! This is because of the nature of threads – they can be suspended at any time. So at times, the suspension occurs *between* the execution of the above two lines, and the **Watcher** thread happens to come along and perform the comparison at just this moment, thus finding the two counters to be different.

This example shows a fundamental problem when using threads. You never know when a thread may be run. Imagine sitting at a table with a fork, about to spear the last piece of food on your plate and as your fork reaches for it, it suddenly vanishes (because your thread was suspended, and another thread came in and stole the food). That's the problem that you're dealing with.

Sometimes you don't care if a resource is being accessed at the same time you're trying to use it (the food is on some other plate). But for multithreading to work, you need some way to prevent two threads from accessing the same resource, at least during critical periods.

Preventing this kind of collision is simply a matter of putting a lock on a resource when one thread is using it. The first thread that accesses a resource locks it, and then the other threads cannot access that resource until it is unlocked, at which time another thread locks and uses it, etc. If the front seat of the car is the limited resource, the child who shouts "dibs!" asserts the lock.

how Java shares resources

Java has built-in support to prevent collisions over one kind of resource: the memory in an object. Since you typically make the data elements of a class **private** and access to that memory is then only provided through methods, collision prevention is achieved by making a particular method **synchronized**. Only one **synchronized** method at a time can be called for a particular object. Here are simple **synchronized** methods:

```
synchronized void f() { /* ... */ }
synchronized void g(){ /* ... */ }
```

Each object contains a single lock that's automatically part of the object (you don't have to write any special code). When you call any **synchronized** method, that object is locked and no other **synchronized** methods can be called until the first one finishes and thus releases the lock. In the above example, if **f( )** is called for an object, **g( )** cannot be called for the same object until **f( )** is completed and releases the lock. Thus there's a single lock that's shared by all the **synchronized** methods of a particular object, and this lock prevents common memory from being written by more than one method at a time (i.e. more than one thread at a time).

There's also a single lock per class, so that **synchronized static** methods can lock each other out from **static** data on a class-wide basis.

Note that if you want to guard some other resource from simultaneous access by multiple threads, you can do so by forcing access to that resource through **synchronized** methods.

synchronizing the counters

Armed with this new keyword it appears that the solution is at hand: we'll simply make the methods in **TwoCounter synchronized**. The following example is the same as before, with the addition of the new keyword:

```
//: Sharing2.java
// Using the synchronized keyword to prevent
// multiple access to a particular resource.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

class TwoCounter extends Thread {
  TextField
    t1 = new TextField(5),
    t2 = new TextField(5);
  Label l = new Label("count1 == count2");
  int count1 = 0, count2 = 0;
  public synchronized void run() {
    while (true) {
      t1.setText(Integer.toString(count1++));
      t2.setText(Integer.toString(count2++));
      try {
        sleep(500);
      } catch (InterruptedException e){}
    }
  }
```

```
  synchronized void synchTest() {
    Sharing2.incrementAccess();
    if(count1 != count2)
      l.setText("Unsynched");
  }
}

class Watcher extends Thread {
  Sharing2 p;
  Watcher(Sharing2 p) {
    this.p = p;
    start();
  }
  public void run() {
    while(true) {
      for(int i = 0; i < p.s.length; i++)
        p.s[i].synchTest();
      try {
        sleep(500);
      } catch (InterruptedException e){}
    }
  }
}

public class Sharing2 extends Applet {
  static int accessCount = 0;
  static TextField aCount = new TextField("0", 10);
  static void incrementAccess() {
    accessCount++;
    aCount.setText(Integer.toString(accessCount));
  }
  Button
    start = new Button("Start"),
    observer = new Button("Observe");
  TwoCounter s[];
  boolean isApplet = true;
  int numCounters = 0;
  int numObservers = 0;
  public void init() {
    if(isApplet) {
      numCounters =
        Integer.parseInt(getParameter("size"));
      numObservers =
        Integer.parseInt(
          getParameter("observers"));
    }
    s = new TwoCounter[numCounters];
    for(int i = 0; i < s.length; i++) {
      s[i] = new TwoCounter();
      Panel p = new Panel();
      p.add(s[i].t1);
      p.add(s[i].t2);
      p.add(s[i].l);
      add(p);
    }
    Panel p = new Panel();
    start.addActionListener(new StartL());
    p.add(start);
    observer.addActionListener(new ObserverL());
    p.add(observer);
```

```
    p.add(new Label("Access Count"));
    p.add(aCount);
    add(p);
  }
  class StartL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      for(int i = 0; i < s.length; i++)
        s[i].start();
    }
  }
  class ObserverL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      for(int i = 0; i < numObservers; i++)
        new Watcher(Sharing2.this);
    }
  }
  static class WL extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
      System.exit(0);
    }
  }
  public static void main(String args[]) {
    Sharing2 applet = new Sharing2();
    // This isn't an applet, so set the flag and
    // produce the parameter values from args[]:
    applet.isApplet = false;
    applet.numCounters =
      (args.length == 0 ? 5 :
        Integer.parseInt(args[0]));
    applet.numObservers =
      (args.length < 2 ? 5 :
        Integer.parseInt(args[1]));
    Frame aFrame = new Frame("Sharing2");
    aFrame.addWindowListener(new WL());
    aFrame.add(applet, BorderLayout.CENTER);
    aFrame.setSize(350, applet.numCounters*100);
    applet.init();
    applet.start();
    aFrame.setVisible(true);
  }
} ///:~
```

You'll notice that *both* **run( )** and **synchTest( )** are **synchronized**. If you only synchronize one of the methods, then the other is free to ignore the object lock and can be run with impunity. This is an important point: every method that accesses a critical shared resource must be **synchronized**, or it won't work right.

Now a new issue arises. The **Watcher2** can never get a peek at what's going on because the entire **run( )** method has been **synchronized**, and since **run( )** is always running for each object the lock is always tied up and **synchTest( )** can never be called. You can see this because the **accessCount** never changes.

What we'd like for this example is a way to isolate only *part* of the code inside **run( )**. The section of code you want to isolate this way is called a *critical section* and you use the **synchronized** keyword in a different way to set up a critical section. Java supports critical sections with the *synchronized block;* this time **synchronized** is used to specify the object whose lock is being used to synchronize the enclosed code:

```
synchronized(syncObject) {
  // This code can only be accessed by
  // one thread at a time, assuming all
```

```
    // threads respect syncObject's lock
  }
```

Before the synchronized block can be entered, the lock must be acquired on **syncObject**. If some other thread already has this lock, then the block cannot be entered until the lock is given up.

The **Sharing2** example can be modified by removing the **synchronized** keyword from the entire **run( )** method and instead putting a **synchronized** block around the two critical lines. But what object should be used as the lock? The one that is already respected by **synchTest( )**, which is the current object (**this**)! So the modified **run( )** looks like this:

```
public void run() {
  while (true) {
    synchronized(this) {
      t1.setText(Integer.toString(count1++));
      t2.setText(Integer.toString(count2++));
    }
    try {
      sleep(500);
    } catch (InterruptedException e){}
  }
}
```

This is the only change that must be made to **Sharing2.java**, and you'll see that while the two counters are never out of synch (according to when the **Watcher** is allowed to look at them) there is still adequate access provided to the **Watcher** during the execution of **run( )**.

Of course, all synchronization depends on programmer diligence: every piece of code that may access a shared resource must be wrapped in an appropriate synchronized block.

synchronized efficiency

Since having two methods write to the same piece of data *never* sounds like a particularly good idea, it might seem to make sense for all methods to be automatically **synchronized** and eliminate the **synchronized** keyword altogether (of course, the example with a **synchronized run( )** shows that this wouldn't work either). But it turns out that acquiring a lock is not a cheap process – it multiplies the cost of a method call (that is, entering and exiting from the method, not executing the body of the method) by a minimum of four times, and may be more depending on your implementation. Thus if you know that a particular method will not cause contention problems it is expedient to leave off the **synchronized** keyword.

JavaBeans revisited

Now that you understand synchronization you can take another look at JavaBeans. Whenever you create a Bean, you must assume that it will run in a multithreaded environment. This means that:

1. Whenever possible, all the public methods of a Bean should be **synchronized**. Of course this incurs the **synchronized** runtime overhead. If that's a problem, methods that will not cause problems in critical sections may be left un-**synchronized**, but keep in mind this is not always obvious. Methods that qualify tend to be very small (such as **getCircleSize( )** in the following example) and/or "atomic," that is, the method call executes in such a short amount of code that the object cannot be changed during execution. Making such methods un-**synchronized** may not have a significant effect on the execution speed of your program. You might as well make all **public** methods of a Bean **synchronized** and only remove the **synchronized** keyword when you know for sure that it's necessary and that it makes a difference.

2. When firing a multicast event to a bunch of listeners interested in that event, you must assume that listeners may be added or removed while moving through the list.

The first point is fairly easy to deal with, but the second point requires a little more thought. Consider the **BangBean.java** example presented in the last chapter. That ducked out of the multithreading

question by ignoring the **synchronized** keyword (which hadn't been introduced yet) and making the event unicast. Here's that example modified to work in a multithreaded environment and to use multicasting for events:

```
//: BangBean2.java
// You should write your Beans this way so they
// can run in a multithreaded environment.
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;

public class BangBean2 extends Canvas
    implements Serializable {
  private int xm, ym;
  private int cSize = 20; // Circle size
  private String text = "Bang!";
  private int fontSize = 48;
  private Color tColor = Color.red;
  private Vector actionListeners = new Vector();
  public BangBean2() {
    addMouseListener(new ML());
    addMouseMotionListener(new MM());
  }
  public synchronized int getCircleSize() {
    return cSize;
  }
  public synchronized void
      setCircleSize(int newSize) {
    cSize = newSize;
  }
  public synchronized String getBangText() {
    return text;
  }
  public synchronized void
      setBangText(String newText) {
    text = newText;
  }
  public synchronized int getFontSize() {
    return fontSize;
  }
  public synchronized void
      setFontSize(int newSize) {
    fontSize = newSize;
  }
  public synchronized Color getTextColor() {
    return tColor;
  }
  public synchronized void
      setTextColor(Color newColor) {
    tColor = newColor;
  }
  public void paint(Graphics g) {
    g.setColor(Color.black);
    g.drawOval(xm - cSize/2, ym - cSize/2,
      cSize, cSize);
  }
  // This is a multicast listener, which is
  // more typically used than the unicast
  // approach taken in BangBean.java:
```

```
  public synchronized void addActionListener (
      ActionListener l) {
    actionListeners.addElement(l);
  }
  public synchronized void removeActionListener(
      ActionListener l) {
    actionListeners.removeElement(l);
  }
  // Notice this isn't synchronized:
  public void notifyListeners() {
    ActionEvent a =
      new ActionEvent(BangBean2.this,
        ActionEvent.ACTION_PERFORMED, null);
    Vector lv = null;
    // Make a copy of the vector in case someone
    // adds a listener while we're
    // calling listeners:
    synchronized(this) {
      lv = (Vector)actionListeners.clone();
    }
    // Call all the listener methods:
    for(int i = 0; i < lv.size(); i++) {
      ActionListener al =
        (ActionListener)lv.elementAt(i);
      al.actionPerformed(a);
    }
  }
  class ML extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
      Graphics g = getGraphics();
      g.setColor(tColor);
      g.setFont(
        new Font(
          "TimesRoman", Font.BOLD, fontSize));
      int width =
        g.getFontMetrics().stringWidth(text);
      g.drawString(text,
        (getSize().width - width) /2,
        getSize().height/2);
      g.dispose();
      notifyListeners();
    }
  }
  class MM extends MouseMotionAdapter {
    public void mouseMoved(MouseEvent e) {
      xm = e.getX();
      ym = e.getY();
      repaint();
    }
  }
}

class TestBangBean2 extends Frame {
  BangBean2 bb = new BangBean2();
  public TestBangBean2() {
    setTitle("BangBean2 Test");
    addWindowListener(new WL());
    setSize(300,300);
    add(bb, BorderLayout.CENTER);
    bb.addActionListener(new BBL1());
```

```
    bb.addActionListener(new BBL2());
  }
  class WL extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
      System.exit(0);
    }
  }
  class BBL1 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      System.out.println("BangBean2 action");
    }
  }
  class BBL2 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      System.out.println("More action");
    }
  }
  public static void main(String args[]) {
    Frame f = new TestBangBean2();
    f.setVisible(true);
  }
} ///:~
```

Adding **synchronized** to the methods is an easy change. However, notice in **addActionListener( )** and **removeActionListener( )** that the **ActionListener**s are now added to and removed from a **Vector**, so you can have as many as you want.

You can see that the method **notifyListeners( )** is *not* **synchronized**. This means that it can be called from more than one thread at a time. It also means that it's possible for **addActionListener( )** or **removeActionListener( )** to be called in the middle of a call to **notifyListeners( )**, which is a problem since it traverses the **Vector actionListeners**. To alleviate the problem, the **Vector** is cloned inside a **synchronized** clause, and the clone is traversed. This way the original **Vector** can be manipulated without impact on **notifyListeners( )**.

The **paint( )** method is also not **synchronized**. Deciding whether to synchronize overridden methods is not so clear as when you're just adding your own methods. In this example it turns out that **paint( )** seems to work OK whether it's **synchronized** or not. But the issues you must consider are:

1. Does the method modify the state of "critical" variables within the object. To discover whether the variables are "critical" you must determine whether they will be read or set by other threads in the program (in this case, the reading or setting is virtually always accomplished via **synchronized** methods, so you can just examine those). In the case of **paint( )**, no modification takes place.

2. Does the method depend upon the state of these "critical" variables. If a **synchronized** method modifies a variable that your method uses, then you may very well want to make your method **synchronized** as well. Based on this, you may observe that **cSize** is changed by **synchronized** methods and therefore **paint( )** should be **synchronized**. Here, however, you can ask "what's the worst thing that will happen if **cSize** is changed during a **paint( )**?" When you see that it's nothing too bad, and a transient effect at that, it's best to leave **paint( )** un-**synchronized** to prevent the extra overhead from the **synchronized** method call.

3. A third clue is to notice whether the base-class version of **paint( )** is **synchronized**, which it isn't. This isn't an airtight argument, just a clue. In this case, for example, a field that *is* changed via **synchronized** methods (that is **cSize**) has been mixed into the **paint( )** formula and may have changed the situation.

The test code in **TestBangBean2** has been modified from that in the previous chapter to demonstrate the multicast ability of **BangBean2** by adding an extra listener.

blocking

A thread can be in any of four states:

1. *New*: the thread object has been created but it hasn't been started yet so it cannot run.
2. *Runnable*: This means that a thread *can* be run, when the time-slicing mechanism has CPU cycles available for the thread. Thus the thread may or may not be running, but there's nothing to prevent it from being run if the scheduler can arrange it: that is, it's not dead or blocked.
3. *Dead*: the normal way for a thread to die is by returning from it's **run( )** method. You can also call **stop( )**, but this throws an exception that's a subclass of **Error** (which means you normally don't catch it). Remember that throwing an exception should be a special event and not part of normal program execution; thus the use of **stop( )** is discouraged. There's also a **destroy( )** method that you should never call if you can avoid it, since it's drastic and doesn't release object locks.
4. *Blocked*: the thread could be run but there's something that prevents it. While a thread is in the blocked state the scheduler will simply skip over it and not give it any CPU time. Until a thread re-enters the runnable state it won't perform any operations.

becoming blocked

The blocked state is the most interesting and is worth further examination. A thread can become blocked for five reasons:

1. You've put the thread to sleep by calling **sleep(milliseconds)** in which case it will not be run for the specified time.
2. You've suspended the execution of the thread with **suspend( )**. It will not become runnable again until the thread gets the **resume( )** message.
3. You've suspended the execution of the thread with **wait( )**. It will not become runnable again until the thread gets the **notify( )** or **notifyAll( )** message (yes, this looks just like number 2, but there's a distinct difference that will be revealed).
4. The thread is waiting for some IO to complete.
5. The thread is trying to call a **synchronized** method on another object and that object's lock is not available.

You can also call **yield( )** (a method of the **Thread** class) to voluntarily give up the CPU so other threads may run. However, this is no different than if the scheduler decides that your thread has had enough time and jumps to another thread. That is, nothing prevents the scheduler from re-starting your thread. When a thread is blocked, there's some reason that it cannot continue running.

The following example shows all five ways of becoming blocked. It all exists in a single file called **Blocking.java** but it will be examined here in discrete pieces (you'll note the "Continued" and "Continuing" tags that allow the tool shown in Chapter 17 to piece everything together). First, the basic framework:

```
//: Blocking.java
// Demonstrates the various ways a thread
// can be blocked.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.io.*;

////////////// The basic framework ///////////
```

```
class Blockable extends Thread {
  TextField state = new TextField(40);
  Peeker peeker = new Peeker(this);
  protected int i;
  public synchronized int read() { return i; }
  protected synchronized void update() {
    state.setText(getClass().getName()
      + " state: i = " + i);
  }
}

class Peeker extends Thread {
  Blockable b;
  int session;
  TextField status = new TextField(40);
  Peeker(Blockable b) {
    this.b = b;
    start();
  }
  public void run() {
    while (true) {
      status.setText(b.getClass().getName()
        + " Peeker " + (++session)
        + "; value = " + b.read());
       try {
        sleep(100);
      } catch (InterruptedException e){}
    }
  }
} ///:Continued
```

The **Blockable** class is meant to be a base class for all the classes in this example that demonstrate blocking. A **Blockable** object contains a **TextField** called **state** which is used to display information about the object. The method that displays this information is **update( )**. You can see it uses **getClass( ).getName( )** to produce the name of the class instead of just printing it out; this is because **update( )** cannot know the exact name of the class it is called for, since it will be a class derived from **Blockable**.

The indicator of change in **Blockable** is an **int i,** which will be incremented by the **run( )** method of the derived class.

There's a thread of class **Peeker** that is started for each **Blockable** object, and the **Peeker**'s job is to watch it's associated **Blockable** object to see changes in **i** by calling **read( )** and reporting them in its **status TextField**. This is important: notice that **read( )** and **update( )** are both **synchronized**, which means they require that the object lock be free.

sleeping

The first test in this program is with **sleep( )**

```
///:Continuing
////////////// Blocking via sleep() ///////////
class Sleeper1 extends Blockable {
  public synchronized void run() {
    while(true) {
      i++;
      update();
       try {
        sleep(1000);
      } catch (InterruptedException e){}
    }
  }
```

```
}

class Sleeper2 extends Blockable {
  public void run() {
    while(true) {
      change();
       try {
        sleep(1000);
      } catch (InterruptedException e){}
    }
  }
  synchronized void change() {
      i++;
      update();
  }
} ///:Continued
```

In **Sleeper1** the entire **run( )** method is synchronized. You'll see that the **Peeker** associated with this object will run along merrily *until* you start the thread, and then the **Peeker** stops cold. This is one form of blocking: since **Sleeper1.run( )** is **synchronized**, and once the thread starts it's always inside **run( )**, the method never gives up the object lock and the **Peeker** is blocked.

Sleeper2 provides a solution by making run un-**synchronized**. Only the **change( )** method is **synchronized**, which means that while **run( )** is in **sleep( )**, the **Peeker** may access the **synchronized** method it needs, namely **read( )**. Here you'll see that the **Peeker** continues running when you start the **Sleeper2** thread.

suspending and resuming

The next part of the example introduces the concept of suspension. The **Thread** class has a method **suspend( )** to temporarily halt the thread and **resume( )** that re-starts it at the point it was halted. Presumably, **resume( )** is called by some thread outside the suspended one, and in this case there's a separate class called **Resumer** which does just that. Each of the classes demonstrating suspend/resume has an associated resumer:

```
///:Continuing
////////////// Blocking via suspend() ////////////
class SuspendResume extends Blockable {
  SuspendResume() { new Resumer(this); }
}

class SuspendResume1 extends SuspendResume {
  public synchronized void run() {
    while(true) {
      i++;
      update();
      suspend();
    }
  }
}

class SuspendResume2 extends SuspendResume {
  public void run() {
    while(true) {
      change();
      suspend();
    }
  }
  synchronized void change() {
      i++;
      update();
```

```
    }
  }

  class Resumer extends Thread {
    SuspendResume sr;
    Resumer(SuspendResume sr) {
      this.sr = sr;
      start();
    }
    public void run() {
      while(true) {
         try {
          sleep(1000);
        } catch (InterruptedException e){}
        sr.resume();
      }
    }
  } ///:Continued
```

SuspendResume1 also has a **synchronized run( )** method. Again, when you start this thread you'll see that its associated **Peeker** gets blocked waiting for the lock to become available, which never happens. This is fixed as before in **SuspendResume2**, which does not **synchronize** the entire **run( )** method, but instead uses a separate **synchronized change( )** method.

wait and notify

The point with the first two examples is that both **sleep( )** and **suspend( )** *do not* release the lock as they are called. You must be aware of this when working with locks. On the other hand, the method **wait( )** *does* release the lock when it is called, which means that other **synchronized** methods in the thread object may be called during a **wait( )**. In the following two classes, you'll see that the **run( )** method is fully **synchronized** in both cases, however the **Peeker** still has full access to the **synchronized** methods during a **wait( )**. This is because **wait( )** releases the lock on the object as it suspends the method it's called within.

You'll also see that there are two forms of **wait( ).** The first takes an argument in milliseconds that has the same meaning as in **sleep( )**: pause for this period of time. The difference is that in **wait( )**, the object lock is released *and* you can come out of the **wait( )** because of a **notify( )** as well as having the clock run out.

The second form takes no arguments, and means that the **wait( )** will continue until a **notify( )** comes along, and will not automatically terminate after a time.

One fairly unique aspect of **wait( )** and **notify( )** is both methods are part of the base class **Object** and not part of **Thread** as are **sleep( )**, **suspend( )** and **resume( )**. Although this seems a bit strange at first – to have something that's exclusively for threading as part of the universal base class – it's essential because they manipulate the lock that's also part of every object. As a result, you can put a **wait( )** inside any **synchronized** method, regardless of whether there's any threading going on inside that particular class. In fact, the *only* place you can call **wait( )** is within a **synchronized** method or block. If you call **wait( )** or **notify( )** within a method that's not **synchronized** the program will compile, but when you run it you'll get a **IllegalMonitorStateException** with the somewhat non-intuitive message "current thread not owner." Note that **sleep( )**, **suspend( )** and **resume( )** can all be called within non-**synchronized** methods since they don't manipulate the lock.

You can only call **wait( )** or **notify( )** for your own lock. Again, you can compile code that tries to use the wrong lock but it will produce the same **IllegalMonitorStateException** message as before. You can't fool with someone else's lock, but you can ask another object to perform an operation that manipulates its own lock. So one approach is to create a **synchronized** method that calls **notify( )** for its own object. However, in **Notifier** you'll see the **notify( )** call inside a **synchronized** block:

```
synchronized(wn2) {
  wn2.notify();
}
```

where **wn2** is the object of type **WaitNotify2**. This means that this method, which is not part of **WaitNotify2**, acquires the lock on the **wn2** object at which point it's legal for it to call **notify( )** for **wn2** and you won't get the **IllegalMonitorStateException**.

```
///:Continuing
////////////// Blocking via wait() ////////////
class WaitNotify1 extends Blockable {
  public synchronized void run() {
    while(true) {
      i++;
      update();
       try {
        wait(1000);
      } catch (InterruptedException e){}
    }
  }
}

class WaitNotify2 extends Blockable {
  WaitNotify2() { new Notifier(this); }
  public synchronized void run() {
    while(true) {
      i++;
      update();
       try {
        wait();
      } catch (InterruptedException e){}
    }
  }
}

class Notifier extends Thread {
  WaitNotify2 wn2;
  Notifier(WaitNotify2 WN2) {
    wn2 = WN2;
    start();
  }
  public void run() {
    while(true) {
       try {
        sleep(2000);
      } catch (InterruptedException e){}
      synchronized(wn2) {
        wn2.notify();
      }
    }
  }
} ///:Continued
```

wait( ) is typically used when you've gotten to the point where you're waiting for some other condition, under the control of forces outside your thread, to change and you don't want to just idly wait by inside the thread. So **wait( )** allows you to put the thread to sleep while waiting for the world to change, and only when a **notify( )** occurs does the thread wake up and check for changes. Thus it provides a way to synchronize between threads.

blocking on IO

If a stream is waiting for some IO activity, it will automatically block. In the following portion of the example the two classes work with generic **Reader** and **Writer** objects (using the new Java 1.1 Streams), but in the test framework a piped stream will be set up to allow the two threads to safely pass data to each other (which is the purpose of piped streams).

The **Sender** puts data into the **Writer** and sleeps for a random amount of time. However, **Receiver** has no **sleep( )** or **suspend( )** or **wait( )**. But when it does a **read( )** it automatically blocks when there is no more data.

```
///:Continuing
class Sender extends Blockable { // send
  Writer out;
  Sender(Writer out) { this.out = out; }
  public void run() {
    while(true) {
      for(char c = 'A'; c <= 'z'; c++) {
         try {
          i++;
          out.write(c);
          state.setText("Sender sent: "
            + (char)c);
          sleep((int)(3000 * Math.random()));
        } catch (InterruptedException e){}
          catch (IOException e) {}
      }
    }
  }
}

class Receiver extends Blockable {
  Reader in;
  Receiver(Reader in) { this.in = in; }
  public void run() {
    try {
      while(true) {
        i++; // show peeker it's alive
        // Blocks until characters are there:
        state.setText("Receiver read: "
          + (char)in.read());
      }
    } catch(IOException e) { e.printStackTrace(); }
  }
} ///:Continued
```

Both classes also put information into their **state** fields and change **i** so the **Peeker** can see that the thread is running.

testing

The main applet class is surprisingly simple because most of the work has been put into the **Blockable** framework. Basically, an array of **Blockable** objects is created, and since each one is a thread they perform their own activities when you press the "start" button. There's also a button and **actionPerformed( )** clause to stop all the **Peeker** objects, which provides a demonstration of the **stop( )** method of **Thread**.

To set up a connection between the **Sender** and **Receiver** objects, a **PipedWriter** and **PipedReader** are created. Notice that the **PipedReader in** must be connected to the **PipedWriter out** via a constructor argument. After that, anything that's placed in **out** can later be extracted from **in**, as if it passed through a pipe (thus the name). The **in** and **out** objects are then passed to the **Receiver** and **Sender** constructors, respectively, which treat them as **Reader** and **Writer** objects of any type (that is, they are upcast).

The array of **Blockable** handles **b[]** is not initialized at it's point of definition because the piped streams cannot be set up before that definition takes place (the need for the **try** block prevents this). And inside **init( )**, you cannot simply assign **b** to an aggregate initialization statement. So instead **b2[]** is created to allow for the aggregate initialization, and then it is assigned directly to **b**.

```
///:Continuing
////////// Testing Everything //////////
public class Blocking extends Applet {
  Button
    start = new Button("Start"),
    stopPeekers = new Button("Stop Peekers");
  boolean started = false;
  Blockable b[];
  PipedWriter out;
  PipedReader in;
  public void init() {
     out = new PipedWriter();
    try {
      in = new PipedReader(out);
    } catch(IOException e) {}
    Blockable b2[] = {
      new Sleeper1(),
      new Sleeper2(),
      new SuspendResume1(),
      new SuspendResume2(),
      new WaitNotify1(),
      new WaitNotify2(),
      new Sender(out),
      new Receiver(in)
    };
    b = b2;
    for(int i = 0; i < b.length; i++) {
      add(b[i].state);
      add(b[i].peeker.status);
    }
    start.addActionListener(new StartL());
    add(start);
    stopPeekers.addActionListener(
      new StopPeekersL());
    add(stopPeekers);
  }
  class StartL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if(!started) {
        started = true;
        for(int i = 0; i < b.length; i++)
          b[i].start();
      }
    }
  }
  class StopPeekersL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      // Demonstration of Thread.stop():
      for(int i = 0; i < b.length; i++)
        b[i].peeker.stop();
    }
  }
  static class WL extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
      System.exit(0);
    }
  }
  public static void main(String args[]) {
    Blocking applet = new Blocking();
    Frame aFrame = new Frame("Blocking");
```

```
    aFrame.addWindowListener(new WL());
    aFrame.add(applet, BorderLayout.CENTER);
    aFrame.setSize(350,550);
    applet.init();
    applet.start();
    aFrame.setVisible(true);
  }
} ///:~
```

In **init( )**, note the loop that moves through the entire array and adds the **state** and **peeker.status** text fields to the page.

When the **Blockable** threads are initially created, each one automatically creates and starts its own **Peeker**. Thus you'll see the **Peeker**s running before the **Blockable** threads are started. This is essential, as some of the **Peeker**s will get blocked and stop when the **Blockable** threads start, and it's essential to see this to understand that particular aspect of blocking.

deadlock

Because threads can become blocked *and* because objects can have **synchronized** methods that prevent threads from accessing that object until the synchronization lock is released, it's possible for one thread to get stuck waiting for another thread, which in turn waits for another thread, etc. until the chain leads back to a thread waiting on the first one. Thus there's a continuous loop of threads waiting on each other and no one can move. This is called *deadlock*. The claim is that it doesn't happen that often, but when it happens to you it's very frustrating to debug.

There is no language support to help prevent deadlock; it's up to you to avoid it by careful design. These are not very comforting words to the person who's trying to debug a deadlocking program.

priorities

The *priority* of a thread tells the scheduler how important this thread is. If there are a number of threads blocked and waiting to be run, the scheduler will run the one with the highest priority first. However, this doesn't mean that threads with lower priority don't get run (that is, you can't get deadlocked because of priorities). Lower priority threads just tend to run less often.

You can read the priority of a thread with **getPriority( )** and change it with **setPriority( )**. The form of the prior "counter" examples can be used to show the effect of changing the priorities. In this applet you'll see that the counters slow down as the associated threads have their priorities lowered:

```
//: Counter5.java
// Adjusting the priorities of threads
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

class Ticker2 extends Thread {
  Button
    b = new Button("Toggle"),
    incPriority = new Button("up"),
    decPriority = new Button("down");
  TextField
    t = new TextField(10),
    p = new TextField(3); // Display priority
  int count = 0;
  boolean runFlag = true;
  Ticker2() {
    b.addActionListener(new ToggleL());
    incPriority.addActionListener(new UpL());
```

```
    decPriority.addActionListener(new DownL());
  }
  class ToggleL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      runFlag = !runFlag;
    }
  }
  class UpL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      int newPriority = getPriority() + 1;
      if(newPriority > Thread.MAX_PRIORITY)
        newPriority = Thread.MAX_PRIORITY;
      setPriority(newPriority);
    }
  }
  class DownL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      int newPriority = getPriority() - 1;
      if(newPriority < Thread.MIN_PRIORITY)
        newPriority = Thread.MIN_PRIORITY;
      setPriority(newPriority);
    }
  }
  public void run() {
    while (true) {
      if(runFlag) {
        t.setText(Integer.toString(count++));
        p.setText(Integer.toString(getPriority()));
      }
      yield();
    }
  }
}

public class Counter5 extends Applet {
  Button
    start = new Button("Start"),
    upMax = new Button("Inc Max Priority"),
    downMax = new Button("Dec Max Priority");
  boolean started = false;
  static final int size = 10;
  Ticker2 s[] = new Ticker2[size];
  TextField mp = new TextField(3);
  public void init() {
    for(int i = 0; i < s.length; i++) {
      s[i] = new Ticker2();
      Panel p = new Panel();
      p.add(s[i].t);
      p.add(s[i].p);
      p.add(s[i].b);
      p.add(s[i].incPriority);
      p.add(s[i].decPriority);
      add(p);
    }
    add(new Label("MAX_PRIORITY = "
      + Thread.MAX_PRIORITY));
    add(new Label("MIN_PRIORITY = "
      + Thread.MIN_PRIORITY));
    add(new Label("Thread Group Max Priority = "));
    add(mp);
```

```
    add(start);
    add(upMax); add(downMax);
    start.addActionListener(new StartL());
    upMax.addActionListener(new UpMaxL());
    downMax.addActionListener(new DownMaxL());
    showMaxPriority();
    // Recursively display parent thread groups:
    ThreadGroup parent =
      s[0].getThreadGroup().getParent();
    while(parent != null) {
      add(new Label(
        "Parent threadgroup max priority = "
        + parent.getMaxPriority()));
      parent = parent.getParent();
    }
  }
  void showMaxPriority() {
    mp.setText(Integer.toString(
      s[0].getThreadGroup().getMaxPriority()));
  }
  class StartL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if(!started) {
        started = true;
        for(int i = 0; i < s.length; i++)
          s[i].start();
      }
    }
  }
  class UpMaxL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      int maxp =
        s[0].getThreadGroup().getMaxPriority();
      if(++maxp > Thread.MAX_PRIORITY)
        maxp = Thread.MAX_PRIORITY;
      s[0].getThreadGroup().setMaxPriority(maxp);
      showMaxPriority();
    }
  }
  class DownMaxL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      int maxp =
        s[0].getThreadGroup().getMaxPriority();
      if(--maxp < Thread.MIN_PRIORITY)
        maxp = Thread.MIN_PRIORITY;
      s[0].getThreadGroup().setMaxPriority(maxp);
      showMaxPriority();
    }
  }
  static class WL extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
      System.exit(0);
    }
  }
  public static void main(String args[]) {
    Counter5 applet = new Counter5();
    Frame aFrame = new Frame("Counter5");
    aFrame.addWindowListener(new WL());
    aFrame.add(applet, BorderLayout.CENTER);
    aFrame.setSize(300, 600);
```

```
    applet.init();
    applet.start();
    aFrame.setVisible(true);
  }
} ///:~
```

The **Ticker2** follows the form established earlier in this chapter, but there's an extra **TextField** for displaying the priority of the thread and two more buttons, for incrementing and decrementing the priority.

Also notice the use of **yield( )**, which voluntarily hands control back to the scheduler. Without this the multithreading mechanism still works, but you'll notice it runs very slowly (try removing the call to **yield( )**!). You could also call **sleep( )**, but then the rate of counting would be controlled by the **sleep( )** duration instead of the priority.

The **init( )** in **Counter5** creates an array of 10 **Ticker2**s and places their buttons and fields on the form, along with buttons to start up the whole process as well as increment and decrement the maximum priority of the threadgroup. In addition, there are labels that display the maximum and minimum priorities possible for a thread, and a **TextField** to show the thread group's maximum priority (the next section will fully describe thread groups). Finally, the priorities of the parent thread groups are also displayed as labels.

When you press an "up" or "down" button, that **Ticker2**'s priority is fetched and incremented or decremented, accordingly.

When you run this program, you'll notice several things. First of all, the thread group's default priority is 5. Even if you decrement the maximum priority below 5 before starting the threads (or before creating the threads, which requires a code change) each thread will have a default priority of 5.

The simple test is to take one counter and decrement its priority to one, and observe that it counts much slower. But now try to increment it again. You can get it back up to the thread group's priority, but no higher. Now decrement the thread group's priority a couple of times. The thread priorities are unchanged, but if you try to modify them either up or down you'll see that they'll automatically pop to the priority of the thread group. Also, new threads will still be given a default priority, even if that's higher than the group priority (thus the group priority is not a way to prevent new threads from having higher priorities than existing ones).

Finally, try to increment the group maximum priority. It can't be done. You can only reduce thread group maximum priorities, not increase them.

thread groups

All threads belong to a thread group. This may be either the default thread group or a group you explicitly specify when you create the thread. At creation, the thread is bound to a group and cannot change to a different group. Each application has at least one thread that belongs to the system thread group. If you create more threads without specifying a group, they will also belong to the system thread group.

Thread groups must also belong to other thread groups. The thread group that a new one belongs to must be specified in the constructor. If you create a thread group without specifying a thread group for it to belong to, it will be placed under the system thread group. Thus, all thread groups in your application will ultimately have the system thread group as the parent.

The reason for the existence of thread groups is hard to determine from the literature, which tends to be confusing on this subject. It's often cited as "security reasons." According to Arnold & Gosling[1] "Threads within a thread group can modify the other threads in the group, including any farther down the hierarchy. A thread cannot modify threads outside of its own group or contained groups." It's hard to know what "modify" is supposed to mean here. The following example shows a thread in a "leaf"

[1] *The Java Programming Language*, by Ken Arnold and James Gosling, Addison-Wesley 1996 pp 179.

subgroup modifying the priorities of all the threads in its tree of thread groups, as well as calling a method for all the threads in its tree.

```
//: TestAccess.java
// How threads can access other threads
// in a parent thread group

public class TestAccess {
  public static void main(String args[]) {
    ThreadGroup
      x = new ThreadGroup("x"),
      y = new ThreadGroup(x, "y"),
      z = new ThreadGroup(y, "z");
    Thread
      one = new TestThread1(x, "one"),
      two = new TestThread2(z, "two");
  }
}

class TestThread1 extends Thread {
  private int i;
  TestThread1(ThreadGroup g, String name) {
    super(g, name);
  }
  void f() {
    i++; // modify this thread
    System.out.println(getName() + " f()");
  }
}

class TestThread2 extends TestThread1 {
  TestThread2(ThreadGroup g, String name) {
    super(g, name);
    start();
  }
  public void run() {
    ThreadGroup g =
      getThreadGroup().getParent().getParent();
    g.list();
    Thread gAll[] = new Thread[g.activeCount()];
    g.enumerate(gAll);
    for(int i = 0; i < gAll.length; i++) {
      gAll[i].setPriority(Thread.MIN_PRIORITY);
      ((TestThread1)gAll[i]).f();
    }
    g.list();
  }
} ///:~
```

In **main( )**, several **ThreadGroup**s are created, leafing off from each other: **x** has no argument but its name (a **String**) and thus it is automatically placed in the "system" thread group, while **y** is under **x** and **z** is under **y**. Notice that initialization happens in textual order so this code is legal.

Two threads are created and placed in different thread groups. **TestThread1** doesn't have a **run( )** method, but it does have an **f( )** that modifies the thread and prints something so you can see it was indeed called. **TestThread2** is a subclass of **TestThread1** and it's **run( )** is fairly elaborate: it first gets the thread group of the current thread, and then moves up the heritage tree by two levels using **getParent( )** (this is contrived since I purposely place the **TestThread2** object two levels down in the hierarchy). At this point, an array of handles to **Thread**s is created, using the method **activeCount( )** to ask how many threads are in this thread group and all the child thread groups. The **enumerate( )** method places handles to all these threads in the array **gAll**, and then I simply move through the entire

array calling the **f( )** method for each thread, as well as modifying the priority. Thus, a thread in a "leaf" thread group modifies threads in parent thread groups.

The debugging method **list( )** prints all the information about a thread group to standard output, and is very helpful when investigating thread group behavior. Here's the output of the program:

```
java.lang.ThreadGroup[name=x,maxpri=10]
    Thread[one,5,x]
    java.lang.ThreadGroup[name=y,maxpri=10]
        java.lang.ThreadGroup[name=z,maxpri=10]
            Thread[two,5,z]
one f()
two f()
java.lang.ThreadGroup[name=x,maxpri=10]
    Thread[one,1,x]
    java.lang.ThreadGroup[name=y,maxpri=10]
        java.lang.ThreadGroup[name=z,maxpri=10]
            Thread[two,1,z]
```

Not only does **list( )** print the class name of **ThreadGroup** or **Thread**, but it also prints the thread group name and its maximum priority. For threads, the thread name is printed, followed by the thread priority and the group it belongs to. Note that **list( )** indents the threads and thread groups to indicate that they are children of the un-indented thread group.

You can see that **f( )** is called by the **TestThread2 run( )** method, so it's obvious that all threads in a group are vulnerable. However, you can only access threads that branch off from your own **system** thread group tree, and perhaps this is what is meant by "safety." You cannot access anyone else's system thread group tree.

controlling thread groups

Putting aside the safety issue, one thing thread groups do seem to be useful for is control: you can perform certain operations on an entire thread group with a single command. The following example demonstrates this and the restrictions on priorities within thread groups. The commented numbers in parentheses provide a reference to compare to the output.

```
//: ThreadGroup1.java
// How thread groups control priorities
// of the threads inside them.

public class ThreadGroup1 {
  public static void main(String[] args) {
    // Get the system thread & print its Info:
    ThreadGroup sys =
      Thread.currentThread().getThreadGroup();
    sys.list(); // (1)
    // Reduce the system thread group priority:
    sys.setMaxPriority(Thread.MAX_PRIORITY - 1);
    // Increase the main thread priority:
    Thread curr = Thread.currentThread();
    curr.setPriority(curr.getPriority() + 1);
    sys.list(); // (2)
    // Attempt to set a new group to the max:
    ThreadGroup g1 = new ThreadGroup("g1");
    g1.setMaxPriority(Thread.MAX_PRIORITY);
    // Attempt to set a new thread to the max:
    Thread t = new Thread(g1, "A");
    t.setPriority(Thread.MAX_PRIORITY);
    g1.list(); // (3)
    // Reduce g1's max priority, then attempt
    // to increase it:
    g1.setMaxPriority(Thread.MAX_PRIORITY - 2);
```

```
    g1.setMaxPriority(Thread.MAX_PRIORITY);
    g1.list(); // (4)
    // Attempt to set a new thread to the max:
    t = new Thread(g1, "B");
    t.setPriority(Thread.MAX_PRIORITY);
    g1.list(); // (5)
    // Lower the max priority below the default
    // thread priority:
    g1.setMaxPriority(Thread.MIN_PRIORITY + 2);
    // Look at a new thread's priority before
    // and after changing it:
    t = new Thread(g1, "C");
    g1.list(); // (6)
    t.setPriority(t.getPriority() -1);
    g1.list(); // (7)
    // Make g2 a child Threadgroup of g1 and
    // try to increase its priority:
    ThreadGroup g2 = new ThreadGroup(g1, "g2");
    g2.list(); // (8)
    g2.setMaxPriority(Thread.MAX_PRIORITY);
    g2.list(); // (9)
    // Add a bunch of new threads to g2:
    for (int i = 0; i < 5; i++)
      new Thread(g2, Integer.toString(i));
    // Show information about all threadgroups
    // and threads:
    sys.list(); // (10)
    System.out.println("Starting all threads:");
    Thread[] all = new Thread[sys.activeCount()];
    sys.enumerate(all);
    for(int i = 0; i < all.length; i++)
      if(!all[i].isAlive())
        all[i].start();
    // Suspends & Stops all threads in
    // this group and its subgroups:
    System.out.println("All threads started");
    sys.suspend();
    // Never gets here...
    System.out.println("All threads suspended");
    sys.stop();
    System.out.println("All threads stopped");
  }
} ///:~
```

The output that follows has been edited to allow it to fit on the page (the **java.lang.** has been removed) and to add numbers to correspond to the commented numbers in the listing above.

```
(1) ThreadGroup[name=system,maxpri=10]
      Thread[main,5,system]
(2) ThreadGroup[name=system,maxpri=9]
      Thread[main,6,system]
(3) ThreadGroup[name=g1,maxpri=9]
      Thread[A,9,g1]
(4) ThreadGroup[name=g1,maxpri=8]
      Thread[A,9,g1]
(5) ThreadGroup[name=g1,maxpri=8]
      Thread[A,9,g1]
      Thread[B,8,g1]
(6) ThreadGroup[name=g1,maxpri=3]
      Thread[A,9,g1]
```

```
      Thread[B,8,g1]
      Thread[C,6,g1]
(7) ThreadGroup[name=g1,maxpri=3]
      Thread[A,9,g1]
      Thread[B,8,g1]
      Thread[C,3,g1]
(8) ThreadGroup[name=g2,maxpri=3]
(9) ThreadGroup[name=g2,maxpri=3]
(10)ThreadGroup[name=system,maxpri=9]
      Thread[main,6,system]
      ThreadGroup[name=g1,maxpri=3]
        Thread[A,9,g1]
        Thread[B,8,g1]
        Thread[C,3,g1]
        ThreadGroup[name=g2,maxpri=3]
          Thread[0,6,g2]
          Thread[1,6,g2]
          Thread[2,6,g2]
          Thread[3,6,g2]
          Thread[4,6,g2]
Starting all threads:
All threads started
```

All programs have at least one thread running, and the first action in **main( )** is to call the **static** method of **Thread** called **currentThread( )**. From this thread, the thread group is produced, and **list( )** is called for the result. The output is:

```
(1) ThreadGroup[name=system,maxpri=10]
      Thread[main,5,system]
```

You can see that the name of the main thread group is **system**, and the name of the main thread is **main**, and it belongs to the **system** thread group.

The second exercise shows that the **system** group's maximum priority can be reduced and the **main** thread can have it's priority increased:

```
(2) ThreadGroup[name=system,maxpri=9]
      Thread[main,6,system]
```

The third exercise creates a new thread group **g1**, which automatically belongs to the **system** thread group since it isn't otherwise specified. A new thread **A** is placed in **g1**. After attempting to set this group's maximum priority to the highest level and **A**'s priority to the highest level, the result is:

```
(3) ThreadGroup[name=g1,maxpri=9]
      Thread[A,9,g1]
```

Thus it's not possible to change the thread group's maximum priority to be higher than it's parent thread group.

The fourth exercise reduces **g1**'s maximum priority by two and then tries to increase it up to **Thread.MAX_PRIORITY**. The result is:

```
(4) ThreadGroup[name=g1,maxpri=8]
      Thread[A,9,g1]
```

You can see that the increase in maximum priority didn't work. You can only decrease a thread group's maximum priority, not increase it. Also, note that thread **A**'s priority didn't change, and now it is higher than the thread group's maximum priority. Changing a thread group's maximum priority doesn't affect existing threads.

The fifth exercise attempts to set a new thread to maximum priority:

```
(5) ThreadGroup[name=g1,maxpri=8]
```

```
      Thread[A,9,g1]
      Thread[B,8,g1]
```

The new thread cannot be changed to anything higher than the maximum thread group priority.

The default thread priority for this program is 6; that's the priority a new thread will be created at and where it will stay if you don't manipulate the priority. Exercise six lowers the maximum thread group priority below the default thread priority to see what happens when you create a new thread under this condition:

```
(6) ThreadGroup[name=g1,maxpri=3]
      Thread[A,9,g1]
      Thread[B,8,g1]
      Thread[C,6,g1]
```

Even though the maximum priority of the thread group is 3, the new thread is still created using the default priority of 6. Thus, maximum thread group priority does not affect default priority (in fact, there appears to be no way to set the default priority for new threads).

After changing the priority, attempting to decrement it by one, the result is:

```
(7) ThreadGroup[name=g1,maxpri=3]
      Thread[A,9,g1]
      Thread[B,8,g1]
      Thread[C,3,g1]
```

Only when you attempt to change the priority is the thread group's maximum priority enforced.

A similar experiment is performed in (8) and (9), when a new thread group **g2** is created as a child of **g1** and it's maximum priority is changed. You can see that it's impossible for **g2**'s maximum to go higher than **g1**'s:

```
(8) ThreadGroup[name=g2,maxpri=3]
(9) ThreadGroup[name=g2,maxpri=3]
```

Also note that **g2** is automatically set to the thread group maximum priority of **g1** as **g2** is created.

After all these experiments, the entire system of thread groups and threads is printed out:

```
(10)ThreadGroup[name=system,maxpri=9]
      Thread[main,6,system]
      ThreadGroup[name=g1,maxpri=3]
        Thread[A,9,g1]
        Thread[B,8,g1]
        Thread[C,3,g1]
        ThreadGroup[name=g2,maxpri=3]
          Thread[0,6,g2]
          Thread[1,6,g2]
          Thread[2,6,g2]
          Thread[3,6,g2]
          Thread[4,6,g2]
```

So because of the rules of thread groups, a child group must always have a maximum priority that's less than or equal to its parent's maximum priority.

The last part of this program demonstrates methods for an entire group of threads. First the program moves through the entire tree of threads and starts each one that hasn't been started. For drama, the **system** group is then suspended and finally stopped. But when you suspend the **system** group you also suspend the **main** thread and the whole program shuts down, so it never gets to the point where the threads are stopped. Actually, if you do stop the **main** thread it throws a **ThreadDeath** exception, so this is not a very typical thing to do. Since **ThreadGroup** is inherited from **Object** which contains the **wait( )** method, you can also choose to suspend the program for any number of seconds by calling **wait(seconds * 1000)**. This must acquire the lock inside a synchronized block, of course.

The **ThreadGroup** class also has **suspend( )** and **resume( )** methods so you can stop and start an entire thread group and all its threads and subgroups with a single command.

Thread groups can seem a bit mysterious at first, but keep in mind you probably won't be using them directly very often.

Runnable revisited

Earlier in this chapter, I suggested that you think carefully before making an applet or main **Frame** as an implementation of **Runnable**, because then you can only make one of those threads in your program. This limits your flexibility if you decide you want to have more than one thread of that type.

Of course, if you must inherit from a class *and* you want to add threading behavior to the class, **Runnable** is the correct solution. The final example in this chapter exploits this by making a **Runnable Canvas** class that paints different colors on itself. This application is set up to take values from the command line to determine how big the grid of colors is and how long to **sleep( )** between color changes, and by playing with these values you'll discover some interesting and possibly inexplicable features of threads:

```
//: ColorBoxes.java
// Using the Runnable interface
import java.awt.*;
import java.awt.event.*;

class CBox extends Canvas implements Runnable {
  Thread t;
  int pause;
  static final Color colors[] = {
    Color.black, Color.blue, Color.cyan,
    Color.darkGray, Color.gray, Color.green,
    Color.lightGray, Color.magenta,
    Color.orange, Color.pink, Color.red,
    Color.white, Color.yellow
  };
  Color cColor = newColor();
  static final Color newColor() {
    return colors[
      (int)(Math.random() * colors.length)
    ];
  }
  public void paint(Graphics  g) {
    g.setColor(cColor);
    Dimension s = getSize();
    g.fillRect(0, 0, s.width, s.height);
  }
  CBox(int pause) {
    this.pause = pause;
    t = new Thread(this);
    t.start();
  }
  public void run() {
    while(true) {
      cColor = newColor();
      repaint();
      try {
        t.sleep(pause);
      } catch(InterruptedException e) {}
    }
  }
```

```
}

public class ColorBoxes extends Frame {
  public ColorBoxes(int pause, int grid) {
    setTitle("ColorBoxes");
    setLayout(new GridLayout(grid, grid));
    for (int i = 0; i < grid * grid; i++)
      add(new CBox(pause));
    addWindowListener(new WL());
  }
  class WL extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
      System.exit(0);
    }
  }
  public static void main(String[] args) {
    int pause = 50;
    int grid = 8;
    if(args.length > 0)
      pause = Integer.parseInt(args[0]);
    if(args.length > 1)
      grid = Integer.parseInt(args[1]);
    Frame f = new ColorBoxes(pause, grid);
    f.setSize(500, 400);
    f.setVisible(true);
  }
} ///:~
```

ColorBoxes is a typical application with a constructor that sets up the GUI. This constructor takes an argument of **int grid** to set up the **GridLayout** so it has **grid** cells in each dimension. Then it adds the appropriate number of **CBox** objects to fill the grid, passing the **pause** value to each one. In **main( )** you can see how **pause** and **grid** have default values that can be changed if you pass in command-line arguments.

CBox is where all the work takes place. This is inherited from **Canvas** and it implements the **Runnable** interface so each **Canvas** can also be a **Thread**. Remember that when you implement **Runnable**, you don't make a **Thread** object, just a class that has a **run( )** method. Thus you must explicitly create a **Thread** object and hand the **Runnable** object to the constructor, then call **start( )** (this happens in the constructor). In **CBox** this thread is called **t**.

Note the array **colors**, which is an enumeration of all the colors in class **Color**. This is used in **newColor( )** to produce a randomly-selected color. The current cell color is **cColor**.

paint( ) is quite simple – it just sets the color to **cColor** and fills the whole canvas with that color.

In **run( )**, you see the infinite loop that sets the **cColor** to a new random color and then calls **repaint( )** to show it. Then the thread goes to **sleep( )** for the amount of time specified on the command line.

Precisely because this design is flexible and threading is tied to each **Canvas** element, you can experiment by making as many as you want (in reality, there is a restriction imposed by the number of threads your JVM can comfortably handle).

This program also makes an interesting benchmark, since it can show dramatic speed differences between one JVM implementation and another.

too many threads

At some point, you'll find that **ColorBoxes** bogs down. On my machine, this happened somewhere after a 10 by 10 grid. Why does this happen? You're naturally suspicious that the AWT might have something to do with it, so here's an example that tests the premise by making fewer threads. The code is reorganized so a **Vector implements Runnable** and that **Vector** holds a number of color

blocks, and randomly chooses ones to update. Then a number of these **Vector** objects are created, roughly depending on the grid dimension you choose. As a result, you have far fewer threads than color blocks, so if there's a speedup we'll know it was because there were too many threads in the previous example:

```
//: ColorBoxes2.java
// Balancing thread use
import java.awt.*;
import java.awt.event.*;
import java.util.*;

class CBox2 extends Canvas {
  static final Color colors[] = {
    Color.black, Color.blue, Color.cyan,
    Color.darkGray, Color.gray, Color.green,
    Color.lightGray, Color.magenta,
    Color.orange, Color.pink, Color.red,
    Color.white, Color.yellow
  };
  Color cColor = newColor();
  static final Color newColor() {
    return colors[
      (int)(Math.random() * colors.length)
    ];
  }
  void nextColor() {
    cColor = newColor();
    repaint();
  }
  public void paint(Graphics  g) {
    g.setColor(cColor);
    Dimension s = getSize();
    g.fillRect(0, 0, s.width, s.height);
  }
}

class CBoxVector
  extends Vector implements Runnable {
  Thread t;
  int pause;
  CBoxVector(int pause) {
    this.pause = pause;
    t = new Thread(this);
  }
  void go() { t.start(); }
  public void run() {
    while(true) {
      int i = (int)(Math.random() * size());
      ((CBox2)elementAt(i)).nextColor();
      try {
        t.sleep(pause);
      } catch(InterruptedException e) {}
    }
  }
}

public class ColorBoxes2 extends Frame {
  CBoxVector v[];
  public ColorBoxes2(int pause, int grid) {
    setTitle("ColorBoxes2");
```

```
    setLayout(new GridLayout(grid, grid));
    v = new CBoxVector[grid];
    for(int i = 0; i < grid; i++)
      v[i] = new CBoxVector(pause);
    for (int i = 0; i < grid * grid; i++) {
      v[i % grid].addElement(new CBox2());
      add((CBox2)v[i % grid].lastElement());
    }
    for(int i = 0; i < grid; i++)
      v[i].go();
    addWindowListener(new WL());
  }
  class WL extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
      System.exit(0);
    }
  }
  public static void main(String[] args) {
    // Shorter default pause than ColorBoxes:
    int pause = 5;
    int grid = 8;
    if(args.length > 0)
      pause = Integer.parseInt(args[0]);
    if(args.length > 1)
      grid = Integer.parseInt(args[1]);
    Frame f = new ColorBoxes2(pause, grid);
    f.setSize(500, 400);
    f.setVisible(true);
  }
} ///:~
```

In **ColorBoxes2** an array of **CBoxVector** is created and initialized to hold **grid CBoxVector**s, each of which knows how long to sleep. An equal number of **Cbox2** objects are then added to each **CBoxVector**, and each vector is told to **go( )**, which starts its thread.

CBox2 is very similar to **CBox**: it paints itself with a randomly-chosen color. But that's *all* a **CBox2** does. All of the threading has been moved into **CBoxVector**.

The **CBoxVector** could also have inherited **Thread** and had a member object of type **Vector**. That design has the advantage that the **addElement( )** and **elementAt( )** methods could then be given specific argument and return value types rather than just generic **Object**s (their names could also be changed to something shorter). However, the design used here seemed at first glance to require less code. In addition it automatically retains all the other behaviors of a **Vector**. With all the casting and parentheses necessary for **elementAt( )**, this might not be the case as your body of code grows.

As before, when you implement **Runnable** you don't get all the equipment that comes with **Thread**, so you actually have to create a new **Thread** and hand yourself to it's constructor in order to have something to **start( )**, as you can see in the **CBoxVector** constructor and in **go( )**. The **run( )** method simply chooses a random element number within the vector and calls **nextColor( )** for that element to cause it to choose a new randomly-selected color.

Upon running this program, you see that it does indeed run faster and respond more quickly (for instance, when you interrupt it, it stops more quickly), and it doesn't seem to bog down as much at higher grid sizes. Thus a new factor is added into the threading equation: you must watch to see that you don't have "too many threads" (whatever that turns out to mean for your particular program and platform). If so, you must try to use techniques like the one above to "balance" the number of threads in your program. Thus, if you see performance problems in a multithreaded program you now have a number of issues to examine:

1. Do you have enough calls to **sleep( )**, **yield( )** and/or **wait( )** ?

2. Are calls to **sleep( )** long enough?

3. Are you running too many threads?

Issues like this are one reason multithreaded programming is often considered an art.

summary

Threading is like stepping into an entirely new world and learning a whole new programming language, or at least a new set of language concepts. With the appearance of operating system support for threading in most microcomputers, extensions for threads have also been appearing in programming languages or libraries. In all cases thread programming (1) seems mysterious and requires a shift in the way you think about programming and (2) looks very similar to thread support in other languages, so when you understand threads, you understand a common tongue. And although support for threads can make Java seem like a more complicated language, don't blame Java. Threads are tricky.

One of the biggest difficulties with threads occurs because more than one thread may be sharing a resource, like the memory in an object, and you must make sure that multiple threads don't try to read & change that resource at the same time. This requires judicious use of the **synchronized** keyword, which is a very helpful tool but must be understood thoroughly as it can quietly introduce deadlock situations.

In addition, there's a certain art to the application of threads. Java is designed to allow you to create as many objects as you need to solve your problem (at least in theory. Creating millions of objects for an engineering finite-element analysis, for example, may not be practical in Java). However, it seems that there is an upper bound to the number of threads you'll want to because at some point a large number of threads seems to become unwieldy. This critical point is not in the many thousands as it might be with objects, but rather in the neighborhood of less than 100. As you may often only create a handful of threads to solve a problem this is typically not much of a limit, yet in a more general design it becomes a constraint.

A significant non-intuitive issue in threading is that, because of thread scheduling, you can typically make your applications run *faster* by inserting calls to **sleep( )** inside **run( )**'s main loop. This definitely makes it feel like an art, in particular when the longer delays seem to speed up performance. Of course, the reason this happens is that shorter delays can cause the end-of-**sleep( )** scheduler interrupt to happen before the running thread is ready to go to sleep, thus forcing the scheduler to stop it, and later restart it so it can finish what it was doing and then go to sleep. It takes extra thought to realize how messy things can get.

One thing you may notice missing in this chapter is an animation example, which is one of the most popular things to do with applets. However, a complete solution (with sound) to this problem comes with the Java JDK (available at the Sun/Javasoft site) in the demo section. In addition, we can expect better animation support to become part of future versions of Java, while completely different non-Java, non-programming solutions to animation for the Web are appearing that will probably be superior to traditional approaches. For explanations about how Java animation works, see *Core Java* (by Cornell & Horstmann, Prentice-Hall 1997).

exercises

1. Inherit a class from Thread and override the **run( )** method. Inside **run( )**, print a message, then call **sleep( )**. Repeat this 3 times, then return from **run( )**. Put a start-up message in the constructor and override **finalize( )** to print a shut-down message. Make a separate thread class that calls **System.gc( )** and **System.runFinalization( )** inside **run( )**, printing a message as it does so. Make several thread objects of both types and run them to see what happens.

2. Modify **Counter2.java** so that the thread is an inner class, and doesn't need to explicitly store a handle to a **Counter2**.

3. Modify **Sharing2.java** to add a **synchronized** block inside the **run( )** method of **TwoCounter**, instead of synchronizing the entire **run( )** method.

4. Create two Thread classes, one with a **run( )** that starts up, captures the handle of the second thread object and then calls **wait( )**. The other class' **run( )** should call **notifyAll( )** for the first thread after some number of seconds have passed, so the first thread can print out a message.

5. In **Counter5.java** inside **Ticker2**, remove the **yield( )** and explain the results. Replace the **yield( )** with a **sleep( )** and explain the results.

6. In **ThreadGroup1.java**, replace the call to **sys.suspend( )** with a call to **wait( )** for the thread group, causing it to wait for 2 seconds. For this to work correctly you'll need to acquire the lock for **sys** inside a **synchronized** block.

15

15: network programming

Historically, network programming has been error-prone, difficult, and complex.

The programmer had to know many details about the network and sometimes even the hardware. You usually needed to understand the various "layers" of the networking protocol, and there were lots of different methods in each different networking library concerned with connecting, packing and unpacking blocks of information, shipping those blocks back and forth, and handshaking. It was a daunting task.

However, the concept of networking is not so very difficult. You want to get some information from that machine over there and move it to this machine here, or vice-versa. It's quite similar to reading and writing files, except that the file exists on a remote machine and the remote machine can decide exactly what it wants to do about the information you're requesting or sending.

One of Java's great strengths is painless networking. As much as possible, the underlying details of networking have been abstracted away and taken care of within the JVM and local machine installation of Java. The programming model you use is that of a file; in fact, you actually wrap the network connection (a "socket") with **InputStream** and **OutputStream** objects, so you end up using exactly the same method calls as you do with files. In addition, Java's built-in multithreading is

exceptionally handy when dealing with another networking issue: handling multiple connections at once.

This chapter introduces Java's networking support using easy-to-understand examples.

identifying a machine

Of course, in order to tell one machine from another and to make sure you are connected with exactly the machine you want, there must be some way of uniquely identifying machines on a network. Early networks were satisfied to provide unique names for machines within the local network. However, Java works within the Internet, which requires a way to uniquely identify a machine from all the others *in the world*. This is accomplished with the IP (Internet Protocol) address that can exist in two forms:

1. The familiar DNS (Domain Name Service) form. My domain name is **EckelObjects.com**, so suppose I have a computer called **Opus** in my domain. Its domain name would be **Opus.EckelObjects.com**. This is exactly the kind of name that you use when you send email to people, and is often incorporated into a world-wide-Web address.

2. Alternatively, you can use the "dotted quad" form, which is four numbers separated by dots, such as **123.255.28.120**.

In both cases, the IP address is represented internally as a 32-bit number[1] (so each number cannot exceed 255), and you can get a special Java object to represent this number from either of the above forms by using the **static InetAddress.getByName( )** method that's in **java.net**. The result is an object of type **InetAddress** that you can use to build a "socket" as you shall see later.

servers and clients

The whole point of a network is to allow two machines to connect and talk to each other. Once the two machines have found each other they can have a nice, two-way conversation. But how do they find each other? It's like getting lost in an amusement park: one machine has to stay in one place and listen while the other machine says "hey, where are you?"

The machine that "stays in one place" is called the *server*, and the one that seeks is called the *client*. This distinction is only important while the client is trying to connect to the server. Once they've connected, it becomes a two-way communication process and it doesn't matter anymore that one happened to take the role of server and the other happened to take the role of the client.

So the job of the server is to listen for a connection, and that's performed by the special server object that you create. The job of the client is to try to make a connection to a server, and this is performed by the special client object you create. Once the connection is made, you'll see that at both server and client ends, the connection is just magically turned into an **InputFile** and **OutputFile** object, and from then on you just treat the connection as reading from and writing to a file. Thus, after the connection is made you will just use the familiar IO commands from Chapter 10. This is one of the very nice features of Java networking.

testing programs without a network

For many reasons, you may not have a client machine, a server machine, and a network available to test your programs. You may be performing exercises in a classroom situation, or you may be writing programs that aren't yet stable enough to put onto the network. How can you test your code?

---

[1] This means a maximum of just over 4 billion numbers, which is rapidly running out. The new standard for IP addresses will use a 128-bit number, which should produce enough unique IP addresses for the foreseeable future.

The creators of the Internet Protocol were aware of this issue, and they created a special address called **localhost** to be the "local loopback" IP address for testing without a network. The generic way to produce this address in Java is:

```
InetAddress addr = InetAddress.getByName(null);
```

If you hand **getByName( )** a **null**, it defaults to using the **localhost**. The **InetAddress** is what you use to refer to the particular machine, and you must produce this before you can go any further. You can't manipulate the contents of an **InetAddress** (but you can print them out, as you'll see in the next example). The only way you can create an **InetAddress** is through one of that classes' **static** member methods **getByName( )** (which is what you'll usually use), **getAllByName( )** or **getLocalHost( )**.

You can also produce the local loopback address by handing it the string **localhost**:

```
InetAddress.getByName("localhost");
```

Or by using it's dotted quad form to name the reserved IP number for the loopback:

```
InetAddress.getByName("127.0.0.1");
```

All three forms produce the same result.

port: a unique place within the machine

An IP address isn't enough to identify a unique server, since many servers can exist on one machine. Each IP machine also contains *ports*, and when you're setting up a client or a server you must choose a port to where both client and server agree to meet. As if you're meeting someone: the IP address is the neighborhood and the port is the bar.

The port is not a physical location in a machine, but is instead a software abstraction. The idea is that if you ask for a particular port you're requesting the service offered by that machine that's associated with the port number. For example, one service may simply be the time of day.

The system services reserve the use of ports 1 through 1024, so you shouldn't use those or any other port that you know to be in use. The first choice for examples in this book will be port 8080 (in memory of the venerable old 8-bit Intel 8080 chip in my first computer, a CP/M machine).

sockets

The *socket* is the software abstraction used to represent the "terminals" of a connection between two machines. For a given connection, there's a socket on each machine, and you can imagine a hypothetical "cable" running between the two machines, with each end of the "cable" plugged into a socket. Of course, the physical hardware and cabling between machines is completely unknown. The whole point of the abstraction is that we don't have to know more than is necessary.

In Java, you create a socket to make the connection to the other machine, then you get an **InputStream** and/or **OutputStream** from the socket in order to be able to treat the connection as an IO stream object. At first, it appears that there are two kinds of sockets: a **ServerSocket** which has the ability to wait for a connection from the network, and a **Socket** to represent a client. The **Socket** has the ability to connect to a machine by specifying the IP address and port.

This is another example of a confusing name scheme in the Java libraries, since **ServerSocket** might be better named "ServerConnector" or something without the word "Socket" in it. You might think that **ServerSocket** and **Socket** should both be inherited from some common base class. Indeed, the two classes do have several methods in common but not enough to give them a common base class. Instead, **ServerSocket** has a method called **accept( )** which waits until some other machine connects to it, then returns an actual **Socket**. From then on, you have a true **Socket** to **Socket** connection and you treat both ends the same way because they *are* the same. This is why **ServerSocket** is a bit misnamed, since its job isn't really to be a socket but instead to make a **Socket** object when someone else connects to it.

When you create a **ServerSocket**, you only need to give it the port number that it's going to use. You don't have to give it an IP address because it's already on the machine it's representing. When you create a **Socket**, however, you must give both the IP address and the port number that you're trying to connect to.

Once you get **Socket** objects on both ends, you use the methods **getInputStream( )** and **getOutputStream( )** to produce the corresponding **InputStream** and **OutputStream** objects. These must be wrapped inside buffers and formatting classes just like any other stream object described in Chapter 10.

a simple server and client

This example makes the simplest use of servers and clients using sockets. All the server does is wait for a connection, then use the **Socket** produced by that connection to create an **InputStream** and **OutputStream**. Then everything that it reads from the **InputStream** it echoes to the **OutputStream** until it receives the line END, at which time it closes the connection.

The client makes the connection to the server, then creates an **OutputStream**. Lines of text are sent through the **OutputStream**. The client also creates an **InputStream** to hear what the server is saying (which, in this case, is just the words echoed back).

Both the server and client use the same port number and the client uses the local loopback address to connect to the server on the same machine so you don't have to test it over a network.

Here is the server:

```
//: JabberServer.java
// Very simple server that just
// echoes whatever the client sends.
import java.io.*;
import java.net.*;

public class JabberServer {
  // Choose a port outside of the range 1-1024:
  static final int port = 8080;
  public static void main(String[] args ) {
    try {
      ServerSocket s = new ServerSocket(port);
      System.out.println("Server Started: " + s);
      // Blocks until a connection occurs:
      Socket socket = s.accept();
      System.out.println(
        "Connection accepted, socket: "+ socket);
      BufferedReader in =
        new BufferedReader(
          new InputStreamReader(
            socket.getInputStream()));
      // Output is automatically flushed
      // by PrintWriter:
      PrintWriter out =
        new PrintWriter(
          new BufferedWriter(
            new OutputStreamWriter(
              socket.getOutputStream())), true);
      while (true) {
        String str = in.readLine();
        if (str.equals("END")) break;
        System.out.println("Echoing: " + str);
        out.println(str);
      }
      System.out.println("closing...");
```

```
      socket.close();
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
} ///:~
```

You can see that the **ServerSocket** just needs a port number, not an IP address (since it's running on *this* machine!). When you call **accept( )**, the method blocks until some client tries to connect to it. That is, it's there waiting for a connection but other processes can run (see Chapter 14). When a connection is made, **accept( )** returns with a **Socket** object representing that connection.

Both the **ServerSocket** and the **Socket** produced by **accept( )** are printed to **System.out**. This means their **toString( )** methods are automatically called. These produce:

```
ServerSocket[addr=0.0.0.0,port=0,localport=8080]
Socket[addr=127.0.0.1,port=1077,localport=8080]
```

Shortly, you'll see how these fit together with what the client is doing.

The next part of the program looks just like opening files for reading and writing except that the **InputStream** and **OutputStream** are created from the **Socket** object. Both the **InputStream** and **OutputStream** objects are converted to Java 1.1 **Reader** and **Writer** objects using the "converter" classes **InputStreamReader** and **OutputStreamWriter**, respectively. You could also have used the Java 1.0 **InputStream** and **OutputStream** classes directly, but with output there's a distinct advantage to using the **Writer** approach. This appears with **PrintWriter**, which has an overloaded constructor that takes a second argument, a **boolean** flag which indicates whether or not to automatically flush the output at the end of each **print( )** or **println( )** statement. Every time you write to **out**, its buffer must be flushed so the information actually goes out over the network. This is important for this particular example because the client and server each wait for a line from the other party before proceeding. If flushing doesn't occur, the information would not be put onto the network until the buffer is full which causes lots of problems in this example.

Notice that, like virtually all files you open, these are buffered. There's an exercise at the end of the chapter to show you what happens if you don't buffer the streams (things get very slow).

The infinite **while** loop just reads lines from the **BufferedReader in** and writes information to **System.out** and to the **PrintWriter out**. Notice that these could be any streams at all, they just happen to be connected to the network.

When the client sends the line consisting of "END" the program breaks out of the loop and closes the **Socket**. If you don't call **close( )** in this example it's not disastrous but it's always a good idea to make sure everything's cleaned up properly.

Here's the client:

```
//: JabberClient.java
// Very simple client that just sends
// lines to the server and reads lines
// that the server sends.
import java.net.*;
import java.io.*;

public class JabberClient {
  // Choose a port outside of the range 1-1024:
  static final int port = 8080;
  public static void main(String args[]) {
    try {
      // Passing null to getByName() produces the
      // special "Local Loopback" IP address, for
      // testing on one machine w/o a network:
      InetAddress addr =
```

```
        InetAddress.getByName(null);
      // Alternatively, you can use
      // the address or name:
      // InetAddress addr =
      //    InetAddress.getByName("127.0.0.1");
      // InetAddress addr =
      //    InetAddress.getByName("localhost");
      System.out.println("addr = " + addr);
      Socket socket = new Socket(addr, port);
      System.out.println("socket = " + socket);
      BufferedReader in =
        new BufferedReader(
          new InputStreamReader(
            socket.getInputStream()));
      // Output is automatically flushed
      // by PrintWriter:
      PrintWriter out =
        new PrintWriter(
          new BufferedWriter(
            new OutputStreamWriter(
              socket.getOutputStream())),true);
      for(int i = 0; i < 10; i ++) {
        out.println("howdy " + i);
        String str = in.readLine();
        System.out.println(str);
      }
      out.println("END");
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
} ///:~
```

In **main( )** you can see all three ways to produce the **InetAddress** of the local loopback IP address: using **null**, **localhost** or the explicit reserved address **127.0.0.1**. Of course, if you want to actually connect to a machine across a network you substitute that machine's IP address. When the **InetAddress addr** is printed (via the automatic call to its **toString( )** method) the result is:

```
localhost/127.0.0.1
```

By handing **getByName( )** a **null**, it defaulted to finding the **localhost**, and that produced the special address **127.0.0.1**.

Note that the **Socket** called **socket** is created with both the **InetAddress** and the port number. To understand what it means when you print out one of these **Socket** objects, remember that an Internet connection is determined uniquely by these four pieces of data: **clientHost**, **clientPortNumber**, **serverHost**, and **serverPortNumber**. When the server comes up, it takes up its assigned port (8080) on the localhost (127.0.0.1). When the client comes up, it is allocated the next available port on its machine, 1077 in this case, which also happens to be on the same machine (127.0.0.1) as the server. Now in order for data to move between the client and server, each side has to know where to send it. Therefore, during the process of connecting to the "known" server, the client sends a "return address" so that the server knows where to send its data. This is what you see in the example output for the server side:

```
Socket[addr=127.0.0.1,port=1077,localport=8080]
```

This means the server just accepted a connection from 127.0.0.1 on port 1077 while listening on its local port (8080). On the client side:

```
Socket[addr=localhost/127.0.0.1,port=8080,localport=1077]
```

Which means the client made a connection to 127.0.0.1 on port 8080 using the local port 1077.

You'll notice that every time you start up the client anew, the local port number is incremented. It starts at 1025 (one past the reserved block of ports) and keeps going up until you reboot the machine, at which point it starts back at 1025 again (on UNIX machines, once the upper limit of the socket range is reached the numbers will wrap around to the lowest available number again).

Once the **Socket** object has been created, the process of turning it into a **BufferedReader** and **PrintWriter** is the same as in the server (again, in both cases you start with a **Socket**). Here, the client initiates the conversation by sending the string "howdy" followed by a number. Note that the buffer must again be flushed (which happens automatically via the second argument to the **PrintWriter** constructor); if you don't the whole conversation will hang because the initial "howdy" will never get sent (the buffer isn't full enough to cause the send to happen automatically). Each line that is sent back from the server is written to **System.out** to verify that everything is working correctly. To terminate the conversation, the agreed-upon "END" is sent. If the client simply hangs up then the server throws an exception.

Using sockets produces a "dedicated" connection which persists until it is explicitly disconnected (the dedicated connection can still be disconnected un-explicitly if one side, or an intermediary link, of the connection crashes). This means the two parties are locked in communication and the connection is constantly open. This seems like a logical approach to networking, but it puts an extra load on the network. Later in the chapter you'll see a different approach to networking, where the connections are only temporary.

serving multiple clients

The **JabberServer** works, but it can only handle one client at a time. In a typical server, you'll want to be able to deal with many clients at once. The answer is multithreading, and in languages that don't directly support multithreading this means all sorts of complications. In Chapter 14 you saw that multithreading in Java is about as simple as possible, considering that multithreading is a relatively complex topic. Because threading in Java is reasonably straightforward, making a server that handles multiple clients is relatively easy.

The basic scheme is to make a single **ServerSocket** in the server, and call **accept( )** to wait for a new connection. When **accept( )** returns, you take the resulting **Socket** and use it to create a new thread whose job is to serve that particular client, then you call **accept( )** again to wait for a new client.

In the following server code, you can see that it looks very similar to the **JabberServer.java** example except that all the operations to serve a particular client have been moved inside a separate thread class:

```
//: MultiJabberServer.java
// A server that uses multithreading to handle
// any number of clients.
import java.io.*;
import java.net.*;

class ServeOneJabber extends Thread {
  private Socket socket;
  private BufferedReader in;
  private PrintWriter out;
  ServeOneJabber(Socket s) {
    socket = s;
    try {
      in =
        new BufferedReader(
          new InputStreamReader(
            socket.getInputStream()));
      // Enable auto-flush:
      out =
        new PrintWriter(
```

```
            new BufferedWriter(
              new OutputStreamWriter(
                socket.getOutputStream())), true);
      } catch(IOException e) {
        e.printStackTrace();
      }
      start(); // Calls run()
    }
    public void run() {
      try {
        while (true) {
          String str = in.readLine();
          if (str.equals("END")) break;
          System.out.println("Echoing: " + str);
          out.println(str);
        }
        System.out.println("closing...");
        in.close();
        out.close();
        socket.close();
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
}

public class MultiJabberServer {
  static final int port = 8080;
  public static void main(String[] args ) {
    try {
      ServerSocket s = new ServerSocket(port);
      System.out.println("Server Started");
      while(true) {
        // Blocks until a connection occurs:
        Socket socket = s.accept();
        new ServeOneJabber(socket);
      }
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
} ///:~
```

The **ServeOneJabber** thread takes the **Socket** object that's produced by **accept( )** in **main( )** every time a new client makes a connection. Then, as before, it creates a **BufferedReader** and auto-flushed **PrintWriter** object using the **Socket**. Finally, it calls the special **Thread** method **start( )**, which performs thread initialization and then calls **run( )**. This performs the same kind of action as in the previous example: reading something from the socket and then echoing it back until it reads the special "END" signal. Notice that both **in** and **out** are explicitly cleaned up with calls to **close( )**. If you forget to do this, you'll use up system file resources and start seeing exceptions.

Notice the simplicity of the **MultiJabberServer**. As before, a **ServerSocket** is created and **accept( )** is called to allow a new connection. But this time, the return value of **accept( )** (a **Socket**) is passed to the constructor for **ServeOneJabber** which creates a new thread to handle that connection. When the connection is terminated, the thread simply goes away.

To test that the server really does handle multiple clients, the following program creates many clients (using threads), each of which connects to the same server. Each thread has a limited lifetime, and when it goes away that leaves space for the creation of a new thread. The maximum number of threads allowed is determined by the **final int maxthreads**. You'll notice this value is rather critical,

since if you make it too high the threads seem to run out of file resources and the program mysteriously fails.

```
//: MultiJabberClient.java
// Client that tests the MultiJabberServer
// by starting up multiple clients.
import java.net.*;
import java.io.*;

class JabberClientThread extends Thread {
  static final int port = 8080;
  private Socket socket;
  BufferedReader in;
  PrintWriter out;
  private static int counter = 0;
  private int id = counter++;
  static final int maxthreads = 40;
  static int threadcount = 0;
  JabberClientThread(InetAddress addr) {
    System.out.println("Making client " + id);
    threadcount++;
    try {
      socket = new Socket(addr, port);
      in =
        new BufferedReader(
          new InputStreamReader(
            socket.getInputStream()));
      // Enable auto-flush:
      out =
        new PrintWriter(
          new BufferedWriter(
            new OutputStreamWriter(
              socket.getOutputStream())), true);
      start();
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
  public void run() {
    try {
      for(int i = 0; i < 25; i++) {
        out.println("Client " + id + ": " + i);
        String str = in.readLine();
        System.out.println(str);
      }
      out.println("END");
      in.close();
      out.close();
      socket.close();
      threadcount--; // Ending this thread
    } catch(IOException e) {
      e.printStackTrace();
    }
  }
}

public class MultiJabberClient {
  public static void main(String args[]) {
    try {
      InetAddress addr =
```

```
          InetAddress.getByName(null);
        while(true) {
          if(JabberClientThread.threadcount <
             JabberClientThread.maxthreads)
            new JabberClientThread(addr);
          Thread.currentThread().sleep(100);
        }
      } catch(Exception e ) {
        e.printStackTrace();
      }
    }
  } ///:~
```

The **JabberClientThread** constructor takes an **InetAddress** and uses it to open a **Socket**. You're probably now starting to see the pattern – the **Socket** is always used to create some kind of **Reader** and/or **Writer** (or **InputStream** and/or **OutputStream**) object, which is the only way the **Socket** can be used (you can of course write a class or two to automate this process rather than doing all the typing if it becomes painful). Again, **start( )** performs thread initialization and calls **run( )**. Here, messages are sent to the server and information from the server is echoed to the screen. However, the thread has a limited lifetime and eventually completes. It's very important to call **close( )** for the **Reader** and **Writer** (**InputStream** and **OutputStream**) objects, otherwise you'll exhaust resources and start seeing exceptions.

The **threadcount** keeps track of how many **JabberClientThread** objects currently exist. It is incremented as part of the constructor, and decremented as **run( )** exits (which means the thread is terminating). In **MultiJabberClient.main( )** you can see that the number of threads is tested, and if there are too many the method sleeps. This way, some threads will eventually terminate and more can be created. You can experiment with **maxthreads** to see where your particular system begins to have trouble with too many connections.

datagrams

The examples you've seen so far use the *Transmission Control Protocol* (TCP), which is designed for ultimate reliability and guarantees that the data will get there. It allows retransmission of lost data, it provides multiple paths through different routers in case one goes down, and bytes are delivered in the order they are sent. All this control and reliability comes at a cost: TCP has a high overhead.

There's a second protocol, called *User Datagram Protocol* (UDP), which doesn't guarantee that the packets will be delivered and doesn't guarantee that they will be delivered in the order they were sent. It's called an "unreliable protocol" (TCP is "reliable"), which sounds bad but because it's much faster it can be very useful. There are some applications, such as an audio signal, where it isn't so critical if a few packets are dropped here or there but speed is very important. Or consider a time-of-day server, where it really doesn't matter if one of the messages is lost. Also, some applications may be able to fire off a UDP message to a server and can then assume, if there is no response in a reasonable period of time, that the message was lost.

The support for datagrams in Java has the same feel as its support for TCP sockets, but there are significant differences. With datagrams, you put a **DatagramSocket** on both the client and server, but there is no analogy to the **ServerSocket** that waits around for a connection. That's because there is no "connection," but instead a datagram just shows up. Another fundamental difference is that with TCP sockets, once you've made the connection you don't need to worry about who's talking to whom anymore; you just send the data back and forth through conventional streams. However, with datagrams, the datagram packet itself must know where it came from and where it's supposed to go. That means you'll have to know these things for each datagram packet that you load up and ship off.

A **DatagramSocket** sends and receives the packets, and the **DatagramPacket** contains the information itself. When you're receiving a datagram, you only need to provide a buffer where the data will be placed; the information about the Internet address and port number where the

information came from will be automatically initialized when the packet arrives through the **DatagramSocket**. So the constructor for a **DatagramPacket** to receive datagrams is:

```
DatagramPacket(buf, buf.length)
```

Where **buf** is an array of **byte**. You might wonder that, if **buf** is an array, why couldn't the constructor figure out the length of the array on its own. I wondered this myself, and do not have an answer.

You can reuse a receiving datagram over and over; you don't have to make a new one each time. Every time you reuse it, the data in the buffer is overwritten.

The maximum size of the buffer is only limited by the allowable datagram packet size, which limits it to slightly less than 64Kbytes. However, in many applications you'll want it to be much smaller, certainly when you're sending data. Your chosen packet size depends on what you need for your particular application.

When you send a datagram, the **DatagramPacket** must contain not only the data, but also the Internet address and port where it will be sent. So the constructor for an outgoing **DatagramPacket** is:

```
DatagramPacket(buf, length, inetAddress, port)
```

This time, **buf** (which is a **byte** array) already contains the data that you want to send out. The **length** may be the length of **buf**, but it may also be shorter, indicating that you only want to send that many bytes. The other two arguments are the Internet address where the packet is going and the destination port within that machine.

You might think that the two constructors create two very different objects: one for receiving datagrams, and one for sending them. Good OO design would suggest that these should be two different classes, rather than one class with different behavior depending on how you construct the object. This is probably true, but fortunately the use of **DatagramPacket**s is simple enough that you're not tripped up by the problem, as you can see in the following example. This example is similar to the **MultiJabberServer** and **MultiJabberClient** example for TCP sockets. Multiple clients will send datagrams to a server, which will echo them back to the same client that sent the message.

To simplify the creation of a **DatagramPacket** from a **String** and vice-versa, the example begins with a utility class, **Dgram**, to do the work for you:

```
//: Dgram.java
// A utility class to convert back and forth
// Between Strings and DataGramPackets.
import java.net.*;

public class Dgram {
  public static DatagramPacket toDatagram(
    String s, InetAddress destIA, int destPort) {
    // Deprecated in Java 1.1, but it works:
    byte buf[] = new byte[s.length() + 1];
    s.getBytes(0, s.length(), buf, 0);
    // The correct Java 1.1 approach, but it's
    // Broken (it truncates the String):
    // byte buf[] = s.getBytes();
    return new DatagramPacket(buf, buf.length,
      destIA, destPort);
  }
  public static String toString(DatagramPacket p){
    // The Java 1 approach:
    // return new String(p.getData(),
    //  0, 0, p.getLength());
    // The Java 1.1 approach:
    return new String(
      p.getData(), 0, p.getLength());
  }
```

```
} ///:~
```

The first method of **Dgram** takes a **String**, an **InetAddress** and a port number and builds a **DatagramPacket** by copying the contents of the **String** into a **byte** buffer and passing the buffer into the **DatagramPacket** constructor. Note the "+1" in the buffer allocation – this was necessary to prevent truncation. The **getBytes( )** method of **String** is a special operation that copies the **char**s of a **String** into a **byte** buffer. This method is now deprecated; Java 1.1 has a "better" way to do this but it's commented out here because it truncates the **String**. So you'll get a deprecation message when you compile it under Java 1.1, but the behavior will be correct.

The **Dgram.toString( )** method shows both the Java 1 approach and the new Java 1.1 approach (which is different because there's a new kind of **String** constructor).

Here is the server for the datagram demonstration:

```
//: ChatterServer.java
// A server that echoes datagrams
import java.net.*;
import java.io.*;
import java.util.*;

public class ChatterServer {
  static int inPort = 1711;
  byte buf[] = new byte[1000];
  DatagramPacket dp =
    new DatagramPacket(buf, buf.length);
  // Can listen & send on the same socket:
  DatagramSocket socket;

  public ChatterServer() {
    try {
      socket = new DatagramSocket(inPort);
      System.out.println("Server started");
      while(true) {
        // Block until a datagram appears:
        socket.receive(dp);
        String rcvd = Dgram.toString(dp) +
          ", from address: " + dp.getAddress() +
          ", port: " + dp.getPort();
        System.out.println(rcvd);
        String echoString =
          "Echoed: " + rcvd;
        // Extract the address and port from the
        // received datagram to find out where to
        // send it back:
        DatagramPacket echo =
          Dgram.toDatagram(echoString,
            dp.getAddress(), dp.getPort());
        socket.send(echo);
      }
    } catch(SocketException e) {
      System.err.println("Can't open socket");
      System.exit(1);
    } catch(IOException e) {
      System.err.println("Communication error");
      e.printStackTrace();
    }
  }
  public static void main(String[] args) {
    new ChatterServer();
  }
```

```
} ///:~
```

The **ChatterServer** contains a single **DatagramSocket** for receiving messages, instead of creating one each time you're ready to receive a new message. The single **DatagramSocket** can be used over and over. This **DatagramSocket** has a port number because this is the server and the client must be able to exactly address where it wants a datagram to go. It is given a port number but not an Internet address because it resides on "this" machine so it knows what its Internet address is (in this case, the default **localhost**). In the infinite **while** loop, the **socket** is told to **receive( )**, whereupon it blocks until a datagram shows up, and then sticks it into our designated receiver, the **DatagramPacket dp**. The packet is converted to a **String** along with information about the Internet address and socket where the packet came from. This information is displayed, and then an extra string is added to indicate that it is being echoed back from the server.

Now there's a bit of a quandary. As you shall see, there are potentially many different Internet addresses and port numbers that the messages may come from – that is, the clients may reside on any machine (in this demonstration they all reside on the **localhost**, but the port number for each client is different). To send a message back to the client that originated it, you need to know that client's Internet address and port number. Fortunately, this information is conveniently packaged inside the **DatagramPacket** that sent the message, so all you have to do is pull it out using **getAddress( )** and **getPort( )**, which are used to build the **DatagramPacket echo** which is sent back through the same socket that's doing the receiving. In addition, when the socket sends the datagram, it automatically adds the Internet address and port information of *this* machine, so that when the client receives the message, it can use **getAddress( )** and **getPort( )** to find out where the datagram came from. In fact, the only time that **getAddress( )** and **getPort( )** don't tell you where the datagram came from is if you create a datagram to send, and you call **getAddress( )** and **getPort( )** *before* you send the datagram (in which case it tells the address and port of this machine, the one the datagram is being sent from). This is an essential part of datagrams: you don't need to keep track of where a message came from, because it's always stored inside the datagram. In fact, the most reliable way to program is if you don't try to keep track, but instead always extract the address and port from the datagram in question (as is done here).

To test this server, here's a program that makes a number of clients, all of which fire datagram packets to the server and wait for the server to echo them back.

```
//: ChatterClient.java
// Tests the ChatterServer by starting up multiple
// clients, each of which sends datagrams.
import java.lang.Thread;
import java.net.*;
import java.io.*;

public class ChatterClient extends Thread {
  // Can listen & send on the same socket:
  DatagramSocket s;
  InetAddress hostAddress;
  byte buf[] = new byte[1000];
  DatagramPacket dp =
    new DatagramPacket(buf, buf.length);
  int id;

  ChatterClient(int identifier) {
    id = identifier;
    try {
      // Auto-assign port number:
      s = new DatagramSocket();
      hostAddress =
        InetAddress.getByName("localhost");
    } catch(UnknownHostException e) {
      System.err.println("Cannot find host");
      System.exit(1);
    } catch(SocketException e) {
```

```
      System.err.println("Can't open socket");
      e.printStackTrace();
      System.exit(1);
    }
    System.out.println("ChatterClient starting");
  }
  public void run() {
    try {
      for(int i = 0; i < 25; i++) {
        String outMessage = "Client #" +
          id + ", message #" + i;
        // Make and send a datagram:
        s.send(Dgram.toDatagram(outMessage,
          hostAddress,
          ChatterServer.inPort));
        // Block until it echoes back:
        s.receive(dp);
        // Print out the echoed contents:
        String rcvd = "Client #" + id +
          ", rcvd from " +
          dp.getAddress() + ", " +
          dp.getPort() + ": " +
          Dgram.toString(dp);
        System.out.println(rcvd);
      }
    } catch(IOException e) {
      e.printStackTrace();
      System.exit(1);
    }
  }
  public static void main(String[] args) {
    for(int i = 0; i < 10; i++)
      new ChatterClient(i).start();
  }
} ///:~
```

ChatterClient is created as a **Thread** so that multiple clients may be made to bother the server. Here you can see the receiving **DatagramPacket** looks just like the one used for **ChatterServer**. In the constructor, the **DatagramSocket** is created with no arguments since it doesn't need to advertise itself as being at a particular port number. The Internet address used for this socket will be "this machine" (for the example, **localhost**) and the port number will be automatically assigned as you will see from the output. This **DatagramSocket**, like the one for the server, will be used both for sending and receiving.

The **hostAddress** is the Internet address of the host machine you want to talk to. The one place in the program where you must know an exact Internet address and port number is when you make the outgoing **DatagramPacket**. As is always the case, the host must be at a known address and port number so that clients may originate conversations with the host.

Each thread is given a unique identification number (although the port number automatically assigned to the thread would also provide a unique identifier). In **run( )**, a message **String** is created that contains the thread's identification number and which message number this thread is currently sending. This **String** is used to create a datagram which is sent to the host at it's address; the port number is produced from the value that **ChatterServer** has chosen for it's **static inPort**. Once the message is sent, **receive( )** blocks until the server replies with an echoing message. All the information that's shipped around with the message allows you to see that what comes back to this particular thread is derived from the message that originated from this thread. In this example, even though UDP is an "unreliable" protocol, you'll see that all the datagrams get where they're supposed to (this will be true for localhost and LAN situations, but you may begin to see some failures for non-local connections).

When you run this program, you'll see that each of the threads finishes, which means that each of the datagram packets sent to the server is actually turned around and echoed to the right recipient, otherwise one or more threads would hang, blocking until their input shows up.

You might think that the only correct way to, for example, transfer a file from one machine to another is through TCP sockets, since they're "reliable." However, because of the speed of datagrams they can actually be a better solution. You simply break the file up into packets and number each packet. The receiving machine takes the packets and reassembles them; a "header packet" tells it how many to expect and any other important information. If a packet is lost, the receiving machine sends a datagram back telling the sender to retransmit.

a web application

Now let's consider creating an application to run on the Web, which will show Java in all its glory. Part of this application will be a Java program running on the Web server, and the other part will be an applet that's downloaded to the browser. The applet collects information from the user and sends it back to the application running on the Web server. The program will be very simple –the applet will ask for the email address of the user, and after verifying that this email address is reasonably legitimate (it doesn't contain spaces, and it does contain an '@' symbol) the applet will send the email address to the web server. The application running on the server will capture the data and check a data file where all the email addresses are kept. If that address is already in the file, it will send back a message to that effect which is displayed by the applet. If the address isn't in the file, it is placed in the list and the applet is informed that the address was successfully added.

Traditionally, the way to handle such a problem is to create an HTML page with a text field and a "submit" button. The user can type whatever they want into the text field and it will be submitted to the server without question. As it submits the data, the Web page also tells the server what to do with the data by mentioning the CGI program that the server should run after receiving this data. This CGI program is typically written in either Perl or C (and sometimes C++, if the server supports it), and it must handle everything. First it looks at the data and decides whether it's in the correct format. If not, the CGI program must create an HTML page to describe the problem; this page is handed to the server which sends it back to the user, who must then back up a page and try again. If the data is correct, the CGI program opens the data file and either adds the email address to the file or discovers that the email address is already in the file. In both cases it must format an appropriate HTML page for the server to return to the user.

Since this seems like such an awkward way to solve the problem, we'd like to do the whole thing in Java. First of all, a Java applet to take care of data validation at the client site, without all that tedious Web traffic and page formatting. Then let's skip the Perl CGI script in favor of a Java application running on the server. In fact, let's skip the Web server altogether, and simply make our own network connection from the applet to the Java application on the server!

As you'll see, there are a number of issues that make this a more complicated problem than it seems. First of all, it would be ideal to write the applet using Java 1.1 but that's hardly practical. At this writing, the number of users running Java 1.1 -enabled browsers is virtually nil, and even after such browsers become commonly available you'll probably need to take into account that a significant number of users will be slow to upgrade. So to be on the safe side, the applet will be programmed using only Java 1.0 code. With this in mind, there will be no JAR files to combine **.class** files in the applet, so the applet should be designed to create as few **.class** files as possible to minimize download time.

Well, it turns out the Web server (the one available to me when I wrote the example) *does* have Java on it, but only Java 1.0! So the server application must also be written using Java 1.0.

the server application

Now consider the server application, which will be called **NameCollector**. What happens if more than one user at a time tries to submit their email address? If **NameCollector** uses TCP/IP sockets, then it must use the multithreading approach shown earlier to handle more than one client at a time. But all these threads will try to write to a single file where all the email addresses will be kept. This would

require a locking mechanism to make sure that more than one thread doesn't access the file at once. A semaphore will do the trick, but perhaps there's a simpler way.

If we use datagrams instead, multithreading is unnecessary. A single datagram socket will listen for incoming datagrams, and when one appears the program will process the message and send the reply as a datagram back to whomever sent the request. If the datagram gets lost, then the user will notice that no reply comes and can then re-submit the request.

When the server application receives a datagram and unpacks it, all it must do is extract the email address and check the file to see if that address is there already (and if it isn't, add it). And now we run into another problem. It turns out that Java 1.0 doesn't quite have the horsepower to easily manipulate the file containing the email addresses (Java 1.1 does). However, the problem can be solved in C quite readily, and this will provide an excuse to show you the easiest way to connect a non-Java program to a Java program. A **Runtime** object for a program has a method called **exec( )** which will start up a separate program on the machine, and return a **Process** object. You can get an **OutputStream** that connects to standard input for this separate program, and an **InputStream** that connects to standard output. This means all you need to do is write a program using any language that takes its input from standard input and writes the output to standard output. This is a very convenient trick when you run into a problem that can't be solved easily or quickly enough in Java (or when you have legacy code you don't want to rewrite). You can also use Java's *native methods* (see Appendix A) but those are much more involved.

the C program

The job of this non-Java application is to manage the list of email addresses. Standard input will accept an email address and the program will look up the name in the list to see if it's already there. If not, it will add it and report success, but if the name is already there it will report that. Don't worry if you don't completely understand what the following code means; it's just one example of how you can write a program in another language and use it from Java, but the particular programming language doesn't really matter as long as it can read from standard input and write to standard output.

```
//: Listmgr.c
// Used by NameCollector.java to manage
// the email list file on the server
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define BSIZE 250

int alreadyInList(FILE* list, char* name) {
  char lbuf[BSIZE];
  /* Go to the beginning of the list: */
  fseek(list, 0, SEEK_SET);
  /* Read each line in the list: */
  while(fgets(lbuf, BSIZE, list)) {
    /* Strip off the newline: */
    char * newline = strchr(lbuf, '\n');
    if(newline != 0)
      *newline = '\0';
    if(strcmp(lbuf, name) == 0)
      return 1;
  }
  return 0;
}

int main() {
  char buf[BSIZE];
  FILE* list = fopen("emlist.txt", "a+t");
  if(list == 0) {
    perror("could not open emlist.txt");
    exit(1);
```

```
  }
  while(1) {
    gets(buf); /* From stdin */
    if(alreadyInList(list, buf)) {
      printf("Already in list: %s", buf);
      fflush(stdout);
    }
    else {
      fseek(list, 0, SEEK_END);
      fprintf(list, "%s\n", buf);
      fflush(list);
      printf("%s added to list", buf);
      fflush(stdout);
    }
  }
} ///:~
```

This assumes that the C compiler accepts '//' style comments (many do, and you can also compile this program with a C++ compiler). If yours doesn't, simply delete those comments.

The first function in the file checks to see whether the name you hand it as a second argument (a pointer to a **char**) is in the file. Here, the file is passed as a **FILE** pointer to an already-opened file (the file is opened inside **main( )**). The function **fseek( )** moves around in the file; here it is used to move to the top of the file. **fgets( )** reads a line from the file **list** into the buffer **lbuf**, not exceeding the buffer size **BSIZE**. This is inside a **while** loop so that each line in the file is read. Next, **strchr( )** is used to locate the newline character so that it can be stripped off. Finally, **strcmp( )** is used to compare the name you've passed into the function to the current line int the file. **strcmp( )** returns zero if it finds a match. In this case the function exits and a one is returned to indicate that yes, the name was already in the list (notice the function returns as soon as it discovers the match, so it doesn't waste time looking at the rest of the list). If you get all the way through the list without a match, the function returns zero.

In **main( )**, the file is opened using **fopen( )**. The first argument is the file name and the second is the way to open the file; **a+** means "append, and open (or create if the file does not exist) for update at the end of the file." The **fopen( )** function returns a **FILE** pointer which, if it's zero, means the open was unsuccessful. This is dealt with by printing an error message with **perror( )** and terminating the program with **exit( )**.

Assuming the file was opened successfully, the program enters an infinite loop. The function call **gets(buf)** gets a line from standard input (which will be connected to the Java program, remember) and places it in the buffer **buf**. This is simply passed to the **alreadyInList( )** function, and if it's already in the list, **printf( )** sends that message to standard output (where the Java program is listening). **fflush( )** is a way to flush the output buffer.

If the name is not already in the list, **fseek( )** is used to move to the end of the list and **fprintf( )** "prints" the name to the end of the list. Then **printf( )** is used to indicate that the name was added to the list (again flushing standard output) and the infinite loop goes back to waiting for a new name.

Remember that you normally cannot compile this program on your computer and just load it onto the Web server machine, since that machine may use a different processor and operating system. For example, my Web server runs on an Intel processor but it uses Linux, so I must download the source code and compile using remote commands with the C compiler that comes with the Linux distribution.

the Java program

This program will first start the above C program and make the necessary connections to talk to it. Then it will create a datagram socket which will be used to listen for datagram packets from the applet.

```
//: NameCollector.java
// Extracts email names from datagrams and stores
```

```
// them inside a file, using Java 1.02.
import java.net.*;
import java.io.*;
import java.util.*;

public class NameCollector {
  final static int COLLECTOR_PORT = 8080;
  final static int BUFFER_SIZE = 1000;
  byte buf[] = new byte[BUFFER_SIZE];
  DatagramPacket dp =
    new DatagramPacket(buf, buf.length);
  // Can listen & send on the same socket:
  DatagramSocket socket;
  Process listmgr;
  PrintStream nameList;
  DataInputStream addResult;
  public NameCollector() {
    try {
      listmgr =
        Runtime.getRuntime().exec("listmgr.exe");
      nameList = new PrintStream(
        new BufferedOutputStream(
          listmgr.getOutputStream()));
      addResult = new DataInputStream(
        new BufferedInputStream(
          listmgr.getInputStream()));

    } catch(IOException e) {
      System.err.println(
        "Cannot start listmgr.exe");
      System.exit(1);
    }
    try {
      socket =
        new DatagramSocket(COLLECTOR_PORT);
      System.out.println(
        "NameCollector Server started");
      while(true) {
        // Block until a datagram appears:
        socket.receive(dp);
        String rcvd = new String(dp.getData(),
            0, 0, dp.getLength());
        // Send to listmgr.exe standard input:
        nameList.println(rcvd.trim());
        nameList.flush();
        byte resultBuf[] = new byte[BUFFER_SIZE];
        int byteCount =
          addResult.read(resultBuf);
        if(byteCount != -1) {
          String result =
            new String(resultBuf, 0).trim();
          // Extract the address and port from
          // the received datagram to find out
          // where to send the reply:
          InetAddress senderAddress =
            dp.getAddress();
          int senderPort = dp.getPort();
          byte echoBuf[] = new byte[BUFFER_SIZE];
          result.getBytes(
            0, byteCount, echoBuf, 0);
```

```
            DatagramPacket echo =
              new DatagramPacket(
                echoBuf, echoBuf.length,
                senderAddress, senderPort);
            socket.send(echo);
          }
          else
            System.out.println(
              "Unexpected lack of result from " +
              "listmgr.exe");
        }
      } catch(SocketException e) {
        System.err.println("Can't open socket");
        System.exit(1);
      } catch(IOException e) {
        System.err.println("Communication error");
        e.printStackTrace();
      }
    }
    public static void main(String[] args) {
      new NameCollector();
    }
  } ///:~
```

The first definitions in **NameCollector** should look familiar: the port is chosen, a datagram packet is created, and there's a handle to a **DatagramSocket**. The next three definitions concern the connection to the C program: a **Process** object is what comes back when the C program is fired up by the Java program, and that **Process** object produces the **InputStream** and **OutputStream** objects representing, respectively, the standard output and standard input of the C program. These must of course be "wrapped" as is usual with Java IO, so we end up with a **PrintStream** and **DataInputStream**

All the work for this program happens inside the constructor. To start up the C program, the current **Runtime** object is procured. This is used to call **exec( )**, which returns the **Process** object. You can see that there are simple calls to produce the streams from the **Process** object: **getOutputStream( )** and **getInputStream( )**. From this point on, all you need to consider is sending data to the stream **nameList** and getting the results from **addResult**.

As before, a **DatagramSocket** is connected to a port. Inside the infinite **while** loop, the program calls **receive( )**, which blocks until a datagram shows up. When the datagram appears, its contents are extracted into the **String rcvd**. This is trimmed to remove white space at each end, and sent to the C program in the line:

```
nameList.println(rcvd.trim());
```

This emphasizes how simple it is to interact with a non-Java program (as long as it uses standard input and output).

Capturing the result from the C program is slightly more complicated. You must call **read( )** and provide a buffer where the results will be placed. The return value for **read( )** is the number of bytes that came from the C program, and if this value is -1 it means something is wrong. Otherwise, the **resultBuf** is turned into a **String** and the spaces are trimmed off. This string is then placed into a **DatagramPacket** as before, and shipped back to the same address which sent the request in the first place – notice that the sender's address is part of the **DatagramPacket** we received.

Remember that although the C program must be compiled on the Web server, the Java program can be compiled anywhere since the resulting byte codes will be the same regardless of the platform on which the program will be running.

the NameSender applet

As mentioned earlier, the applet must be written with Java 1.0 so that it will run on the available browsers, and this means that it's best if the number of classes produced is minimized. Thus, instead of using the **Dgram** class developed earlier, all the datagram manipulations will be placed in line.

```
//: NameSender.java
// An applet that sends an email address
// as a datagram, using Java 1.02.
import java.awt.*;
import java.applet.*;
import java.net.*;
import java.io.*;

public class NameSender extends Applet {
  Button send = new Button(
    "Add email address to mailing list");
  TextField t = new TextField(
    "type your email address here", 40);
  String str = new String();
  Label l = new Label(), l2 = new Label();
  DatagramSocket s;
  InetAddress hostAddress;
  byte buf[] = new byte[NameCollector.BUFFER_SIZE];
  DatagramPacket dp =
    new DatagramPacket(buf, buf.length);
  ReplyListener pl = null;
  int vcount = 0;
  public void init() {
    setLayout(new BorderLayout());
    Panel p = new Panel();
    p.setLayout(new GridLayout(2, 1));
    p.add(t);
    p.add(send);
    add("North", p);
    Panel labels = new Panel();
    labels.setLayout(new GridLayout(2, 1));
    labels.add(l);
    labels.add(l2);
    add("Center", labels);
    try {
      // Auto-assign port number:
      s = new DatagramSocket();
      hostAddress = InetAddress.getByName(
        getCodeBase().getHost());
    } catch(UnknownHostException e) {
      l.setText("Cannot find host");
    } catch(SocketException e) {
      l.setText("Can't open socket");
    }
    l.setText("Ready to send your email address");
  }
  public boolean action (Event evt, Object arg) {
    if(evt.target.equals(send)) {
      if(pl != null) {
        pl.stop();
        pl = null;
      }
      l2.setText("");
      // Check for errors in email name:
```

```
        str = t.getText().trim();
        if(str.indexOf(' ') != -1) {
          l.setText("Spaces not allowed in name");
          return true;
        }
        if(str.indexOf(',') != -1) {
          l.setText("Commas not allowed in name");
          return true;
        }
        if(str.indexOf('@') == -1) {
          l.setText("Name must include '@'");
          l2.setText("");
          return true;
        }
        // Everything's OK, so send the name.
        // Get a fresh buffer, so it's zeroed. For
        // some reason you must use a fixed size rather
        // than calculating the size dynamically:
        byte sbuf[] = new byte[NameCollector.BUFFER_SIZE];
        str.getBytes(0, str.length(), sbuf, 0);
        DatagramPacket toSend =
          new DatagramPacket(
            sbuf, 100, hostAddress,
            NameCollector.COLLECTOR_PORT);
        try {
          s.send(toSend);
        } catch(Exception e) {
          l.setText("Couldn't send datagram");
          return true;
        }
        l.setText("Sent: " + str);
        send.setLabel("Re-send");
        pl = new ReplyListener(this);
        l2.setText("Waiting for verification " + ++vcount);
      }
      else return super.action(evt, arg);
      return true;
    }
  }

  class ReplyListener extends Thread {
    NameSender ns;
    ReplyListener(NameSender ns) {
      this.ns = ns;
      start();
    }
    public void run() {
      try {
        ns.s.receive(ns.dp);
      } catch(Exception e) {
        ns.l2.setText("Couldn't receive datagram");
      }
      ns.l2.setText(new String(ns.dp.getData(),
        0, 0, ns.dp.getLength()));
    }
  } ///:~
```

The UI for the applet is quite simple. There's a **TextField** where you type your email address, and a **Button** to send the email address to the server. Two **Label**s are used to report status back to the user.

By now you can recognize the **DatagramSocket**, **InetAddress**, buffer and **DatagramPacket** as trappings of the network connection. Lastly you can see something called a **ReplyListener**, which is a class that's defined later in the file (alas, this applet will require loading two **.class** files across the Internet instead of just one). The **ReplyListener** is a thread whose job it is to listen for the reply sent back by the server.

The **init( )** method sets up the GUI with the familiar layout tools, then creates the **DatagramSocket** which will be used both for sending and receiving datagrams.

The **action( )** method (remember we're confined to Java 1.0 now, so we can't use any slick nested listener classes) only watches to see if you press the "send" button. When the button is pressed, the first action is to check the **ReplyListener pl** to see if it's **null**. If it's not **null**, there's a live **ReplyListener** running. The first time the message is sent a **ReplyListener** thread is started up to watch for the reply. Thus, if a **ReplyListener** is running, it means this is not the first time the user has tried to send the message. The old listener is stopped and the **pl** handle is set to **null**.

Regardless of whether this is the first time the button was pressed, the text in **l2** is erased.

The next group of statements checks the email name for errors. The **String.indexOf( )** method is used to search for illegal characters, and if one is found it is reported to the user. Notice that all this happens without any network activity, so it's fast and it doesn't bog down the Internet.

Once the name is verified, it is packaged into a datagram and sent to the host address and port number in the same way that was described in the earlier datagram example. The first label is changed to show you that the send has occurred, and the button text is changed so that it reads "re-send." At this point, the **ReplyListener** is started up and the second label informs you that the applet is waiting for a reply from the server. Now it's up to the user to wait for a reply, or to get impatient and press the button again. Because a thread is used to listen for the reply, the user still has full use of the UI.

The **ReplyListener** is quite a simple thread. In the constructor, it grabs a handle to the **NameSender** object that created it. The **run( )** method uses the **DatagramSocket** that lives in **NameSender** to **receive( )**, which blocks until the datagram packet comes from the server. The resulting packet is placed into **NameSender**'s **DatagramPacket dp**. The data is retrieved from the packet and placed into the second label in **NameSender**. At the point, the thread terminates and becomes dead. Of course, if the reply doesn't come back from the server in a reasonable amount of time, the user may become impatient and press the button again, thus terminating the current **ReplyListener** thread (and, after re-sending the data, starting a new one).

the Web page

Of course, the applet must go inside a Web page. This is the complete Web page; you can see that it's intended to be used to automatically collect names for my mailing list:

```
<HTML>
<HEAD>
<META CONTENT="text/html">
<TITLE>Add Yourself to Bruce Eckel's Java Mailing List</TITLE>
</HEAD>
<BODY LINK="#0000ff" VLINK="#800080" BGCOLOR="#ffffff">
<FONT SIZE=6><P>Add Yourself to Bruce Eckel's Java Mailing List</P></FONT>
The applet on this page will automatically add your email address
to the mailing list, so you will receive update information about
changes to the online version of "Thinking in Java," notification
when the book is in print, information about upcoming Java seminars,
and notification about the "Hands-on Java Seminar" Multimedia CD.
Type in your email address and press the button to automatically
add yourself to this mailing list.
<HR>
<applet code=NameSender width=400 height=100>
</applet>
<HR>
If after several tries, you do not get verification it means that
```

```
the Java application on the server is having problems. In this case,
you can add yourself to the list by sending email to
<A HREF="mailto:Bruce@EckelObjects.com">Bruce@EckelObjects.com</A>
</HTML>
```

The applet tag itself is quite trivial, no different from the very first one presented in the book.

problems with this approach

This certainly seems like an elegant approach. There's no CGI programming and so there are no delays while the server starts up a CGI program. The datagram approach seems to produce a nice quick response. In addition, when Java 1.1 is available everywhere, the server portion can be written entirely in Java (although it's quite interesting to see how easy it is to connect to a non-Java program using standard input and output).

But of course there are problems. One problem is rather subtle: since the Java application is running constantly on the server and it spends most of its time blocked in the **Datagram.receive( )** method, there *may* be some CPU hogging going on. At least, that's the way it appeared on the server where I was experimenting. On the other hand, there wasn't much else happening on that server, and starting the program using "nice" (a Unix program to prevent a process from hogging the CPU) or its equivalent may solve the problem if you have a more heavily-loaded server. In any event, it's worth keeping your eye on an application like this – a blocked **receive( )** may hog the CPU.

The second problem is much worse, and can be considered a show-stopper. It concerns firewalls. A firewall is a machine that sits between your network and the Internet. It monitors all traffic coming in from the Internet and going out to the Internet, and makes sure that traffic conforms to what it expects.

Firewalls are conservative little beasts. They demand strict conformance to all the rules, and if you're not conforming they assume you're doing something sinful, and shut you out (not quite so bad as the Spanish Inquisition, but close). For example, if you are on a network behind a firewall and you start connecting to the Internet using a Web browser, the firewall expects that all your transactions will connect to the server using the accepted http port, which is 80. Now here comes this Java applet **NameSender** which is trying to send a datagram to port 8080, which is way outside the range of the "protected" ports 0-1024. The firewall naturally assumes the worst, that someone has a virus, and so it doesn't allow the transaction to happen.

As long as someone has a raw connection to the Internet (for example, using a typical Internet Service Provider) there's no problem, but you may have some important customers dwelling behind firewalls, and they'll be shut out.

This is rather disheartening (it was for me, after I got what I considered a pretty slick little system running), because it essentially means you have to give up Java on the server and go back to the relatively primitive approach of writing CGI scripts in C or Perl (you can still submit the data using a Java applet, but you can also use a scripting language like JavaScript).

However, help is on the way. If everything goes as planned (Javasoft's plan), Web servers will be equipped with *servlet servers*. These will take a request from the client (going through the firewall-accepted port 80) and instead of starting up a CGI program, they will start up a Java program called a *servlet*. This is a little application that's only designed to run on the server. A servlet server will automatically start up the servlet to handle the client request, which means you can write all your programs only in Java (further enabling the "100% pure Java initiative"). It is admittedly an appealing idea, once you're comfortable with Java: you don't have to switch to a more primitive language to handle requests on the server.

Since it's only for handling requests on the server, the servlet API has no GUI abilities. This fits quite well with **NameCollector.java**, which doesn't have a GUI anyway.

At this writing, a low-cost servlet server was available from www.Javasoft.com. In addition, Javasoft is encouraging other Web server manufacturers to add servlet capabilities to their servers.

connecting to databases with JDBC

It has been estimated that half of all software development involves client/server operations. A great promise of Java has been the ability to build platform-independent client/server database applications. In Java 1.1 this has come to fruition with Java DataBase Connectivity (JDBC).

One of the major problems with databases has been the feature wars between the database companies. There is a "standard" database language, Structured Query Language (SQL-92), but usually you must know which database vendor you're working with despite the standard. JDBC is designed to be platform-independent, so you don't need to worry about which database you're using while you're programming. However, it's still possible to make vendor-specific calls from JDBC so you aren't restricted from doing what you must.

JDBC, like many of the APIs in Java, is designed for simplicity. The method calls you make correspond to the logical operations you'd think of doing when gathering data from a database: connect to the database, create a statement and execute the query, and look at the result set.

To allow this platform independence, JDBC provides a *driver manager* which dynamically maintains all the driver objects that your database queries will need. Thus, if you have three different kinds of vendor databases to connect to, you'll need three different driver objects. The driver objects register themselves with the driver manager at the time of loading, and you can force the loading using **Class.forName( )**.

To open a database, you must create a "database URL" that specifies:

1. That you're using JDBC with "jdbc"
2. The "subprotocol": the name of the driver or the name of a database connectivity mechanism. Since the design of JDBC was inspired by ODBC, the first subprotocol available is the "jdbc-odbc bridge," specified by "odbc"
3. The database identifier. This varies with the database driver used, but it generally provides a logical name that is mapped by the database administration software to a physical directory where the database tables are located. This means that for your database identifier to have any meaning, you must register the name using your database administration software (the process of registration varies from platform to platform).

All this information is combined together into one string, the "database URL." For example, to connect through the ODBC subprotocol to a database identified as "people," the database URL could be:

```
String dbUrl = "jdbc:odbc:people";
```

When you're ready to connect to the database, you call the **static** method **DriverManager.getConnection( )**, passing it the database URL, the user name and a password to get into the database. You get back a **Connection** object which you can then use to query and manipulate the database.

The following example opens a database of contact information and looks for a person's last name as given on the command line. It selects only the names of people that have email addresses, then prints out all the ones that match the given last name:

```
//: Lookup.java
// Looks up email addresses in a local database
// using JDBC
import java.sql.*;

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

```
    String dbUrl = "jdbc:odbc:people";
    String user = "";
    String password = "";
    try {
      // Load the driver (registers itself)
      Class.forName(
        "sun.jdbc.odbc.JdbcOdbcDriver");
      Connection c = DriverManager.getConnection(
        dbUrl, user, password);
      Statement s = c.createStatement();
      // SQL code:
      ResultSet r =
        s.executeQuery(
          "SELECT FIRST, LAST, EMAIL " +
          "FROM people.csv people " +
          "WHERE " +
          "(LAST='" + args[0] + "') " +
          " AND (EMAIL Is Not Null) " +
          "ORDER BY FIRST");
      while(r.next()) {
        // Capitalization doesn't matter:
        System.out.println(
          r.getString("Last") + ", "
          + r.getString("fIRST")
          + ": " + r.getString("EMAIL") );
      }
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
} ///:~
```

You can see the creation of the database URL as previously described. In this example, there is no password protection on the database so the user name and password are empty strings.

Once the connection is made with **DriverManager.getConnection( )**, you can use the resulting **Connection** object to create a **Statement** object using the **createStatement( )** method. With the resulting **Statement**, you can call **executeQuery( )**, passing in a string containing an SQL-92 standard SQL statement (you'll see shortly how you can generate this statement automatically, so you don't have to know much about SQL).

The **executeQuery( )** method returns a **ResultSet** object, which is quite a bit like an iterator: the **next( )** method moves the iterator to the next record in the statement, or returns **null** if the end of the result set has been reached. You'll always get a **ResultSet** object back from **executeQuery( )** even if a query results in an empty set (that is, an exception is not thrown). Note that you must call **next( )** once before trying to read any record data. If the result set is empty, this first call to **next( )** will return **false**. For each record in the result set, you can select the fields using (among other approaches) the field name as a string. Notice that the capitalization of the field name is ignored – it doesn't matter with an SQL database. You determine the type you'll get back by calling **getInt( )**, **getString( )**, **getFloat( )**, etc. At this point, you've got your database data in Java native format and can do whatever you want with it using ordinary Java code.

getting the example to work

With JDBC, understanding the code is relatively simple. The confusing part is making it work on your particular system. The reason this is confusing is that it requires you to figure out how to get your JDBC driver to load properly, and how to set up a database using your database administration software.

Of course, this process may vary radically from machine to machine, but the process I used to make it work under 32-bit Windows may give you clues to help you attack your own situation.

step 1: find the JDBC driver

The above program contains the statement:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

This implies a directory structure, which is deceiving. With this particular installation of JDK 1.1, there was no file called **JdbcOdbcDriver.class**, so if you looked at this example and went searching for it you'd be frustrated. Other published examples use a pseudo name, like "myDriver.ClassName," which is less than helpful. In fact, the load statement above for the jdbc-odbc driver (the only one that actually comes with JDK 1.1) only appears in a few places in the online documentation (in particular, a page labeled "JDBC-ODBC Bridge Driver"). If the above load statement doesn't work, then the name may have been changed as part of a Java version change so you should hunt through the documentation again.

If the load statement is wrong, you'll get an exception at this point. To test whether your driver load statement is working correctly, comment out the code after the statement and up to the **catch** clause; if the program throws no exceptions it means the driver is loading properly.

step 2: configure the database

Again, this is specific to 32-bit Windows; you may have to do some research to figure it out for your own platform.

First, open the control panel. You may find two icons that say "ODBC." You must use the one that says "32bit ODBC," since the other one is for backwards compatibility with 16-bit ODBC software and will produce no results for JDBC. When you open the "32bit ODBC" icon, you'll see a tabbed dialog with a number of tabs, including "User DSN," "System DSN," "File DSN," etc., where "DSN" means "Data Source Name." It turns out that for the JDBC-ODBC bridge, the only place where it's actually important to set up your database is "System DSN," but you'll also want to test your configuration and create queries, and for that you'll also need to set up your database in "File DSN." This will allow the Microsoft Query tool (that comes with Microsoft Office) to find the database. Note that other query tools are also available from other vendors.

The most interesting database is one that you're already using. Standard ODBC supports a number of different file formats including such venerable workhorses as DBase. However, it also includes the very simple "comma-separated ASCII" format, which virtually every data tool has the ability to write. In my case, I just took my "people" database which I've been maintaining for years using various contact-management tools and exported it as a comma-separated ASCII file (these typically have an extension of **.csv**). In the "File DSN" section I chose "Add," chose the text driver to handle my comma-separated ASCII file and then un-checked "use current directory" to allow me to specify the directory where I exported the data file.

You'll note when you do this that you don't actually specify a file, but only a directory. That's because a database is typically represented as a collection of files under a single directory (although it could be represented in other forms as well). Each file normally contains a single table, and the SQL statements can produce results which are culled from multiple tables in the database (this is called a *join*). A database which contains only a single table (like this one) is usually called a *flat-file database*. Most problems that go beyond the simple storage and retrieval of data generally require multiple tables which must be related by joins to produce the desired results, and these are called *relational* databases.

step 3: test the configuration

To test the configuration you'll need a way to discover whether the database is visible from a program that queries it. Of course, you can simply run the above JDBC program example up to and including the statement:

```
Connection c = DriverManager.getConnection(
  dbUrl, user, password);
```

If an exception is thrown, your configuration was incorrect.

However, it's useful to get a query-generation tool involved at this point. I used Microsoft Query which came with Microsoft Office, but you may have something else you prefer. The query tool must know where the database is, and Microsoft Query required that I go to the ODBC Administrator's "File DSN" tab and add a new entry there, again specifying the text driver and the directory where my database lives. You can name the entry anything you want, but it's helpful to use the same name you used in "System DSN."

Once you've done this, you will see that your database is available when you create a new query using your query tool.

step 4: generate your SQL query

The query that I created using Microsoft Query not only showed me that my database was there and in good order, but it also automatically created the SQL code that I needed to insert into my Java program. I wanted a query that would search for records that had the last name that was typed on the command line when starting the Java program. So as a starting point, I searched for a specific last name, 'Eckel'. I also only wanted to display names that had email addresses associated with them. The steps I took to create this query were:

1. Start a new query and use the Query Wizard. Select the "people" database (this is the equivalent of opening the database connection using the appropriate database URL).
2. Select the "people" table within the database. From within the table, choose the columns FIRST, LAST and EMAIL.
3. Under "Filter Data," choose LAST and select "equals" with an argument of Eckel. Click the "And" radio button.
4. Choose EMAIL and select "Is not Null."
5. Under "Sort By," choose FIRST.

The result of this query will show you whether you're getting what you want.

Now you can press the SQL button and without any research on your part, up will pop the correct SQL code, ready for you to cut and paste. For this query, it looked like this:

```
SELECT people.FIRST, people.LAST, people.EMAIL
FROM people.csv people
WHERE (people.LAST='Eckel') AND
(people.EMAIL Is Not Null)
ORDER BY people.FIRST
```

With more complicated queries it's very easy to get things wrong, but with a query tool you can interactively test your queries and automatically generate the correct code. It's hard to argue the case for doing this by hand.

step 5: modify and paste in your query

You'll notice that the above code looks different than what's used in the program. That's because the query tool uses full qualification for all the names, even when there's only one table involved (when more than one table is involved, the qualification prevents collisions between columns from different tables that have the same names). Since this query only involves one table, you can optionally remove the "people" qualifier from most of the names, like this:

```
SELECT FIRST, LAST, EMAIL
FROM people.csv people
WHERE (LAST='Eckel') AND
(EMAIL Is Not Null)
ORDER BY FIRST
```

In addition, you don't want this program to be hard coded to only look for one name. Instead it should hunt for the name given as the command-line argument. Making these changes and turning the SQL statement into a dynamically-created **String** produces:

```
"SELECT FIRST, LAST, EMAIL " +
"FROM people.csv people " +
"WHERE " +
"(LAST='" + args[0] + "') " +
" AND (EMAIL Is Not Null) " +
"ORDER BY FIRST");
```

SQL has another way to insert names into a query called *stored procedures*, which is used for speed. But for much of your database experimentation and for your first cut, building your own query strings in Java is fine.

You can see from this example that by using the tools currently available – in particular the query-building tool – database programming with SQL and JDBC can be quite straightforward.

a GUI version of the lookup program

It's more useful to leave the lookup program running all the time and simply switch to it and type in a name whenever you want to look someone up. The following program creates the lookup program as an application/applet, and it also adds name completion so the data will show up without forcing you to type the entire last name:

```
//: VLookup.java
// GUI version of Lookup.java
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.sql.*;

public class VLookup extends Applet {
  String dbUrl = "jdbc:odbc:people";
  String user = "";
  String password = "";
  Statement s;
  TextField searchFor = new TextField(20);
  Label completion =
    new Label("                         ");
  TextArea results = new TextArea(40, 20);
  public void init() {
    searchFor.addTextListener(new SearchForL());
    Panel p = new Panel();
    p.add(new Label("Last name to search for:"));
    p.add(searchFor);
    p.add(completion);
    setLayout(new BorderLayout());
    add(p, BorderLayout.NORTH);
    add(results, BorderLayout.CENTER);
    try {
      // Load the driver (registers itself)
      Class.forName(
        "sun.jdbc.odbc.JdbcOdbcDriver");
      Connection c = DriverManager.getConnection(
        dbUrl, user, password);
      s = c.createStatement();
    } catch(Exception e) {
      results.setText(e.getMessage());
    }
  }
  class SearchForL implements TextListener {
    public void textValueChanged(TextEvent te) {
      ResultSet r;
```

```
      if(searchFor.getText().length() == 0) {
        completion.setText("");
        results.setText("");
        return;
      }
      try {
        // Name completion:
        r = s.executeQuery(
          "SELECT LAST FROM people.csv people " +
          "WHERE (LAST Like '" +
          searchFor.getText()  +
          "%') ORDER BY LAST");
        if(r.next())
          completion.setText(
            r.getString("last"));
        r = s.executeQuery(
          "SELECT FIRST, LAST, EMAIL " +
          "FROM people.csv people " +
          "WHERE (LAST='" +
          completion.getText() +
          "') AND (EMAIL Is Not Null) " +
          "ORDER BY FIRST");
      } catch(Exception e) {
        results.setText(
          searchFor.getText() + "\n");
        results.append(e.getMessage());
        return;
      }
      results.setText("");
      try {
        while(r.next()) {
          results.append(
            r.getString("Last") + ", "
            + r.getString("fIRST") +
            ": " + r.getString("EMAIL") + "\n");
        }
      } catch(Exception e) {
        results.setText(e.getMessage());
      }
    }
  }
  static class WL extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
      System.exit(0);
    }
  }
  public static void main(String args[]) {
    VLookup applet = new VLookup();
    Frame aFrame = new Frame("Email lookup");
    aFrame.addWindowListener(new WL());
    aFrame.add(applet, BorderLayout.CENTER);
    aFrame.setSize(500,200);
    applet.init();
    applet.start();
    aFrame.setVisible(true);
  }
} ///:~
```

Much of the database logic is the same, but you can see that a **TextListener** is added to listen to the **TextField**, so that whenever you type a new character it first tries to do a name completion by looking

up the last name in the database and using the first one that shows up (it places it in the **completion Label**, and uses that as the lookup text). This way, as soon as you've typed enough characters for the program to uniquely find the name you're looking for, you can stop.

why the JDBC API seems so complex

When you browse the online documentation for JDBC it can seem very daunting. In particular, in the **DatabaseMetaData** interface – which is just huge, contrary to most of the interfaces you see in Java – there are methods like **dataDefinitionCausesTransactionCommit( )**, **getMaxColumnNameLength( )**, **getMaxStatementLength( )**, **storesMixedCaseQuotedIdentifiers( )**, **supportsANSI92IntermediateSQL( )**, **supportsLimitedOuterJoins( )**, and on and on. What's this all about?

As mentioned earlier, databases have seemed from their inception to be in a constant state of turmoil, primarily because the demand for database applications, and thus database tools, is so great. Only recently has there been any convergence on the common language of SQL (and there are plenty of other database languages in common use). But even with an SQL "standard" there are so many variations on that theme that JDBC must provide the large **DatabaseMetaData** interface so that your code can discover the capabilities of the particular "standard" SQL database that it's currently connected to. In short, you can write simple, transportable SQL but if you want to optimize speed your coding will multiply tremendously as you investigate the capabilities of the database.

This, of course, is not Java's fault. The discrepancies between database products are just something that JDBC tries to help compensate for, that's all. But bear in mind that your life will be easier if you can either write generic queries and not worry too much about performance, or, if you must tune for performance, know the platform you're writing for so you don't need to write all that investigation code.

There is more JDBC information available in the electronic documents that come as part of the Java 1.1 distribution from Javasoft. In addition, you can find more in the book *Java Database Programming with JDBC* (Patel & Moss, Coriolis Books 1997). Other JDBC books are appearing regularly.

remote methods

Traditional approaches to executing code on other machines across a network have been confusing as well as tedious and error-prone to implement. The nicest way to think about this problem is that some object happens to live on another machine, and that you can send a message to that object and get a result as if the object lived on your local machine. This simplification is exactly what Java 1.1 *Remote Method Invocation* (RMI) allows you to do. This section walks you through the steps necessary to create your own RMI objects.

remote interfaces

RMI makes heavy use of interfaces. When you want to create a remote object, you mask the underlying implementation by passing around an interface. Thus, when the client gets a handle to a remote object, what they really get is an interface handle, which *happens* to connect to some local stub code which itself talks across the network. But you don't think about this, you just send messages via your interface handle.

When you create a remote interface, you must follow these guidelines:

1. The remote interface must be **public** (that is, it cannot have "package access" e.g. "friendly"). Otherwise, a client will get an error when attempting to load a remote object that implements the remote interface.

2. The remote interface must extend the interface **java.rmi.Remote**.

3. Each method in the remote interface must declare **java.rmi.RemoteException** in its **throws** clause, in addition to any application-specific exceptions.

4. A remote object passed as an argument or return value (either directly or embedded within a local object) must be declared as the remote interface, not the implementation class.

Here's a simple remote interface that represents a very accurate time service:

```
//: PerfectTimeI.java
// The PerfectTime remote interface
package c15.PTime;
import java.rmi.*;

interface PerfectTimeI extends Remote {
  long getPerfectTime() throws RemoteException;
} ///:~
```

It looks like any other interface, except that it extends **Remote** and all its methods throw **RemoteException**. Remember that an **interface** and all its methods are automatically **public**.

implementing the remote interface

The server must contain a class that extends **UnicastRemoteObject** and implements the remote interface. This class may also have additional methods, but only the methods in the remote interface will be available to the client, of course, since the client will only get a handle to the interface, not the actual class that implements it.

You must explicitly define the constructor for the remote object, even if you're only defining a default constructor that just calls the base-class constructor. You must write it out since it must throw **RemoteException**.

Here's the implementation of the remote interface **PerfectTimeI**:

```
//: PerfectTime.java
// The implementation of the PerfectTime
// remote object
package c15.PTime;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.net.*;

public class PerfectTime
    extends UnicastRemoteObject
    implements PerfectTimeI {
  // Implementation of the interface:
  public long getPerfectTime()
      throws RemoteException {
    return System.currentTimeMillis();
  }
  // Must implement constructor to throw
  // RemoteException:
  public PerfectTime() throws RemoteException {
    // super(); // Called automatically
  }
  // Registration for RMI serving:
  public static void main(String args[]) {
    System.setSecurityManager(
      new RMISecurityManager());
    try {
      PerfectTime pt = new PerfectTime();
      Naming.bind(
        "//colossus:2005/PerfectTime", pt);
```

```
        System.out.println("Ready to do time");
      } catch(Exception e) {
        e.printStackTrace();
      }
    }
  } ///:~
```

Here, **main( )** handles all the details of setting up the server. When you're serving RMI objects, at some point in your program you must:

1. Create and install a security manager that supports RMI. The only one available for RMI as part of the Java distribution is **RMISecurityManager**.
2. Create one or more instances of a remote object. Here, you can see the creation of the **PerfectTime** object.
3. Register at least one of the remote objects with the RMI remote object registry, for bootstrapping purposes. One remote object may have methods that produce handles to other remote objects. This allows you to set it up so the client only has to go to the registry once, to get the first remote object.

setting up the registry

Here, you see a call to the **static** method **Naming.bind( )**. However, this call requires that the registry be running as a separate process on the computer. The name of the registry server is **rmiregistry**, and under 32-bit Windows you say:

```
start rmiregistry
```

to start it in the background. On Unix, it is

```
rmiregistry &
```

Like many network programs, the **rmiregistry** is located at the IP address of whatever machine started it up, but it must also be listening at a port. If you invoke the **rmiregistry** as above, with no argument, the registry's port will default to 1099. If you want it to be at some other port, you add an argument on the command line to specify the port. For this example, the port will be located at 2005, so the **rmiregistry** should be started like this under 32-bit Windows:

```
start rmiregistry 2005
```

or for Unix:

```
rmiregistry 2005 &
```

The information about the port must also be given to the **bind( )** command, as well as the IP address of the machine where the registry is located. But this brings up what can be a very frustrating problem if you're expecting to test RMI programs locally the way the network programs have been tested so far in this chapter. In the JDK 1.1.1 release, there are a couple of problems[2]:

1. **localhost** does not work with RMI. Thus, to experiment with RMI on a single machine, you must provide the actual name of the machine. To find out the actual name of your machine under 32-bit Windows, go to the control panel and select "Network." Select the "Identification" tab, and you'll see your computer name. In my case, I called my computer "Colossus" (for all the hard disks I've had to put on to hold all the different development systems). It appears that capitalization is ignored.
2. RMI will not work unless your computer has an active TCP/IP connection, even if all your components are just talking to each other on the local machine. This means you must

[2] Many brain cells died in agony to discover this information.

connect to your Internet service provider before trying to run the program, or you'll get some obscure exception messages.

Will all this in mind, the **bind( )** command becomes:

```
Naming.bind("//colossus:2005/PerfectTime", pt);
```

If you are using the default port 1099, you don't need to specify a port, so you could say:

```
Naming.bind("//colossus/PerfectTime", pt);
```

In a future release of the JDK (after 1.1) when the **localhost** bug is fixed, you will be able to perform local testing by leaving off the IP address and only putting in the identifier:

```
Naming.bind("PerfectTime", pt);
```

The name for the service is arbitrary; it happens to be PerfectTime here, just like the name of the class, but you could call it anything you want. The important thing is that it's a unique name in the registry that the client knows to look for to procure the remote object. If the name is already in the registry, you'll get an **AlreadyBoundException**. To prevent this, you can always use **rebind( )** instead of **bind( )**, since **rebind( )** either adds a new entry or replaces the one that's already there.

Even though **main( )** exits, your object has been created and registered so it's kept alive by the registry, waiting for a client to come along and request it. As long as the **rmiregistry** is running and you don't call **Naming.unbind( )** on your name, the object will be there. For this reason, when you're developing your code you need to shut down the **rmiregistry** and restart it when you compile a new version of your remote object.

You aren't forced to start up **rmiregistry** as an external process. If you know that your application is the only one that's going to use the registry, you can start it up inside your program with the line:

```
LocateRegistry.createRegistry(2005);
```

Where, as before, 2005 is the port number we happen to be using in this example. This is the equivalent of running **rmiregistry 2005** from a command line, but can often be more convenient when you're developing RMI code since it eliminates the extra steps of starting and stopping the registry. Once you've executed this code, you can **bind( )** using **Naming** as before.

creating stubs and skeletons

If you compile and run **PerfectTime.java**, it won't work even if you have the **rmiregistry** running correctly. That's because the framework for RMI isn't all there yet – you must first create the stubs and skeletons that provide the network connection operations and allow you to pretend that the remote object is just another local object on your machine.

What's going on under the covers is complex. Any objects that you pass into or return from a remote object must **implement Serializable**, so you can imagine that the stubs and skeletons are automatically performing serialization and deserialization as they "marshal" all the arguments across the network and return the result. Fortunately, you don't have to know about any of this, but you *do* have to create the stubs and skeletons. This is a simple process: you invoke the **rmic** tool on your compiled code, and it creates the necessary files. So the only requirement is that another step be added to your compilation process.

The **rmic** tool is particular about packages and classpaths, however. **PerfectTime.java** is in the package **c15.Ptime**, and even if you invoke **rmic** in the same directory where **PerfectTime.class** is located, **rmic** won't find the file since it searches the classpath. So you must specify the location off the class path, like so:

```
rmic c15.PTime.PerfectTime
```

You don't have to be in the directory containing **PerfectTime.class** when you execute this command, but the results will be placed in the current directory.

When **rmic** runs successfully, you'll have two new classes in the directory:

```
PerfectTime_Stub.class
PerfectTime_Skel.class
```

corresponding to the stub and skeleton. Now you're ready to get the server and client to talk to each other.

using the remote object

The whole point of RMI is to make the use of remote objects very simple. The only extra thing you must do in your client program is to look up and fetch the remote interface from the server. From then on, it's just regular Java programming: sending messages to objects. Here's the program that uses **PerfectTime**:

```
//: DisplayPerfectTime.java
// Uses remote object PerfectTime
package c15.PTime;
import java.rmi.*;
import java.rmi.registry.*;

public class DisplayPerfectTime {
  public static void main(String args[]) {
    System.setSecurityManager(
      new RMISecurityManager());
    try {
      PerfectTimeI t =
        (PerfectTimeI)Naming.lookup(
          "//colossus:2005/PerfectTime");
      for(int i = 0; i < 10; i++)
        System.out.println("Perfect time = " +
          t.getPerfectTime());
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
} ///:~
```

The ID string is the same as the one used to register the object with **Naming**, and the first part represents the URL and port number. Since you're using a URL, you can also specify a machine on the Internet.

What comes back from **Naming.lookup( )** must be cast to the remote interface, *not* the class itself. If you use the class instead, you'll get an exception.

You can see in the method call

```
t.getPerfectTime( )
```

that once you have a handle to the remote object, programming with it is indistinguishable from programming with a local object (with one difference: remote methods throw **RemoteException**).

alternatives to RMI

RMI is just one way to create objects that can be distributed across a network. It has the advantage of being a "pure Java" solution, but if you have a lot of code written in some other language, it may not meet your needs. The two most compelling alternatives are Microsoft's DCOM (which, according to Microsoft's plan, will eventually be hosted on platforms other than Windows) and CORBA, which is supported in Java 1.1 and was designed from the start to be cross-platform. You can get an in-depth treatment of distributed objects in Java (albeit with a clear bias towards CORBA) in *Client/Server Programming with Java and CORBA* by Orfali & Harkey (John Wiley & Sons, 1997).

summary

There's actually a lot more to networking than can be covered in this introductory treatment. Java networking also provides fairly extensive support for URLs, including protocol handlers for different types of content that can be discovered at an Internet site.

In addition, an up-and-coming technology is the *Servlet Server* which is an Internet server that uses Java to handle requests instead of the slow and rather awkward CGI (Common Gateway Interface) protocol. This means that to provide services on the server side you'll be able to write in Java instead of using some other language you may not know as well. In addition, you'll get the portability benefits of Java so you won't have to worry about the particular platform the server is hosted upon.

These and other features are fully and carefully described in *Java Network Programming* by Elliotte Rusty Harold (O'Reilly, 1997).

exercises

1. Compile and run the **JabberServer** and **JabberClient** programs in this chapter. Now edit the files to remove all the buffering for the input and output, then compile and run them again to observe the results.

2. Create a server that asks for a password, then opens a file and sends the file over the network connection. Create a client that connects to this server, gives the appropriate password, then captures and saves the file. Test the pair of programs on your machine using the **localhost** (the local loopback IP address **127.0.0.1** produced by calling **InetAddress.getByName(null)**).

3. Modify the server in exercise 2 so that it uses multithreading to handle multiple clients.

4. Modify **JabberClient** so that output flushing doesn't occur and observe the effect.

5. (More challenging) Create a client/server pair of programs that uses datagrams to transmit a file from one machine to the other (see the description at the end of the datagram section of this chapter).

6. (More challenging) Take the **VLookup.java** program and modify it so that when you click on the resulting name it automatically takes that name and copies it to the clipboard (so you can simply paste it in to your email). You'll have to look back at the IO stream chapter to remember how to use the new Java 1.1 clipboard.

16

16: design patterns

This chapter introduces the very important and yet non-traditional "patterns" approach to program design.

Probably the most important step forward in object-oriented design is the "design patterns" movement, chronicled in *Design Patterns*, by Gamma, Helm, Johnson & Vlissides (Addison-Wesley 1995)[1]. This book shows 23 different solutions to particular classes of problems. In this chapter, the basic concepts of design patterns will be introduced along with several examples. This should whet your appetite to read *Design Patterns*.

The latter part of the chapter contains an example of the design evolution process, starting with an initial solution and moving through the logic and process of evolving the solution to more appropriate designs. The program shown (a trash sorting simulation) has evolved over time, and you can look at that evolution as a prototype for the way your own design can start as an adequate solution to a particular problem and evolve into a flexible approach to a class of problems.

[1] But be warned: the examples are in C++.

the pattern concept

Initially, you can think of a pattern as an especially clever and insightful way of solving a particular class of problems. That is, it looks like a lot of people have worked out all the angles of a problem and have come up with the most general, flexible solution for it. The problem may actually be one you have seen and solved before, but your solution probably didn't have the kind of completeness you'll see embodied in a pattern.

Although they're called "design patterns," they really aren't tied to the realm of design. A pattern actually seems to stand apart from the traditional way of thinking about analysis, design and implementation. Instead, a pattern embodies a complete idea within a program, and thus it can sometimes appear at the analysis phase or high-level design phase. This is interesting because a pattern has a direct implementation in code and so you might not expect it to show up before low-level design or implementation (and in fact you may not realize you need a particular pattern until you get to those phases).

The basic concept of a pattern can also be seen as the basic concept of program design: adding a layer of abstraction. Whenever you abstract something you're isolating particular details, and one of the most compelling motivations behind this is to *separate things that change from things that stay the same*. Another way to put this is that once you find some part of your program that's likely to change for one reason or another, you'll want to keep those changes from propagating other changes throughout your code. Not only does this make the code much cheaper to maintain, but it turns out that it is generally simpler to understand (which also results in lowered costs). The goal of design patterns is to isolate changes in your code.

If you look at it this way, you've actually been seeing some design patterns already in this book. For example, inheritance can be thought of as a design pattern (albeit one implemented by the compiler). It allows you to express differences in behavior (that's the thing that changes) in objects that all have the same interface (that's what stays the same). Composition can also be considered a pattern, since it allows you to change – dynamically or statically – the objects that implement your class, and thus the way that class works.

You've also already seen another pattern that appears in *Design Patterns*: the *iterator*, which Java capriciously calls the **Enumeration**. This hides the particular implementation of the container as you're stepping through and selecting the elements one by one. The iterator allows you to write generic code that performs an operation on all the elements in a sequence without regard to the way that sequence is built. Thus your generic code can be used with any container that can produce an iterator.

the singleton

Possibly the simplest design pattern is the *singleton*, which is a way to provide one and only one instance of an object. This has also been seen in the Java libraries, but here's a more direct example:

```
//: SingletonPattern.java
// The Singleton design pattern: you can
// only ever have one.

final class Singleton { // final: can't clone
  private static Singleton s = new Singleton(47);
  private int i;
  private Singleton(int x) { i = x; }
  public static Singleton getHandle() {
    return s;
  }
  public int getValue() { return i; }
  public void setValue(int x) { i = x; }
}

public class SingletonPattern {
```

```
  public static void main(String[] args) {
    Singleton s = Singleton.getHandle();
    System.out.println(s.getValue());
    Singleton s2 = Singleton.getHandle();
    s2.setValue(9);
    System.out.println(s.getValue());
    try {
      // Can't do this: compile-time error.
      // Singleton s3 = (Singleton)s2.clone();
    } catch(Exception e) {}
  }
} ///:~
```

The key to creating a singleton is to prevent the client programmer from having any way to create an object except the ways you provide. Thus, you must make all constructors **private**, and you must create at least one constructor to prevent the compiler from synthesizing a default constructor for you (which it will create as "friendly").

At this point, you decide how you're going to create your object. Here, it's created statically, but you can also wait until the client programmer asks for one and create it on demand. In any case, the object should be stored privately. You provide access through public methods. Here, **getHandle( )** produces the handle to the **Singleton** object. The rest of the interface (**getValue( )** and **setValue( )**) is the normal class interface.

Java also allows the creation of objects through cloning. In this example, making the class **final** prevents cloning. Since **Singleton** is inherited directly from **Object**, the **clone( )** method remains **protected** so it cannot be used (doing so produces a compile-time error). If you're inheriting from a class hierarchy that has already redefined **clone( )** as **public** and implemented **Cloneable**, you must override **clone( )** and throw a **CloneNotSupportedException** as described in Chapter 12.

Note that you aren't restricted to only creating one object. This is also a technique to create a limited pool of objects. In that situation, however, you may be confronted with the problem of sharing objects in the pool. If this is an issue, you can create a solution involving a check-out and check-in of the shared objects.

classifying patterns

In *Design Patterns*, there are 23 different patterns classified under three purposes (all of which revolve around the particular aspect that can vary). The three purposes are:

1. **Creational**: how an object can be created. This often involves isolating the details of object creation so your code isn't dependent on what types of objects there are and thus doesn't have to be changed when you add a new type of object. The aforementioned *Singleton* is classified as a creational pattern, and later in this chapter you'll see examples of the *Factory Method* and *Prototype*.

2. **Structural**: designing objects to satisfy particular project constraints. These work with the way objects are connected with other objects to ensure that changes in the system don't require changes to those connections.

3. **Behavioral**: objects that handle particular types of actions within a program. These encapsulate processes that you want to perform, like interpreting a language, fulfilling a request, moving through a sequence (as in an **Enumeration**) or implementing an algorithm. This chapter contains examples of the *Observer* and the *Visitor* patterns.

The *Design Patterns* book has a section on each of the 23 patterns along with an example, typically in C++ but sometimes in Smalltalk (you'll find this doesn't matter too much since you can easily translate the concepts from either language into Java). In this book I will not attempt to repeat all the patterns shown in *Design Patterns* since that book stands on its own and should be studied by the reader separately. Instead, this chapter will give some examples that should provide you with a decent feel for what patterns are about and why they are so important.

the observer pattern

The observer pattern solves a fairly common problem: what if a group of objects needs to update themselves when some object changes state? This can be seen in the "model-view" aspect of Smalltalk's MVC (model-view-controller), or the "document-view architecture." Suppose you have some data (the "document") and more than one view, say a graph and a textual view. When you change the data, the two views must know to update themselves, and that's what the observer facilitates. It's a common enough problem that it's part of the standard **java.util** library.

There are two types of objects used to implement the observer pattern in Java. The **Observable** class keeps track of everybody who wants to be informed when a change happens, and whether the "state" has changed or not. When someone says "OK, everybody should check and potentially update themselves," the **Observable** class performs this task by calling the **notifyObservers( )** method for each one on the list. The **notifyObservers( )** method is part of the base class **Observable**.

There are actually two "things that change" in the observer pattern: the quantity of observing objects, and the way an update occurs. That is, the observer pattern allows you to modify both of these without affecting the surrounding code.

The following example is similar to the **ColorBoxes** example from Chapter 14. Boxes are placed in a grid on the screen and each one is initialized to a random color. In addition, each box **implements** the **Observer** interface and is registered with an **Observable** object. When you click on a box, all the other boxes are notified that a change has been made because the **Observable** object automatically calls each **Observer** object's **update( )** method. Inside this method, the box checks to see if it's adjacent to the one that was clicked, and if so it changes its color to match the clicked box.

```
//: BoxObserver.java
// Demonstration of Observer pattern using
// Java's built-in observer classes.
import java.awt.*;
import java.awt.event.*;
import java.util.*;

// You must inherit a new type of Observable:
class BoxObservable extends Observable {
  public void notifyObservers(Object b) {
    // Otherwise it won't propagate changes:
    setChanged();
    super.notifyObservers(b);
  }
}

public class BoxObserver extends Frame {
  Observable notifier = new BoxObservable();
  public BoxObserver(int grid) {
    setTitle("Demonstrates Observer pattern");
    setLayout(new GridLayout(grid, grid));
    for(int x = 0; x < grid; x++)
      for(int y = 0; y < grid; y++)
        add(new OCBox(x, y, notifier));
  }
  static class WL extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
      System.exit(0);
    }
  }
  public static void main(String[] args) {
    int grid = 8;
    if(args.length > 0)
```

```
      grid = Integer.parseInt(args[0]);
    Frame f = new BoxObserver(grid);
    f.setSize(500, 400);
    f.setVisible(true);
    f.addWindowListener(new WL());
  }
}

class OCBox extends Canvas implements Observer {
  Observable notifier;
  int x, y; // Locations in grid
  Color cColor = newColor();
  static final Color colors[] = {
    Color.black, Color.blue, Color.cyan,
    Color.darkGray, Color.gray, Color.green,
    Color.lightGray, Color.magenta,
    Color.orange, Color.pink, Color.red,
    Color.white, Color.yellow
  };
  static final Color newColor() {
    return colors[
      (int)(Math.random() * colors.length)
    ];
  }
  OCBox(int x, int y, Observable notifier) {
    this.x = x;
    this.y = y;
    notifier.addObserver(this);
    this.notifier = notifier;
    addMouseListener(new ML());
  }
  public void paint(Graphics  g) {
    g.setColor(cColor);
    Dimension s = getSize();
    g.fillRect(0, 0, s.width, s.height);
  }
  class ML extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
      notifier.notifyObservers(OCBox.this);
    }
  }
  public void update(Observable o, Object arg) {
    OCBox clicked = (OCBox)arg;
    if(nextTo(clicked)) {
      cColor = clicked.cColor;
      repaint();
    }
  }
  private final boolean nextTo(OCBox b) {
    return Math.abs(x - b.x) <= 1 &&
           Math.abs(y - b.y) <= 1;
  }
} ///:~
```

When you first look at the online documentation for **Observable**, it's a bit confusing because it appears that you can just use an ordinary **Observable** object to manage the updates. But this doesn't work (try it – inside **BoxObserver**, create an **Observable** object instead of a **BoxObservable** object and see what happens: nothing). To get an effect, you *must* inherit from **Observable** and somewhere in your derived-class code call **setChanged( )**. This is the method that sets the "changed" flag, which means that when you call **notifyObservers( )** all the observers will, in fact, get notified. In the above

example **setChanged( )** is simply called within **notifyObservers( )**, but you could use any criterion you want to decide when to call **setChanged( )**.

BoxObserver contains a single **Observable** object called **notifier**, and every time an **OCBox** object is created, it is tied to **notifier**. In **OCBox**, whenever you click the mouse the **notifyObservers( )** method is called, passing the clicked object in as an argument so that all the boxes receiving the message (in their **update( )** method) know who was clicked and can decide whether to change themselves or not. Using a combination of code in **notifyObservers( )** and **update( )** you can work out some fairly complex schemes.

It may appear that the way that observers are notified must be frozen at compile time in the **notifyObservers( )** method. However, if you look more closely at the above code you'll see that the only place in **BoxObserver** or **OCBox** that you know you're working with a **BoxObservable** is at the point of creation of the **Observable** object – from then on everything just uses the basic **Observable** interface. This means that you could inherit other **Observable** classes and swap them at run-time if you want to change notification behavior.

simulating the trash recycler

The nature of this problem is that the trash is thrown unclassified into a single bin, so the specific type information is lost. But later, the specific type information must be recovered to properly sort the trash. In the initial solution, RTTI (described in Chapter 11) is used.

This is not a trivial design because it has an added constraint. That's what makes it interesting – it's more like the messy problems you're likely to encounter in your work. The extra constraint is that the trash arrives at the trash recycling plant all mixed together. The program must model the sorting of that trash. This is where RTTI comes in: you have a bunch of anonymous pieces of trash, and the program figures out exactly what type they are.

```
//: RecycleA.java
// Recycling with RTTI
package c16.RecycleA;
import java.util.*;
import java.io.*;

abstract class Trash {
  private double weight;
  Trash(double wt) { weight = wt; }
  abstract double value();
  double weight() { return weight; }
  // Sums the value of Trash in a bin:
  static void sumValue(Vector bin) {
    Enumeration e = bin.elements();
    double val = 0.0f;
    while(e.hasMoreElements()) {
      // One kind of RTTI:
      // A dynamically-checked cast
      Trash t = (Trash)e.nextElement();
      // Polymorphism in action:
      val += t.weight() * t.value();
      System.out.println(
        "weight of " +
        // Using RTTI to get type
        // information about the class:
        t.getClass().getName() +
        " = " + t.weight());
    }
    System.out.println("Total value = " + val);
  }
```

```
}

class Aluminum extends Trash {
  static double val  = 1.67f;
  Aluminum(double wt) { super(wt); }
  double value() { return val; }
  static void value(double newval) {
    val = newval;
  }
}

class Paper extends Trash {
  static double val = 0.10f;
  Paper(double wt) { super(wt); }
  double value() { return val; }
  static void value(double newval) {
    val = newval;
  }
}

class Glass extends Trash {
  static double val = 0.23f;
  Glass(double wt) { super(wt); }
  double value() { return val; }
  static void value(double newval) {
    val = newval;
  }
}

public class RecycleA {
  public static void main(String args[]) {
    Vector bin = new Vector();
    // Fill up the Trash bin:
    for(int i = 0; i < 30; i++)
      switch((int)(Math.random() * 3)) {
        case 0 :
          bin.addElement(new
            Aluminum(Math.random() * 100));
          break;
        case 1 :
          bin.addElement(new
            Paper(Math.random() * 100));
          break;
        case 2 :
          bin.addElement(new
            Glass(Math.random() * 100));
      }
    Vector
      glassBin = new Vector(),
      paperBin = new Vector(),
      alBin = new Vector();
    Enumeration sorter = bin.elements();
    // Sort the Trash:
    while(sorter.hasMoreElements()) {
      Object t = sorter.nextElement();
      // RTTI to show class membership:
      if(t instanceof Aluminum)
        alBin.addElement(t);
      if(t instanceof Paper)
        paperBin.addElement(t);
```

```
        if(t instanceof Glass)
          glassBin.addElement(t);
      }
      Trash.sumValue(alBin);
      Trash.sumValue(paperBin);
      Trash.sumValue(glassBin);
      Trash.sumValue(bin);
    }
} ///:~
```

The first thing you'll notice is the **package** statement:

```
package c16.RecycleA;
```

What this means is that in the source code listings available for the book, this file will be placed in the subdirectory **RecycleA** which branches off from the subdirectory **c16** (for Chapter 16). The unpacking tool in Chapter 17 takes care of placing it in the right subdirectory. The reason for doing this is that this chapter rewrites this particular example a number of times and by putting each version in its own **package** the class names will not clash.

The program uses the classic structure of methods in the base class that are redefined in the derived class. Those redefined methods are called polymorphically from a base-class handle.

Several **Vector** objects are created to hold **Trash** handles. Of course, **Vector**s actually hold **Object**s so they'll hold anything at all. The reason they hold **Trash** (or something derived from **Trash**) is only because you've been careful not to put in anything except **Trash**. If you do put something "wrong" into the **Vector**, you won't get any compile-time warnings or errors – you'll only find out via an exception at run-time.

When the **Trash** handles are added, they lose their specific identities and become simply **Object** handles (they are *upcast*). However, because of polymorphism the proper behavior still occurs when the dynamically-bound methods are called through the **Enumeration sorter**, once the resulting **Object** has been cast back to **Trash**. **sumValue( )** also uses an **Enumeration** to perform operations on every object in the **Vector**.

It looks silly to upcast the types of **Trash** into a container holding base type handles, and then turn around and downcast. Why not just put the Trash into the appropriate receptacle in the first place? (Indeed, this is the whole enigma of recycling). In this program it would be easy to repair, but sometimes a system's structure and flexibility can benefit greatly from downcasting.

The program satisfies the design requirements: it works. This may be fine as long as it's a one-shot solution. However, a useful program tends to evolve over time, so you must ask: what if the situation changes? For example, cardboard is now a valuable recyclable commodity, so how will that be integrated into the system (especially if the program is large and complicated). Since the above type-check coding in the **switch** statement could be scattered throughout the program, you'll have to go find all that code every time a new type is added, and if you miss one the compiler will not give you any help by pointing out an error.

The key to the misuse of RTTI here is that *every type is tested*. If you're only looking for a subset of types because that subset needs special treatment, that's probably fine. But if you're hunting for every type inside a switch statement, then you're probably missing an important point, and definitely making your code less maintainable. In the next section we'll look at how this program evolved over several stages to become much more flexible. This should prove a valuable example in program design.

improving the design

The solutions in *Design Patterns* are organized around the question "what will change as this program evolves?" This is generally the most important question you can ask about any design. If you can build your system around the answer, the results will be two-pronged: not only will your system allow easy (and inexpensive) maintenance, but you may also produce components that are reusable, so that

other systems can be built more cheaply. This is the promise of object-oriented programming, but it doesn't happen automatically – it requires thought and insight on your part. In this section we'll look at how this process can happen during the refinement of a system.

The answer to the question "what will change" for the recycling system is a common one: more types will be added to the system. The goal of the design, then, is to make this addition of types as painless as possible. In the recycling program, we'd like to encapsulate all places where specific type information is mentioned, so (if for no other reason) any changes can be localized to those encapsulations. It turns out this process also cleans up the rest of the code considerably.

"make more objects"

This brings up a general object-oriented design principle which I first heard spoken by Grady Booch: "if the design is too complicated, make more objects." This is simultaneously counterintuitive and ludicrously simple, and yet it's the most useful guideline I've found. In general, if you find a place with messy code, consider what sort of class would clean that up. Very often the side effect of cleaning up the code will be a system that has better structure and is more flexible.

Consider first the place where **Trash** objects are created, which is a **switch** statement inside **main( )**:

```
for(int i = 0; i < 30; i++)
  switch((int)(Math.random() * 3)) {
    case 0 :
      bin.addElement(new
        Aluminum(Math.random() * 100));
      break;
    case 1 :
      bin.addElement(new
        Paper(Math.random() * 100));
      break;
    case 2 :
      bin.addElement(new
        Glass(Math.random() * 100));
  }
```

This is definitely messy, and also a place where you must change code whenever a new type is added. If new types are commonly added, a better solution is a single method that takes all the necessary information and produces a handle to an object of the correct type, already upcast to a trash object. In *Design Patterns* this is broadly referred to as a *creational pattern* (of which there are several). The specific pattern that will be applied here is a variant of the *Factory Method*. Here, the factory method is a **static** member of **Trash**, but more commonly it may be a method that is redefined in the derived class.

The idea of the factory method is that you pass it the essential information it needs to know to create your object, then stand back and wait for the handle (already upcast to the base type) to pop out as the return value. From then on, you treat the object polymorphically. Thus, you never even need to know the exact type of object that's created. In fact, the factory method hides it from you to prevent accidental misuse. That is, if you want to use the object without polymorphism, you'll have to explicitly use a cast.

But there's a little problem, especially when you use the more complicated approach (not shown here) of making the factory method in the base class and overriding it in the derived classes. What if the information required in the derived class requires more or different arguments? "Creating more objects" solves this problem. To implement the factory method, the **Trash** class gets a new method called **factory**. To hide the creational data, there's a new class called **Info** that contains all the necessary information for the **factory** method to create the appropriate **Trash** object. Here's a simple implementation of **Info**:

```
class Info {
  int type;
  // Must change this to add another type:
```

```
  static final int maxNum = 4;
  double data;
  Info(int TypeNum, double Data) {
    type = TypeNum % maxNum;
    data = Data;
  }
}
```

An **Info** object's only job is to hold information for the **factory( )** method. Now if there's a situation where **factory( )** needs more or different information to create a new type of **Trash** object, the **factory( )** interface doesn't need to be changed. The **Info** class can be changed by adding new data and new constructors, or in the more typical object-oriented fashion of subclassing.

The **factory( )** method for this simple example looks like this:

```
static Trash factory(Info i) {
  switch(i.type) {
    default: // to quiet the compiler
    case 0:
      return new Aluminum(i.data);
    case 1:
      return new Paper(i.data);
    case 2:
      return new Glass(i.data);
    case 3: // two lines here: (*1*)
      return new Cardboard(i.data);
  }
}
```

Here, the determination of the exact type of object is very simple, but you can imagine a more complicated system where **factory( )** uses an elaborate algorithm. The point is that it's now hidden away in one place, and you know to come to this place when you add new types.

The creation of new objects is now much simpler in **main( )**:

```
  for(int i = 0; i < 30; i++)
    bin.addElement(
      Trash.factory( // (*8*)
        new Info(
          (int)(Math.random() * Info.maxNum),
          Math.random() * 100)));
```

An **Info** object is created to pass the data into **factory( )**, which in turn produces some kind of **Trash** object on the heap and returns the handle that's added to the **Vector bin**. Of course, if you change the quantity and type of argument this statement will still need to be modified, but that can be eliminated if the creation of the **Info** object is itself automated. For example, a **Vector** of arguments can be passed into the constructor of an **Info** object (or directly into a **factory( )** call, for that matter). This requires that the arguments be parsed and checked at runtime, but it does provide the greatest flexibility.

a pattern for prototyping creation

A problem with the above design is that it still requires a central location where all the types of the objects must be known: inside the **factory( )** method. If new types are regularly being added to the system, the **factory( )** method must be changed for each new type. When you discover something like this, it is useful to try to go one step further and move *all* the information about the type – including its creation – into the class representing that type. This way, the only thing you need to do to add a new type to the system is to inherit a single class.

To move the information about type creation into each specific type of **Trash**, the "prototype" pattern will be used. The general idea is that you have a master sequence of objects, one of each type you're interested in making. The objects in this sequence are *only* used for making new objects, using an

operation that's not unlike the **clone( )** scheme built into Java's root class **Object**. In this case, the cloning method is called **tClone( )**. When you're ready to make a new object, presumably you have some sort of information that establishes the type of object you want to create, then you move through the master sequence comparing your information with whatever appropriate information is in the prototype objects in the master sequence. When you find one that matches your needs, you clone it.

In this scheme there is no hard-coded information for creation. Each object knows how to expose appropriate information and how to clone itself. Thus the **factory( )** method doesn't need to be changed when a new type is added to the system.

One approach to the problem of prototyping is to add a number of methods to support the creation of new objects. However, in Java 1.1 there's already support for creating new objects if you have a handle to the **Class** object. With Java 1.1 *reflection* (introduced in Chapter 11) you can call a constructor even if you only have a handle to the **Class** object. This is the perfect solution for the prototyping problem.

The list of prototypes will be represented indirectly by a list of handles to all the **Class** objects you want to create. In addition, if the prototyping fails, the **factory( )** method will assume that it's because a particular **Class** object wasn't in the list, and will attempt to load it. By loading the prototypes dynamically like this, the **Trash** class doesn't need to know what types it is working with and so doesn't need any modifications when you add new types. This allows it to be easily reused throughout the rest of the chapter.

```
//: Trash.java
// Base class for Trash recycling examples
package c16.Trash;
import java.util.*;
import java.lang.reflect.*;

public abstract class Trash {
  private double weight;
  Trash(double wt) { weight = wt; }
  Trash() {}
  public abstract double value();
  public double weight() { return weight; }
  // Sums the value of Trash in a bin:
  public static void sumValue(Vector bin) {
    Enumeration e = bin.elements();
    double val = 0.0f;
    while(e.hasMoreElements()) {
      // One kind of RTTI:
      // A dynamically-checked cast
      Trash t = (Trash)e.nextElement();
      val += t.weight() * t.value();
      System.out.println(
        "weight of " +
        // Using RTTI to get type
        // information about the class:
        t.getClass().getName() +
        " = " + t.weight());
    }
    System.out.println("Total value = " + val);
  }
  // Remainder of class provides support for
  // prototyping:
  public static class PrototypeNotFoundException
      extends Exception {}
  public static class CannotCreateTrashException
      extends Exception {}
  private static Vector TrashTypes =
    new Vector();
```

```
  public static Trash factory(Info info)
      throws PrototypeNotFoundException,
      CannotCreateTrashException {
    for(int i = 0; i < TrashTypes.size(); i++) {
      // Somehow determine the new type
      // to create, and create one:
      Class tc =
        (Class)TrashTypes.elementAt(i);
      if (tc.getName().indexOf(info.id) != -1) {
        try {
          // Get the dynamic constructor method
          // that takes a double argument:
          Constructor ctor =
            tc.getConstructor(
              new Class[] {double.class});
          // Call the constructor to create a
          // new object:
          return (Trash)ctor.newInstance(
            new Object[]{new Double(info.data)});
        } catch(Exception ex) {
          ex.printStackTrace();
          throw new CannotCreateTrashException();
        }
      }
    }
    // Class was not in the list. Try to load it,
    // but it must be in your class path!
    try {
      System.out.println("Loading " + info.id);
      TrashTypes.addElement(
        Class.forName(info.id));
    } catch(Exception e) {
      e.printStackTrace();
      throw new PrototypeNotFoundException();
    }
    // Loaded successfully. Recursive call
    // should work this time:
    return factory(info);
  }
  public static class Info {
    public String id;
    public double data;
    public Info(String name, double data) {
      id = name;
      this.data = data;
    }
  }
} ///:~
```

The basic **Trash** class and **sumValue( )** are the same as before. The rest of the class supports the prototyping pattern. You can first see two inner classes (which are **static**, so they are inner classes only for code organization) describing exceptions that can occur. This is followed by a **Vector TrashTypes** which is used to hold the **Class** handles.

In **Trash.factory( )**, the **String** inside the **Info** object **i** (a different version of the **Info** class than in the prior discussion) contains the type name of the **Trash** to be created; this **String** is compared to the **Class** names in the list. If there's a match, then that's the object to create. Of course, there are many ways to determine what object you want to make. This one is used so that information read in from a file can be turned into objects.

Once you've discovered which kind of **Trash** to create, then the reflection methods come into play. The **getConstructor( )** method takes an argument that is an array of **Class** handles. This array represents the arguments, in their proper order, for the constructor that you're looking for. Here, the array is dynamically created using the Java 1.1 array-creation syntax:

```
new Class[] {double.class}
```

This code assumes that every **Trash** type has a constructor that takes a **double** (and note that **double.class** is distinct from **Double.class**). It's also possible, for a more flexible solution, to call **getConstructors( )** which returns an array of the possible constructors.

What comes back from **getConstructor( )** is a handle to a **Constructor** object (part of **java.lang.reflect**). You call the constructor dynamically with the method **newInstance( )**, which takes an array of **Object** containing the actual arguments. This array is again created using the new Java 1.1 syntax:

```
new Object[]{new Double(info.data)}
```

In this case, however, the **double** must be placed inside a wrapper class so it can be part of this array of objects. The process of calling **newInstance( )** extracts the **double**, but you can see it is a bit confusing – an argument might be a **double** or a **Double**, but when you make the call you must always pass in a **Double**. Fortunately this issue only exists for the primitive types.

Once you understand how to do it, the process of creating a new object given only a **Class** handle is remarkably simple. Reflection also allows you to call methods in this same dynamic fashion.

Of course, the appropriate **Class** handle may not be in the **TrashTypes** list. In this case, the **return** in the inner loop is never executed and you'll drop out at the end. Here, the program tries to rectify the situation by loading the **Class** object dynamically and adding it to the **TrashTypes** list. If it still can't be found something is really wrong, but if the load is successful then the **factory** method is called recursively to try again.

As you'll see, the beauty of this design is that this code doesn't need to be changed, regardless of the different situations where it will be used (assuming all **Trash** subclasses contain a constructor that takes a single **double** argument).

Trash subclasses

To fit into the prototyping scheme, the only thing that's required of each new subclass of **Trash** is that it contains a constructor that takes a **double** argument. Java 1.1 reflection handles everything else.

Here are the different types of **Trash**, each in their own file but part of the **Trash** package (again, to facilitate reuse within the chapter):

```
//: Aluminum.java
// The Aluminum class with prototyping
package c16.Trash;

public class Aluminum extends Trash {
  private static double val = 1.67f;
  public Aluminum(double wt) { super(wt); }
  public double value() { return val; }
  public static void value(double newVal) {
    val = newVal;
  }
} ///:~

//: Paper.java
// The Paper class with prototyping
package c16.Trash;

public class Paper extends Trash {
  private static double val = 0.10f;
```

```
  public Paper(double wt) { super(wt); }
  public double value() { return val; }
  public static void value(double newVal) {
    val = newVal;
  }
} ///:~

//: Glass.java
// The Glass class with prototyping
package c16.Trash;

public class Glass extends Trash {
  private static double val = 0.23f;
  public Glass(double wt) { super(wt); }
  public double value() { return val; }
  public static void value(double newVal) {
    val = newVal;
  }
} ///:~
```

And here's a new type of **Trash**:

```
//: Cardboard.java
// The Cardboard class with prototyping
package c16.Trash;

public class Cardboard extends Trash {
  private static double val = 0.23f;
  public Cardboard(double wt) { super(wt); }
  public double value() { return val; }
  public static void value(double newVal) {
    val = newVal;
  }
} ///:~
```

You can see that, other than the constructor, there's nothing special about any of these classes.

parsing Trash from an external file

The information about **Trash** objects will be read from an outside file. The file has information about each piece of Trash on a single line in the form **Trash:weight**, such as:

```
c16.Trash.Glass:54
c16.Trash.Paper:22
c16.Trash.Paper:11
c16.Trash.Glass:17
c16.Trash.Aluminum:89
c16.Trash.Paper:88
c16.Trash.Aluminum:76
c16.Trash.Cardboard:96
c16.Trash.Aluminum:25
c16.Trash.Aluminum:34
c16.Trash.Glass:11
c16.Trash.Glass:68
c16.Trash.Glass:43
c16.Trash.Aluminum:27
c16.Trash.Cardboard:44
c16.Trash.Aluminum:18
c16.Trash.Paper:91
c16.Trash.Glass:63
c16.Trash.Glass:50
c16.Trash.Glass:80
```

```
c16.Trash.Aluminum:81
c16.Trash.Cardboard:12
c16.Trash.Glass:12
c16.Trash.Glass:54
c16.Trash.Aluminum:36
c16.Trash.Aluminum:93
c16.Trash.Glass:93
c16.Trash.Paper:80
c16.Trash.Glass:36
c16.Trash.Glass:12
c16.Trash.Glass:60
c16.Trash.Paper:66
c16.Trash.Aluminum:36
c16.Trash.Cardboard:22
```

Note that the class path must be respected when giving the class names, otherwise the class will not be found.

To parse this, the line is read in and the **String** method **indexOf( )** is used to produce the index of the '**:'.** This is first used with the **String** method **substring( )** to extract the name of the Trash type, and next to get the weight which is turned into a **double** with the **static Double.valueOf( )** method. The **trim( )** method removes white space at both ends of a string.

The **Trash** parser is placed in a separate file since it will be reused throughout this chapter:

```
//: ParseTrash.java
// Open a file and parse its contents into
// Trash objects, placing each into a Vector
package c16.Trash;
import java.util.*;
import java.io.*;

public class ParseTrash {
  public static void
      fillBin(String filename, Fillable bin) {
    try {
      BufferedReader data =
        new BufferedReader(
          new FileReader(filename));
      String buf;
      while((buf = data.readLine())!= null) {
        String type =
          buf.substring(0, buf.indexOf(':')).trim();
        double weight = Double.valueOf(
          buf.substring(buf.indexOf(':') + 1)
          .trim()).doubleValue();
        bin.addTrash(
          Trash.factory(
            new Trash.Info(type, weight)));
      }
      data.close();
    } catch(IOException e) {
      e.printStackTrace();
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
  // Special case to handle Vector:
  public static void
      fillBin(String filename, Vector bin) {
    fillBin(filename, new FillableVector(bin));
```

```
  }
} ///:~
```

In **RecycleA.java**, a **Vector** was used to hold the **Trash** objects. However, other types of containers may be used as well. To allow for this, the first version of **fillBin( )** takes a handle to a **Fillable**, which is simply an **interface** that supports a method called **addTrash( )**:

```
//: Fillable.java
// Any object that can be filled with Trash
package c16.Trash;

public interface Fillable {
  void addTrash(Trash t);
} ///:~
```

Anything that supports this interface can be used with **fillBin**. Of course, **Vector** doesn't implement **Fillable**, so it won't work. Since **Vector** is used in most of the examples, it makes sense to add a second overloaded **fillBin( )** method that takes a **Vector**. To allow the **Vector** to be used as a **Fillable** object, it must be adapted. This is easily accomplished using a class:

```
//: FillableVector.java
// Adapter that makes a Vector Fillable
package c16.Trash;
import java.util.*;

public class FillableVector implements Fillable {
  private Vector v;
  public FillableVector(Vector vv) { v = vv; }
  public void addTrash(Trash t) {
    v.addElement(t);
  }
} ///:~
```

You can see that the only job of this class is to connect **Fillable**'s **addTrash( )** method to **Vector**'s **addElement( )**. With this class in hand, the overloaded **fillBin( )** method can be used with a **Vector** in **ParseTrash.java**:

```
  public static void
      fillBin(String filename, Vector bin) {
    fillBin(filename, new FillableVector(bin));
  }
```

This approach works for any container class that's used frequently. Alternatively, the container class can provide its own adapter that implements **Fillable** (you'll see this later, in **DynaTrash.java**).

recycling with prototyping

Now you can see the revised version of **RecycleA.java** using the prototyping technique:

```
//: RecycleAP.java
// Recycling with RTTI and Prototypes
package c16.RecycleAP;
import c16.Trash.*;
import java.util.*;

public class RecycleAP {
  public static void main(String args[]) {
    Vector bin = new Vector();
    // Fill up the Trash bin:
    ParseTrash.fillBin("Trash.dat", bin);
    Vector
      glassBin = new Vector(),
```

```
      paperBin = new Vector(),
      alBin = new Vector();
    Enumeration sorter = bin.elements();
    // Sort the Trash:
    while(sorter.hasMoreElements()) {
      Object t = sorter.nextElement();
      // RTTI to show class membership:
      if(t instanceof Aluminum)
        alBin.addElement(t);
      if(t instanceof Paper)
        paperBin.addElement(t);
      if(t instanceof Glass)
        glassBin.addElement(t);
    }
    Trash.sumValue(alBin);
    Trash.sumValue(paperBin);
    Trash.sumValue(glassBin);
    Trash.sumValue(bin);
  }
} ///:~
```

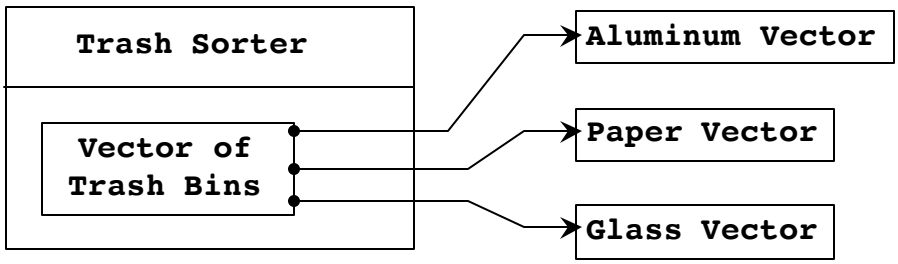
All the **Trash** objects, as well as the **ParseTrash** and support classes, are now part of the package **c16.Trash** so they are simply imported.

The process of opening the data file containing **Trash** descriptions and the parsing of that file has been wrapped in the **static** method **ParseTrash.fillBin( )** so now it's no longer a part of our design focus. You will see that throughout the rest of the chapter, no matter what new classes are added, **ParseTrash.fillBin( )** will continue to work without change, which indicates a good design.

In terms of object creation, this design does indeed severely localize the changes you need to make to add a new type to the system. However, there's a significant problem in the use of RTTI that shows up very clearly here. The program seems to run just fine, and yet it never detects any cardboard, even though there is cardboard in the list! This happens *because* of the use of RTTI, which only looks for the types that you tell it to look for. The clue that RTTI is being misused is that *every type in the system* is being tested, rather than just a single type or subset of types. As you will see later, there are ways to use polymorphism instead when you're testing for every type. But if you use RTTI a lot in this fashion, and you add a new type to your system, you can easily forget to make the necessary changes in your program and produce a difficult-to-find bug. So it's worth trying to eliminate RTTI in this case, not just for aesthetic reasons – it produces more maintainable code.

abstracting usage

With creation out of the way, it's time to tackle the remainder of the design: where the classes are used. Since it's the act of sorting into bins that's particularly ugly and exposed, why not take that process and hide it inside a class? This is the principle of "if you must do something ugly, at least localize the ugliness." It looks like this:



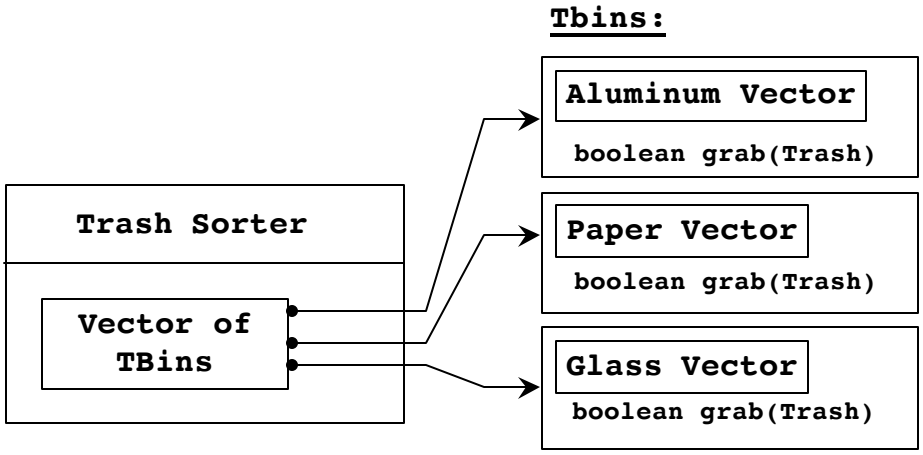
The **TrashSorter** object initialization must now be changed whenever a new type of **Trash** is added to the model. You could imagine that the **TrashSorter** class might look something like this:

```
class TrashSorter extends Vector {
  sort(Trash t) { /* ... */ }
}
```

That is, **TrashSorter** is a **Vector** of handles to **Vector**s of **Trash** handles, and with the **addElement( )** method you can install another one, like so:

```
TrashSorter ts = new TrashSorter();
ts.addElement(new Vector());
```

Now, however, the **sort( )** method becomes a problem. How does the statically-coded method deal with the fact that a new type has been added? To solve this, the type information must be removed from **sort( )** so all it needs to do is call a generic method which takes care of the details of type. This, of course, is another way to describe a dynamically-bound method. So **sort( )** will simply move through the sequence and call a dynamically-bound method for each **Vector**. Since the job of this method is to grab the pieces of trash it is interested in, it's called **grab(Trash)**. The structure now looks like:



TrashSorter needs to call each **grab( )** method and get a different result depending on what type of **Trash** the current **Vector** is holding. That is, each **Vector** must be aware of the type it holds. The classic approach to this problem is to create a base "**Trash** bin" class and inherit a new derived class for each different type you want to hold. If Java had a parameterized type mechanism that would probably be the most straightforward approach. But rather than hand-coding all the classes that such a mechanism should be building for us, further observation can produce a better approach.

A basic OOP design principle is "use data members for variation in state, use polymorphism for variation in behavior." Your first thought may be that the **grab( )** method certainly behaves differently for a **Vector** that holds **Paper** than one that holds **Glass**. But what it does is strictly dependent on the type, and nothing else. This could be interpreted as a different state, and since Java has a class to represent type (**Class**) this can be used to determine the type of **Trash** a particular **Tbin** will hold.

The constructor for this **Tbin** requires that you hand it the **Class** of your choice. This tells the **Vector** what type it is supposed to hold. Then the **grab( )** method uses **Class BinType** and RTTI to see if the **Trash** object you've handed it matches the type it's supposed to grab.

Here is the whole program. The commented numbers (e.g. (*1*)) mark sections that will be described following the code.

```
//: RecycleB.java
// Adding more objects to the recycling problem
package c16.RecycleB;
```

```
import c16.Trash.*;
import java.util.*;

// A vector that only admits the right type:
class Tbin extends Vector {
  Class binType;
  Tbin(Class binType) {
    this.binType = binType;
  }
  boolean grab(Trash t) {
    // Comparing class types:
    if(t.getClass().equals(binType)) {
      addElement(t);
      return true; // Object grabbed
    }
    return false; // Object not grabbed
  }
}

class TbinList extends Vector { //(*1*)
  boolean sort(Trash t) {
    Enumeration e = elements();
    while(e.hasMoreElements()) {
      Tbin bin = (Tbin)e.nextElement();
      if(bin.grab(t)) return true;
    }
    return false; // bin not found for t
  }
  void sortBin(Tbin bin) { // (*2*)
    Enumeration e = bin.elements();
    while(e.hasMoreElements())
      if(!sort((Trash)e.nextElement()))
        System.out.println("Bin not found");
  }
}

public class RecycleB {
  static Tbin bin = new Tbin(Trash.class);
  public static void main(String args[]) {
    // Fill up the Trash bin:
    ParseTrash.fillBin("Trash.dat", bin);

    TbinList trashBins = new TbinList();
    trashBins.addElement(
      new Tbin(Aluminum.class));
    trashBins.addElement(
      new Tbin(Paper.class));
    trashBins.addElement(
      new Tbin(Glass.class));
    // add one line here: (*3*)
    trashBins.addElement(
      new Tbin(Cardboard.class));

    trashBins.sortBin(bin); // (*4*)

    Enumeration e = trashBins.elements();
    while(e.hasMoreElements()) {
      Tbin b = (Tbin)e.nextElement();
      Trash.sumValue(b);
    }
```

```
    Trash.sumValue(bin);
  }
} ///:~
```

1. **TbinList** holds a set of **Tbin** handles, so that **sort( )** can iterate through the **Tbin**s when it's looking for a match for the **Trash** object you've handed it.

2. **sortBin( )** allows you to pass an entire **Tbin** in, and it moves through the **Tbin**, picks out each piece of **Trash** and sorts it into the appropriate specific **Tbin**. Note the genericity of this code: it doesn't change at all if new types are added. If the bulk of your code doesn't need changing when a new type is added (or some other change occurs) then you have an easily-extensible system.

3. Now you can see how easy it is to add a new type. Very few lines must be changed to support the addition. If it's really important, you can squeeze out even more by further manipulating the design.

4. One method call causes the contents of **bin** to be sorted into the respective specifically-typed bins.

multiple dispatching

The above design is certainly satisfactory. Adding new types to the system consists of adding or modifying distinct classes without causing code changes to be propagated throughout the system. In addition, RTTI is not "misused" as it was in **RecycleA.java**. However, it's possible to go one step further and take a purist viewpoint about RTTI and say that it should be eliminated altogether from the operation of sorting the trash into bins.

To accomplish this, you must first take the perspective that all type-dependent activities – such as detecting the type of a piece of trash and putting it into the appropriate bin – should be controlled through polymorphism and dynamic binding.

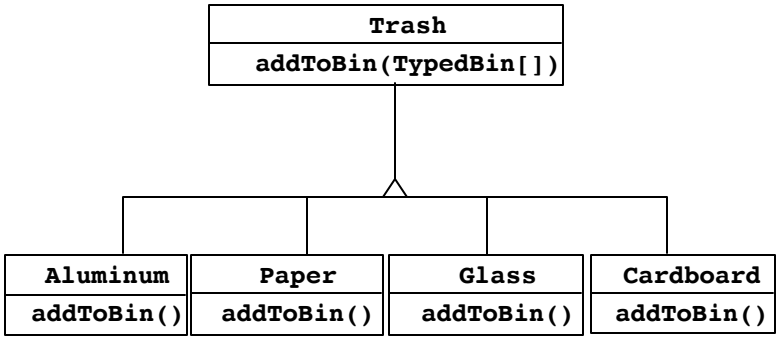
The previous examples first sorted by type, then acted on sequences of elements that were all of a particular type. But whenever you find yourself picking out particular types, stop and think. The whole idea of polymorphism (dynamically-bound method calls) is to handle type-specific information for you. So why are you hunting for types?

The answer is something you probably don't think about: Java only performs single dispatching. That is, if you are performing an operation on more than one object whose type is unknown, Java will only invoke the dynamic binding mechanism on one of those types. This doesn't solve the problem, so you end up detecting some types manually and effectively producing your own dynamic binding behavior.

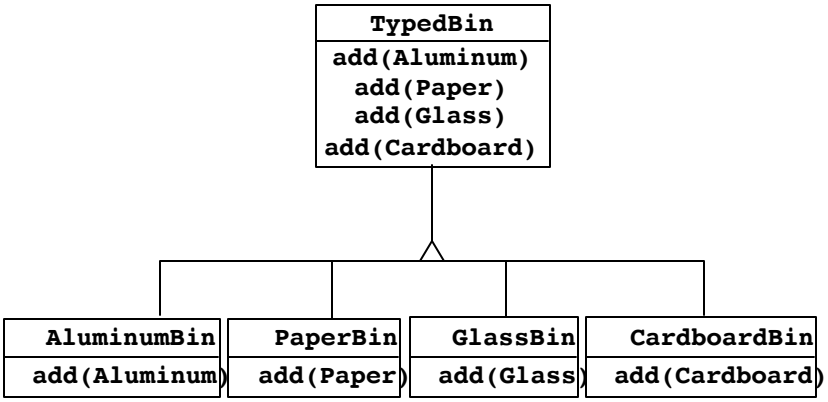
The solution is called *multiple dispatching*, which means setting up a configuration such that a single method call produces more than one dynamic method call, and thus determines more than one type in the process. To get this effect, then, you need to work with more than one type hierarchy: you'll need a type hierarchy for each dispatch. The following example works with two hierarchies: the existing **Trash** family and a hierarchy of the types of trash bins that the trash will be placed into. This second hierarchy isn't always obvious and in this case it needed to be created in order to produce multiple dispatching (in this case there will only be two dispatches, which is referred to as *double dispatching*).

implementing the double dispatch

Remember that polymorphism can only occur via method calls, so if you want double dispatching to occur there must be two method calls, one to determine the type in each hierarchy. In the **Trash** hierarchy there will be a new method called **addToBin( )**, which takes an argument of an array of **TypedBin**. It uses this array to step through and try to add itself to the appropriate bin, and this is where you'll see the double dispatch.



The new hierarchy is **TypedBin**, and it contains its own method called **add( )** which is also used polymorphically. But here's an additional twist: **add( )** is *overloaded* to take arguments of the different types of trash. So an essential part of the double dispatching scheme also involves overloading.



Redesigning the program encounters a dilemma: it's now necessary that the base class **Trash** contain a method **addToBin( )**. One approach to this is to copy all the code and change the base class. Another approach, which is one you can take when you don't have control of the source code, is to put the **addToBin( )** method in an interface, leave **Trash** alone, and inherit new specific types of **Aluminum**, **Paper**, **Glass** and **Cardboard**. This is the approach taken here.

Most of the classes in this design must be **public** so they are placed in their own files. Here's the interface

```
//: TypedBinMember.java
// An interface for adding the double dispatching
// method to the trash hierarchy without
// modifying the original hierarchy.
package c16.DoubleDispatch;

interface TypedBinMember {
  // The new method:
  boolean addToBin(TypedBin[] tb);
} ///:~
```

In each particular subtype of **Aluminum**, **Paper**, **Glass** and **Cardboard**, the **addToBin( )** method in the **interface TypedBinMember** is implemented,, but it *looks* like the code is exactly the same in each case:

```
//: DDAluminum.java
// Aluminum for double dispatching
package c16.DoubleDispatch;
import c16.Trash.*;
```

```
public class DDAluminum extends Aluminum
    implements TypedBinMember {
  public DDAluminum(double wt) { super(wt); }
  public boolean addToBin(TypedBin[] tb) {
    for(int i = 0; i < tb.length; i++)
      if(tb[i].add(this))
        return true;
    return false;
  }
} ///:~

//: DDPaper.java
// Paper for double dispatching
package c16.DoubleDispatch;
import c16.Trash.*;

public class DDPaper extends Paper
    implements TypedBinMember {
  public DDPaper(double wt) { super(wt); }
  public boolean addToBin(TypedBin[] tb) {
    for(int i = 0; i < tb.length; i++)
      if(tb[i].add(this))
        return true;
    return false;
  }
} ///:~

//: DDGlass.java
// Glass for double dispatching
package c16.DoubleDispatch;
import c16.Trash.*;

public class DDGlass extends Glass
    implements TypedBinMember {
  public DDGlass(double wt) { super(wt); }
  public boolean addToBin(TypedBin[] tb) {
    for(int i = 0; i < tb.length; i++)
      if(tb[i].add(this))
        return true;
    return false;
  }
} ///:~

//: DDCardboard.java
// Cardboard for double dispatching
package c16.DoubleDispatch;
import c16.Trash.*;

public class DDCardboard extends Cardboard
    implements TypedBinMember {
  public DDCardboard(double wt) { super(wt); }
  public boolean addToBin(TypedBin[] tb) {
    for(int i = 0; i < tb.length; i++)
      if(tb[i].add(this))
        return true;
    return false;
  }
} ///:~
```

The code in each **addToBin( )** calls **add( )** for each **TypedBin** object in the array. But notice the argument: **this**. The type of **this** is different for each subclass of **Trash**, so the code is in fact different

(although this code will benefit if a parameterized type mechanism is ever added to Java). So this is the first part of the double dispatch, because once you're inside this method you know you're **Aluminum**, or **Paper**, etc. During the call to **add( )**, this information is passed via the type of **this**. The compiler resolves the call to the proper overloaded version of **add( )**. But since **tb[i]** produces a handle to the base type **TypedBin**, this call will end up calling a different method depending on the actual type of **TypedBin** that's currently selected. This is the second dispatch.

Here's the base class for **TypedBin**:

```
//: TypedBin.java
// Vector that knows how to grab the right type
package c16.DoubleDispatch;
import c16.Trash.*;
import java.util.*;

public abstract class TypedBin {
  Vector v = new Vector();
  protected boolean addIt(Trash t) {
    v.addElement(t);
    return true;
  }
  public Enumeration elements() {
    return v.elements();
  }
  public boolean add(DDAluminum a) {
    return false;
  }
  public boolean add(DDPaper a) {
    return false;
  }
  public boolean add(DDGlass a) {
    return false;
  }
  public boolean add(DDCardboard a) {
    return false;
  }
} ///:~
```

You can see that the overloaded **add( )** methods all return **false**. This means that if the method is not overloaded in a derived class, it will continue to return **false** and the caller (the method **addToBin( )**, in this case) will assume that the current **Trash** object has not successfully been added to a container, and continue searching for the right container.

In each of the subclasses of **TypedBin**, only one overloaded method is overridden, according to the type of bin that's being created. For example, **CardboardBin** overrides **add(DDCardboard)**. The overridden method adds the trash object to its container and returns **true**, while all the rest of the **add( )** methods in **CardboardBin** continue to return **false** since they haven't been overridden. You'll see that this is another case where a parameterized type mechanism in Java would allow automatic generation of code (with C++ **template**s, you wouldn't have to explicitly write the subclasses or the **addToBin( )** method in **Trash**).

Since for this example the trash types have been customized and placed in a different directory, you'll need a different trash data file to make it work. Here's a possible **DDTrash.dat**:

```
c16.DoubleDispatch.DDGlass:54
c16.DoubleDispatch.DDPaper:22
c16.DoubleDispatch.DDPaper:11
c16.DoubleDispatch.DDGlass:17
c16.DoubleDispatch.DDAluminum:89
c16.DoubleDispatch.DDPaper:88
c16.DoubleDispatch.DDAluminum:76
```

```
c16.DoubleDispatch.DDCardboard:96
c16.DoubleDispatch.DDAluminum:25
c16.DoubleDispatch.DDAluminum:34
c16.DoubleDispatch.DDGlass:11
c16.DoubleDispatch.DDGlass:68
c16.DoubleDispatch.DDGlass:43
c16.DoubleDispatch.DDAluminum:27
c16.DoubleDispatch.DDCardboard:44
c16.DoubleDispatch.DDAluminum:18
c16.DoubleDispatch.DDPaper:91
c16.DoubleDispatch.DDGlass:63
c16.DoubleDispatch.DDGlass:50
c16.DoubleDispatch.DDGlass:80
c16.DoubleDispatch.DDAluminum:81
c16.DoubleDispatch.DDCardboard:12
c16.DoubleDispatch.DDGlass:12
c16.DoubleDispatch.DDGlass:54
c16.DoubleDispatch.DDAluminum:36
c16.DoubleDispatch.DDAluminum:93
c16.DoubleDispatch.DDGlass:93
c16.DoubleDispatch.DDPaper:80
c16.DoubleDispatch.DDGlass:36
c16.DoubleDispatch.DDGlass:12
c16.DoubleDispatch.DDGlass:60
c16.DoubleDispatch.DDPaper:66
c16.DoubleDispatch.DDAluminum:36
c16.DoubleDispatch.DDCardboard:22
```

Here's the rest of the program:

```
//: DoubleDispatch.java
// Using multiple dispatching to handle more
// than one unknown type during a method call.
package c16.DoubleDispatch;
import c16.Trash.*;
import java.util.*;

class AluminumBin extends TypedBin {
  public boolean add(DDAluminum a) {
    return addIt(a);
  }
}

class PaperBin extends TypedBin {
  public boolean add(DDPaper a) {
    return addIt(a);
  }
}

class GlassBin extends TypedBin {
  public boolean add(DDGlass a) {
    return addIt(a);
  }
}

class CardboardBin extends TypedBin {
  public boolean add(DDCardboard a) {
    return addIt(a);
  }
}
```

```
class TrashBinSet {
  private TypedBin[] binSet = {
    new AluminumBin(),
    new PaperBin(),
    new GlassBin(),
    new CardboardBin()
  };
  public void sortIntoBins(Vector bin) {
    Enumeration e = bin.elements();
    while(e.hasMoreElements()) {
      TypedBinMember t =
        (TypedBinMember)e.nextElement();
      if(!t.addToBin(binSet))
        System.err.println("Couldn't add " + t);
    }
  }
  public TypedBin[] binSet() { return binSet; }
}

public class DoubleDispatch {
  public static void main(String args[]) {
    Vector bin = new Vector();
    TrashBinSet bins = new TrashBinSet();
    // ParseTrash still works, without changes:
    ParseTrash.fillBin("DDTrash.dat", bin);
    // Sort from the master bin into the
    // individually-typed bins:
    bins.sortIntoBins(bin);
    TypedBin[] tb = bins.binSet();
    // Perform sumValue for each bin...
    for(int i = 0; i < tb.length; i++)
      Trash.sumValue(tb[i].v);
    // ... and for the master bin
    Trash.sumValue(bin);
  }
} ///:~
```

TrashBinSet encapsulates all the different types of **TypedBin**s, along with the **sortIntoBins( )** method which is where all the double dispatching takes place. You can see that once the structure is set up, sorting into the various **TypedBin**s is remarkably easy. In addition, the efficiency of two dynamic method calls is probably better than any other way you could sort.

Note the ease of use of this system inside **main( )**, as well as the complete independence of any specific type information within **main( )**. All other methods that only talk to the **Trash** base-class interface will be equally invulnerable to changes in **Trash** types.

The code changes necessary to add a new type are relatively isolated: you inherit the new type of **Trash** with its **addToBin( )** method, then you inherit a new **TypedBin** (this is really just a copy and simple edit) and finally you add a new type in the aggregate initialization for **TrashBinSet**.

the "visitor" pattern

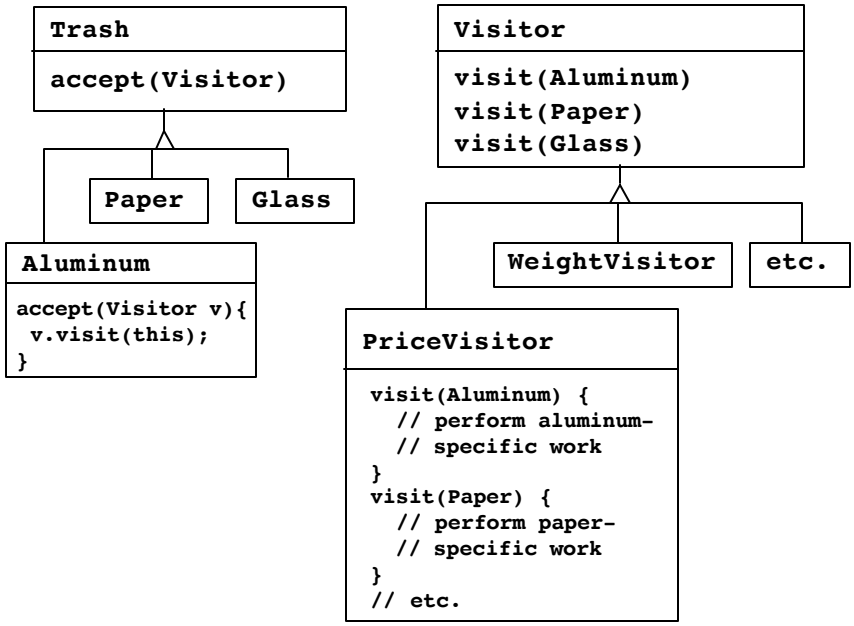
Now consider applying a design pattern with an entirely different goal to the trash-sorting problem.

For this pattern, we are no longer concerned with optimizing the addition of new types of **Trash** to the system. In fact, this pattern makes adding a new type of **Trash** *more* complicated. The assumption is actually that you have a primary class hierarchy that is fixed; perhaps it is from another vendor and you cannot make changes to that hierarchy. However, you'd like to add new polymorphic methods to that

hierarchy, which means that normally you'd have to add something to the base class interface. So the dilemma is that you need to add methods to the base class, but you can't touch the base class. How do you get around this?

The design pattern that solves this kind of problem is called a "visitor" (the last one in the *Design Patterns* book) and it builds on the double dispatching scheme shown in the last section.

The visitor pattern allows you to extend the interface of the primary type by creating a separate class hierarchy of type **Visitor** to virtualize the operations performed upon the primary type. The objects of the primary type simply "accept" the visitor, then call the visitor's dynamically-bound method. It looks like this:



Now, if **v** is a **Visitable** handle to an **Aluminum** object, the code:

```
PriceVisitor pv = new PriceVisitor();
v.accept(pv);
```

causes two polymorphic method calls: the first to select **Aluminum**'s version of **accept( )**, and the second within **accept( )** when the specific version of **visit( )** is called dynamically using the base-class **Visitor** reference **v**.

This configuration means that new functionality can be added to the system in the form of new subclasses of **Visitor**. The **Trash** hierarchy doesn't need to be touched. This is the prime benefit of the visitor pattern: you can add new polymorphic functionality to a class hierarchy without touching that hierarchy (once the **accept( )** methods have been installed). Notice the benefit is helpful here but not exactly what we started out to accomplish, so at first blush you might decide this isn't the desired solution.

But look at one thing that's been accomplished. The visitor solution avoids sorting from the master **Trash** sequence into individual typed sequences. Thus, you can leave everything in the single master sequence and simply pass through that sequence with the appropriate visitor to accomplish the goal. Although this behavior seems to be a side effect of visitor it does give us what we want (avoiding RTTI).

The double dispatching in the visitor pattern takes care of determining both the type of **Trash** and the type of **Visitor**. In the following example, there are two implementations of **Visitor**, **PriceVisitor** to determine and sum the price and **WeightVisitor** to keep track of the weights.

You can see all of this implemented in the new, improved version of the recycling program. As with **DoubleDispatch.java**, the **Trash** class is left alone and a new interface is created to add the **accept( )** method:

```
//: Visitable.java
// An interface to add visitor functionality to
// the Trash hierarchy without modifying the
// base class.
package c16.TrashVisitor;
import c16.Trash.*;

interface Visitable {
  // The new method:
  void accept(Visitor v);
} ///:~
```

The subtypes of **Aluminum**, **Paper**, **Glass** and **Cardboard** just implement the **accept( )** method:

```
//: VAluminum.java
// Aluminum for the visitor pattern
package c16.TrashVisitor;
import c16.Trash.*;

public class VAluminum extends Aluminum
    implements Visitable {
  public VAluminum(double wt) { super(wt); }
  public void accept(Visitor v) {
    v.visit(this);
  }
} ///:~

//: VPaper.java
// Paper for the visitor pattern
package c16.TrashVisitor;
import c16.Trash.*;

public class VPaper extends Paper
    implements Visitable {
  public VPaper(double wt) { super(wt); }
  public void accept(Visitor v) {
    v.visit(this);
  }
} ///:~

//: VGlass.java
// Glass for the visitor pattern
package c16.TrashVisitor;
import c16.Trash.*;

public class VGlass extends Glass
    implements Visitable {
  public VGlass(double wt) { super(wt); }
  public void accept(Visitor v) {
    v.visit(this);
  }
} ///:~

//: VCardboard.java
// Cardboard for the visitor pattern
package c16.TrashVisitor;
import c16.Trash.*;
```

```
public class VCardboard extends Cardboard
    implements Visitable {
  public VCardboard(double wt) { super(wt); }
  public void accept(Visitor v) {
    v.visit(this);
  }
} ///:~
```

Since there's nothing concrete in the **Visitor** base class, it can be created as an **interface**:

```
//: Visitor.java
// The base interface for visitors
package c16.TrashVisitor;
import c16.Trash.*;

interface Visitor {
  void visit(VAluminum a);
  void visit(VPaper p);
  void visit(VGlass g);
  void visit(VCardboard c);
} ///:~
```

Once again custom **Trash** types have been created in a different subdirectory. The new **Trash** data file is called **VTrash.dat** and it looks like this:

```
c16.TrashVisitor.VGlass:54
c16.TrashVisitor.VPaper:22
c16.TrashVisitor.VPaper:11
c16.TrashVisitor.VGlass:17
c16.TrashVisitor.VAluminum:89
c16.TrashVisitor.VPaper:88
c16.TrashVisitor.VAluminum:76
c16.TrashVisitor.VCardboard:96
c16.TrashVisitor.VAluminum:25
c16.TrashVisitor.VAluminum:34
c16.TrashVisitor.VGlass:11
c16.TrashVisitor.VGlass:68
c16.TrashVisitor.VGlass:43
c16.TrashVisitor.VAluminum:27
c16.TrashVisitor.VCardboard:44
c16.TrashVisitor.VAluminum:18
c16.TrashVisitor.VPaper:91
c16.TrashVisitor.VGlass:63
c16.TrashVisitor.VGlass:50
c16.TrashVisitor.VGlass:80
c16.TrashVisitor.VAluminum:81
c16.TrashVisitor.VCardboard:12
c16.TrashVisitor.VGlass:12
c16.TrashVisitor.VGlass:54
c16.TrashVisitor.VAluminum:36
c16.TrashVisitor.VAluminum:93
c16.TrashVisitor.VGlass:93
c16.TrashVisitor.VPaper:80
c16.TrashVisitor.VGlass:36
c16.TrashVisitor.VGlass:12
c16.TrashVisitor.VGlass:60
c16.TrashVisitor.VPaper:66
c16.TrashVisitor.VAluminum:36
c16.TrashVisitor.VCardboard:22
```

The rest of the program creates specific **Visitor** types and sends them through a single list of **Trash** objects:

```
//: TrashVisitor.java
// The "visitor" pattern
package c16.TrashVisitor;
import c16.Trash.*;
import java.util.*;

// Specific group of algorithms packaged
// in each implementation of Visitor:
class PriceVisitor implements Visitor {
  private double alSum; // Aluminum
  private double pSum; // Paper
  private double gSum; // Glass
  private double cSum; // Cardboard
  public void visit(VAluminum al) {
    double v = al.weight() * al.value();
    System.out.println(
      "value of Aluminum= " + v);
    alSum += v;
  }
  public void visit(VPaper p) {
    double v = p.weight() * p.value();
    System.out.println(
      "value of Paper= " + v);
    pSum += v;
  }
  public void visit(VGlass g) {
    double v = g.weight() * g.value();
    System.out.println(
      "value of Glass= " + v);
    gSum += v;
  }
  public void visit(VCardboard c) {
    double v = c.weight() * c.value();
    System.out.println(
      "value of Cardboard = " + v);
    cSum += v;
  }
  void total() {
    System.out.println(
      "Total Aluminum: $" + alSum + "\n" +
      "Total Paper: $" + pSum + "\n" +
      "Total Glass: $" + gSum + "\n" +
      "Total Cardboard: $" + cSum);
  }
}

class WeightVisitor implements Visitor {
  private double alSum; // Aluminum
  private double pSum; // Paper
  private double gSum; // Glass
  private double cSum; // Cardboard
  public void visit(VAluminum al) {
    alSum += al.weight();
    System.out.println("weight of Aluminum = "
        + al.weight());
  }
  public void visit(VPaper p) {
```

```
    pSum += p.weight();
    System.out.println("weight of Paper = "
        + p.weight());
  }
  public void visit(VGlass g) {
    gSum += g.weight();
    System.out.println("weight of Glass = "
        + g.weight());
  }
  public void visit(VCardboard c) {
    cSum += c.weight();
    System.out.println("weight of Cardboard = "
        + c.weight());
  }
  void total() {
    System.out.println("Total weight Aluminum:"
        + alSum);
    System.out.println("Total weight Paper:"
        + pSum);
    System.out.println("Total weight Glass:"
        + gSum);
    System.out.println("Total weight Cardboard:"
        + cSum);
  }
}

public class TrashVisitor {
  public static void main(String args[]) {
    Vector bin = new Vector();
    // ParseTrash still works, without changes:
    ParseTrash.fillBin("VTrash.dat", bin);
    // You could even iterate through
    // a list of visitors!
    PriceVisitor pv = new PriceVisitor();
    WeightVisitor wv = new WeightVisitor();
    Enumeration it = bin.elements();
    while(it.hasMoreElements()) {
      Visitable v = (Visitable)it.nextElement();
      v.accept(pv);
      v.accept(wv);
    }
    pv.total();
    wv.total();
  }
} ///:~
```

Notice that the shape of **main( )** has changed again. Now there's only a single **Trash** bin. The two **Visitor** objects are accepted into every element in the sequence, and they perform their operations. The visitors keep their own internal data to tally the total weights and prices.

Finally, there's no run-time type identification other than the inevitable cast to **Trash** when pulling things out of the sequence. This, too, could be eliminated with the implementation of parameterized types in Java.

One way you can distinguish this solution from the double dispatching solution described previously is to note that in the double dispatching solution only one of the overloaded methods **add( )** was overridden when each subclass was created, while here *each* one of the overloaded **visit( )** methods is overridden in every subclass of **Visitor**.

more coupling?

There's a lot more code here, and there's definite coupling between the **Trash** hierarchy and the **Visitor** hierarchy. However there's also very high cohesion within the respective sets of classes: they each only do one thing (**Trash** describes Trash, while **Visitor** describes actions performed on Trash), which is an indicator of a good design. Of course, in this case it only works well if you're adding new **Visitor**s, but it gets in the way when you add new types of **Trash**.

Low coupling between classes and high cohesion within a class is definitely an important design goal. Applied mindlessly, though, it can prevent you from achieving a more elegant design. In particular it seems that some classes inevitably have a certain intimacy with each other. These often occur in pairs that could perhaps be called *couplets*, for example containers and iterators (**Enumeration**s). The above **Trash-Visitor** pair appears to be another such couplet.

RTTI considered harmful?

Various designs in this chapter attempt to remove RTTI, and this could give you the impression that it's "considered harmful" (the condemnation used for poor, ill-fated **goto** which was thus never put into Java). This isn't true; the *misuse* of RTTI is the problem. The reason these designs removed RTTI is because the misapplication of that feature prevented extensibility, and the stated goal was to be able to add a new type to the system with as little impact on surrounding code as possible. Since RTTI is often misused to look for every single type in your system, it causes code to be non-extensible: when you add a new type, you have to go hunting for all the code where RTTI is used, and if you miss any you won't get help from the compiler.

However, using RTTI doesn't automatically create non-extensible code. Let's revisit the trash recycler once more. This time, a new tool will be introduced: a **Hashtable** that holds **Vector**s. The keys for this **Hashtable**, which I shall call a **TypeMap**, are the types of trash in the associated **Vector**. The beauty of this design (suggested by Larry O'Brien) is that the **TypeMap** dynamically adds a new pair whenever it encounters a new type, so whenever you add a new type to the system (even if you add the new type at run-time) it adapts.

The example will again build on the structure of the **Trash** types in **package c16.Trash** (and the **Trash.dat** file used there can be utilized here without change):

```
//: DynaTrash.java
// Using a Hashtable of Vectors and RTTI
// to automatically sort trash into
// vectors. This solution, despite the
// use of RTTI, is extensible.
package c16.DynaTrash;
import c16.Trash.*;
import java.util.*;

// Generic TypeMap works in any situation:
class TypeMap {
  private Hashtable t = new Hashtable();
  public void add(Object o) {
    Class type = o.getClass();
    if(t.containsKey(type))
      ((Vector)t.get(type)).addElement(o);
    else {
      Vector v = new Vector();
      v.addElement(o);
      t.put(type,v);
    }
  }
  public Vector get(Class type) {
    return (Vector)t.get(type);
  }
```

```
  public Enumeration keys() { return t.keys(); }
  // Returns handle to adapter class to allow
  // callbacks from ParseTrash.fillBin():
  public Fillable filler() {
    // Anonymous inner class:
    return new Fillable() {
      public void addTrash(Trash t) { add(t); }
    };
  }
}

public class DynaTrash {
  public static void main(String args[]) {
    TypeMap bin = new TypeMap();
    ParseTrash.fillBin("Trash.dat",bin.filler());
    Enumeration keys = bin.keys();
    while(keys.hasMoreElements())
      Trash.sumValue(
        bin.get((Class)keys.nextElement()));
  }
} ///:~
```

Although powerful, the definition for **TypeMap** is very simple. It contains a **Hashtable**, and the **add( )** method does most of the work. When you **add( )** a new object, the handle for the **Class** object for that type is extracted. This is used as a key to determine whether a **Vector** holding objects of that type is already present in the **Hashtable**. If so, that **Vector** is extracted and the object is added to the **Vector**. If not, the **Class** object and a new **Vector** are added as a key-value pair.

You can get an **Enumeration** of all the **Class** objects with **keys( )**, and use each **Class** object to fetch the corresponding **Vector** with **get( )**. And that's all there is to it.

The **filler( )** method is interesting because it takes advantage of the design of **ParseTrash.fillBin( )**, which doesn't just try to fill a **Vector** but instead anything that implements the **Fillable** interface with its **addTrash( )** method. All **filler( )** needs to do is return a handle to an **interface** that implements **Fillable**, and then this handle can be used as an argument to **fillBin( )** like this:

```
ParseTrash.fillBin("Trash.dat", bin.filler());
```

To produce this handle, an *anonymous inner class* (described in Chapter 7) is used. You never need a named class to implement **Fillable**, you just need a handle to an object of that class, thus this is an appropriate use of anonymous inner classes.

An interesting thing about this design is that even though it wasn't created to handle the sorting, **fillBin( )** is actually performing a sort every time it inserts a **Trash** object into **bin**.

Much of **class DynaTrash** should be familiar from the previous examples. This time, instead of putting the new **Trash** objects into a **bin** of type **Vector**, the **bin** is of type **TypeMap**, and so when the trash is thrown into **bin** it's immediately sorted by **TypeMap**'s internal sorting mechanism. Stepping through the **TypeMap** and operating on each individual **Vector** becomes a very simple matter:

```
    Enumeration keys = bin.keys();
    while(keys.hasMoreElements())
      Trash.sumValue(
        bin.get((Class)keys.nextElement()));
```

As you can see, adding a new type to the system won't affect this code, or the code in **TypeMap**, at all. This is certainly the smallest solution to the problem, and arguably the most elegant as well. It does rely heavily on RTTI, but notice that each key-value pair in the **Hashtable** is only looking for one type. In addition, there's no way you can "forget" to add the proper code to this system when you add a new type, since there isn't any code you need to add.

summary

Coming up with a design such as **TrashVisitor.java** that contains a larger amount of code than the earlier designs can seem at first to be counterproductive. It pays to notice what you're trying to accomplish with different designs. Design patterns in general strive to *separate the things that change from the things that stay the same*. The "things that change" can refer to many different kinds of changes. Perhaps the change occurs because the program is placed in a new environment, or because something in the current environment changes (this could be: "the user wants to add a new shape to the diagram on the screen"). Or, as in this case, the change could be the evolution of the code body itself. While previous versions of the trash-sorting example emphasized the addition of new *types* of **Trash** to the system, **TrashVisitor.java** allows you to easily add new *functionality* without disturbing the **Trash** hierarchy. There's more code in **TrashVisitor.java**, but adding new functionality to **Visitor** is very cheap. If this is something that happens a lot, then it's worth the extra effort and code to make it happen more easily.

Often, the most difficult part of developing an elegant and cheap-to-maintain design is in discovering what I call "the vector of change" (here, "vector" refers to the maximum gradient and not a container class). This means finding the most important thing that changes in your system, or put another way discovering where your greatest cost is. The discovery of the vector of change is no trivial matter; it's not something that an analyst can usually detect before the program sees its initial design. The necessary information will probably not appear until later phases in the project: sometimes only at the design or implementation phases do you discover a deeper or more subtle need in your system. In the case of adding new types (which was the focus of most of the "recycle" examples) you may only realize that you need a particular inheritance hierarchy when you are in the maintenance phase and you begin extending the system!

One of the most important things you will learn by studying design patterns seems to be an about-face from what has been promoted so far in this book. That is: "OOP is all about polymorphism." This statement can produce the "two-year-old with a hammer" syndrome (everything looks like a nail). That is, it's hard enough to "get" polymorphism, and once you do you try to cast all your designs into that one particular mold.

What design patterns say is that OOP isn't just about polymorphism. It's about "separating the things that change from the things that stay the same." Polymorphism is an especially important way to do this, and it turns out to be very helpful if the programming language directly supports polymorphism (so you don't have to wire it in yourself, which would tend to make it prohibitive). But design patterns in general show *other* ways to accomplish the basic goal, and once your eyes have been opened to this you will begin to search for more creative designs.

Since the *Design Patterns* book came out and made such an impact, people have been searching for other patterns. You can expect to see more of these appear as time goes on. Here are some sites recommended by Jim Coplien (http://www.bell-labs.com/~cope), who is one of the main proponents of the patterns movement:

http://st-www.cs.uiuc.edu/users/patterns
http://c2.com/cgi/wiki
http://c2.com/ppr
http://www.bell-labs.com/people/cope/Patterns/Process/index.html
http://www.bell-labs.com/cgi-user/OrgPatterns/OrgPatterns
http://st-www.cs.uiuc.edu/cgi-bin/wikic/wikic
http://www.cs.wustl.edu/~schmidt/patterns.html
http://www.espinc.com/patterns/overview.html

exercises

1. Using **SingletonPattern.java** as a starting point, create a class that manages a fixed number of its own objects.

2. Add a class **Plastic** to **TrashVisitor.java**.
3. Add a class **Plastic** to **DynaTrash.java**.

17

17: projects

This chapter includes a set of projects that build on the material presented in this book, or otherwise didn't fit in earlier chapters.

Most of these projects are significantly more complex than the examples in the rest of the book, and they often demonstrate new techniques and uses of class libraries.

text processing

If you come from a C or C++ background, you may be skeptical at first of Java's power when it comes to handling text. Indeed, one drawback is that execution speed is slower and that may hinder some of your efforts. However, the tools (in particular the **String** class) are quite powerful as the examples in this section show.

As you'll see, these examples were created to solve problems that arose in the creation of this book. However, they are not restricted to that and the solutions they offer can easily be adapted to other situations. In addition they show the power of Java in an area that has not previously been emphasized.

extracting code listings

You've no doubt noticed that each complete code listing (not code fragment) in this book begins and ends with special comment tag marks '//**:**' and '///**:~**'. This meta-information is included so that the code can be automatically extracted from the book into compilable source-code files. In my previous book, I had a system that allowed me to automatically incorporate tested code files into the book. In this book, however, I discovered that it was often easier to paste the code into the book once it was initially tested and (since you never get it right the first time) perform edits to the code within the book. But how to extract it and test the code? This program is the answer, and it may come in handy when you set out to solve a text processing problem. In addition, it demonstrates many of the **String** class features.

I first save the entire book in ASCII text format, in a separate file. The **CodePackager** program has two modes (which you can see described in **usageString**): if you use the **-p** flag, it expects to see an input file containing the ASCII text from the book. It will go through this file and use the comment tag marks to extract the code, and it uses the file name on the first line to determine the name of the file. In addition, it looks for the **package** statement in case it needs to put the file in a special directory (via the class path).

But that's not all. It also watches for the change in chapters by hunting for special markers at the beginning of the chapters (which are removed for the printed version of the book). It uses these to determine the subdirectory to place the file in – they are divided by chapters.

As each file is extracted, it is placed in a **SourceCodeFile** object which is then placed in a container (this process will be more thoroughly described later). These **SourceCodeFile** objects could simply be stored in files, but that brings us to the second use for this project. If you invoke **CodePackager** *without* the **-p** flag it expects a "packed" file as input, which it will then extract into separate files. So the **-p** flag means that the extracted files will be "packed" into this single file.

Why bother with the packed file? Because different computer platforms have different ways of storing text information in files. A big issue is the end-of-line character or characters, but other issues may also exist. However, since Java has a special type of IO stream – the **DataOutputStream** – which promises that, regardless of what machine the data is coming from, the storage of that data will be in a form that can be correctly retrieved by any other machine by using a **DataInputStream**. That is, Java itself handles all the platform-specific details, which is a large part of the promise of Java. So the **-p** flag stores everything into a single file in a universal format. You download this file and the Java program from the Web, and when you run **CodePackager** on this file *without* the **-p** flag the files will all be extracted to appropriate places on your system (you can specify an alternate subdirectory; otherwise the subdirectories will just be created off the current directory). You'll note there is some extra work to make sure that no system-specific formats remain by using **File** objects everywhere a path or a file is described. In addition, there's a sanity check: an empty file is placed in each subdirectory and the name of that file indicates how many files you should find in that subdirectory.

Here is the code, which will be described in detail at the end of the listing:

```
//: CodePackager.java
// "Packs" and "unpacks" the code in "Thinking
// in Java" for cross-platform distribution.
import java.util.*;
import java.io.*;

class Pr {
  static void error(String e) {
    System.err.println("ERROR: " + e);
    System.exit(1);
  }
}

class IO {
  static BufferedReader disOpen(File f) {
    BufferedReader in = null;
```

```
    try {
      in = new BufferedReader(
        new FileReader(f));
    } catch(IOException e) {
      Pr.error("could not open " + f);
    }
    return in;
  }
  static BufferedReader disOpen(String fname) {
    return disOpen(new File(fname));
  }
  static DataOutputStream dosOpen(File f) {
    DataOutputStream in = null;
    try {
      in = new DataOutputStream(
        new BufferedOutputStream(
          new FileOutputStream(f)));
    } catch(IOException e) {
      Pr.error("could not open " + f);
    }
    return in;
  }
  static DataOutputStream dosOpen(String fname) {
    return dosOpen(new File(fname));
  }
  static PrintWriter psOpen(File f) {
    PrintWriter in = null;
    try {
      in = new PrintWriter(
        new BufferedWriter(
          new FileWriter(f)));
    } catch(IOException e) {
      Pr.error("could not open " + f);
    }
    return in;
  }
  static PrintWriter psOpen(String fname) {
    return psOpen(new File(fname));
  }
  static void close(Writer os) {
    try {
      os.close();
    } catch(IOException e) {
      Pr.error("closing " + os);
    }
  }
  static void close(DataOutputStream os) {
    try {
      os.close();
    } catch(IOException e) {
      Pr.error("closing " + os);
    }
  }
  static void close(Reader os) {
    try {
      os.close();
    } catch(IOException e) {
      Pr.error("closing " + os);
    }
  }
```

```
}

class SourceCodeFile {
  public static final String
    startMarker = "//:", // Start of source file
    endMarker = "} ///:~", // End of source
    beginContinue = "} ///:Continued",
    endContinue = "///:Continuing",
    packMarker = "###", // Packed file header tag
    eol = // Line separator on current system
      System.getProperty("line.separator"),
    filesep = // System's file path separator
      System.getProperty("file.separator");
  public static String copyright = "";
  static {
    try {
      BufferedReader cr =
        new BufferedReader(
          new FileReader("Copyright.txt"));
      String crin;
      while((crin = cr.readLine()) != null)
        copyright += crin + "\n";
      cr.close();
    } catch(Exception e) {
      copyright = "";
    }
  }
  private String filename, dirname,
    contents = new String();
  // The file name separator from the old system:
  public static String oldsep;
  public String toString() {
    return dirname + filesep + filename;
  }
  // Constructor for parsing from document file:
  public SourceCodeFile(String dname,
      String firstLine, BufferedReader in) {
    dirname = dname;
    // Skip past marker:
    filename = firstLine.substring(
        startMarker.length()).trim();
    // Find space that terminates file name:
    if(filename.indexOf(' ') != -1)
      filename = filename.substring(
          0, filename.indexOf(' '));
    System.out.println("listing: " + filename);
    contents = firstLine + eol;
    contents += copyright + eol;
    String s;
    boolean foundEndMarker = false;
    try {
      while((s = in.readLine()) != null) {
        if(s.startsWith(startMarker))
          Pr.error("no end of file marker for " +
            filename);
        // For this program, no spaces before
        // the "package" keyword are allowed
        // in the input source code:
        else if(s.startsWith("package")) {
          // Extract package name:
```

```
          String pdir = s.substring(
            s.indexOf(' ')).trim();
          pdir = pdir.substring(
            0, pdir.indexOf(';')).trim();
          // Convert to path name:
          pdir = pdir.replace(
            '.', filesep.charAt(0));
          System.out.println("Package " + pdir);
          dirname = pdir;
        }
        contents += s + eol;
        // Move past continuations:
        if(s.startsWith(beginContinue))
          while((s = in.readLine()) != null)
            if(s.startsWith(endContinue)) {
              contents += s + eol;
              break;
            }
        // Watch for end of listing:
        if(s.startsWith(endMarker)) {
          foundEndMarker = true;
          break;
        }
      }
      if(!foundEndMarker)
        Pr.error(endMarker +
          " not found before end of file");
    } catch(IOException e) {
      Pr.error("Error reading line");
    }
  }
  // For recovering from a packed file:
  public SourceCodeFile(BufferedReader pFile) {
    try {
      String s = pFile.readLine();
      if(s == null) return;
      if(!s.startsWith(packMarker))
        Pr.error("Can't find " + packMarker
          + " in " + s);
      s = s.substring(
        packMarker.length()).trim();
      dirname = s.substring(0, s.indexOf('#'));
      filename = s.substring(s.indexOf('#') + 1);
      dirname = dirname.replace(
        oldsep.charAt(0), filesep.charAt(0));
      filename = filename.replace(
        oldsep.charAt(0), filesep.charAt(0));
      System.out.println("listing: " + dirname
        + filesep + filename);
      while((s = pFile.readLine()) != null) {
        if(s.startsWith(endMarker)) {
          contents += s;
          break;
        }
        contents += s + eol;
      }
    } catch(IOException e) {
      System.err.println("Error reading line");
    }
  }
```

```
  public boolean hasFile() {
    return filename != null;
  }
  public String directory() { return dirname; }
  public String filename() { return filename; }
  public String contents() { return contents; }
  // To write to a packed file:
  public void writePacked(DataOutputStream out) {
    try {
      out.writeBytes(
        packMarker + dirname + "#"
        + filename + eol);
      out.writeBytes(contents);
    } catch(IOException e) {
      Pr.error("writing " + dirname +
        filesep + filename);
    }
  }
  // To generate the actual file:
  public void writeFile(String rootpath) {
    File path = new File(rootpath, dirname);
    path.mkdirs();
    PrintWriter p =
      IO.psOpen(new File(path, filename));
    p.print(contents);
    IO.close(p);
  }
}

class DirMap {
  private Hashtable t = new Hashtable();
  private String rootpath;
  DirMap() {
    rootpath = System.getProperty("user.dir");
  }
  DirMap(String alternateDir) {
    rootpath = alternateDir;
  }
  public void add(SourceCodeFile f){
    String path = f.directory();
    if(!t.containsKey(path))
      t.put(path, new Vector());
    ((Vector)t.get(path)).addElement(f);
  }
  public void writePackedFile(String fname) {
    DataOutputStream packed = IO.dosOpen(fname);
    try {
      packed.writeBytes("###Old Separator:" +
        SourceCodeFile.filesep + "###\n");
    } catch(IOException e) {
      Pr.error("Writing separator to " + fname);
    }
    Enumeration e = t.keys();
    while(e.hasMoreElements()) {
      String dir = (String)e.nextElement();
      System.out.println(
        "Writing directory " + dir);
      Vector v = (Vector)t.get(dir);
      for(int i = 0; i < v.size(); i++) {
        SourceCodeFile f =
```

```
            (SourceCodeFile)v.elementAt(i);
          f.writePacked(packed);
        }
      }
      IO.close(packed);
    }
    // Write all the files in their directories:
    public void write() {
      Enumeration e = t.keys();
      while(e.hasMoreElements()) {
        String dir = (String)e.nextElement();
        Vector v = (Vector)t.get(dir);
        for(int i = 0; i < v.size(); i++) {
          SourceCodeFile f =
            (SourceCodeFile)v.elementAt(i);
          f.writeFile(rootpath);
        }
        // Add file indicating file quantity
        // written to this directory as a check:
        IO.close(IO.dosOpen(
          new File(new File(rootpath, dir),
            Integer.toString(v.size())+".files")));
      }
    }
  }

  public class CodePackager {
    private static final String usageString =
    "usage: java CodePackager packedFileName" +
    "\nExtracts source code files from packed \n" +
    "version of Tjava.doc sources into " +
    "directories off current directory\n" +
    "java CodePackager packedFileName newDir\n" +
    "Extracts into directories off newDir\n" +
    "java CodePackager -p source.txt packedFile" +
    "\nCreates packed version of source files" +
    "\nfrom text version of Tjava.doc";
    private static void usage() {
      System.err.println(usageString);
      System.exit(1);
    }
    public static void main(String args[]) {
      if(args.length == 0) usage();
      if(args[0].equals("-p")) {
        if(args.length != 3)
          usage();
        createPackedFile(args);
      }
      else {
        if(args.length > 2)
          usage();
        extractPackedFile(args);
      }
    }
    private static String
      currentLine, directory = "error";
    private static BufferedReader in;
    private static DirMap dm;
    private static void
        createPackedFile(String args[]) {
```

```
    dm = new DirMap();
    in = IO.disOpen(args[1]);
    try {
      while((currentLine = in.readLine())
          != null) {
        if(currentLine.startsWith("[[[")) {
          int i =
            currentLine.indexOf("directory:");
          if(i != -1) {
            directory = currentLine.substring(
                i + "directory:".length());
            directory = directory.substring(
                0, directory.indexOf(' '))
                .trim();
          }
        }
        else if(currentLine.startsWith(
            SourceCodeFile.startMarker)) {
          dm.add(new SourceCodeFile(
              directory, currentLine, in));
        }
        else if(currentLine.startsWith(
            SourceCodeFile.endMarker))
          Pr.error("file has no start marker");
        // Else ignore the input line
      }
    } catch(IOException e) {
      Pr.error("Error reading " + args[1]);
    }
    IO.close(in);
    dm.writePackedFile(args[2]);
  }
  private static void
      extractPackedFile(String args[]) {
    if(args.length == 2) // Alternate directory
      dm = new DirMap(args[1]);
    else // Current directory
      dm = new DirMap();
    in = IO.disOpen(args[0]);
    String s = null;
    try {
       s = in.readLine();
    } catch(IOException e) {
      Pr.error("Cannot read from " + in);
    }
    // Capture the separator used in the system
    // that packed the file:
    if(s.indexOf("###Old Separator:") != -1 ) {
      String oldsep = s.substring(
        "###Old Separator:".length());
      oldsep = oldsep.substring(
        0, oldsep. indexOf('#'));
      SourceCodeFile.oldsep = oldsep;
    }
    SourceCodeFile sf = new SourceCodeFile(in);
    while(sf.hasFile()) {
      dm.add(sf);
      sf = new SourceCodeFile(in);
    }
    dm.write();
```

```
  }
} ///:~
```

The first two classes are support/utility classes designed to make the rest of the program more consistent to write and more easily readable. The first, **Pr**, is very similar to the ANSI C library **perror**, since it prints an error message (but also exits the program). The second class encapsulates the creation of files, a process that was shown in Chapter 10 as one that rapidly becomes verbose and annoying. In Chapter 10, the proposed solution created new classes, but here **static** method calls are used. Within those methods the appropriate exceptions are caught and dealt with. These methods make the rest of the code much cleaner to read.

The first class that has anything to do with the actual problem being solved here is **SourceCodeFile**, which represents all the information (including the contents, file name and directory it belongs in) for one source code file in the book. It also contains a set of string constants representing the markers that start and end a file, a marker used inside the packed file, the current system's end-of-line separator and file path separator (notice the use of **System.getProperty( )** to get the local version), and a copyright notice, which is extracted from the following file **Copyright.txt**.

```
//////////////////////////////////////////////////
// Copyright (c) Bruce Eckel, 1997
// Source code file from the book "Thinking in Java"
// All rights reserved EXCEPT as allowed by the
// following statements: You may freely use this file
// for your own work (personal or commercial),
// including modifications and distribution in
// executable form only. You may not copy and
// distribute this file, but instead the sole
// distribution point is
// http://www.EckelObjects.com/Eckel where it is
// freely available. You may not remove this
// copyright and notice. You may not distribute
// modified versions of the source code in this
// package. You may not use this file in printed
// media without the express permission of the
// author. Bruce Eckel makes no representation about
// the suitability of this software for any purpose.
// It is provided "as is" without express or implied
// warranty of any kind, including any implied
// warranty of merchantability, fitness for a
// particular purpose or non-infringement. The entire
// risk as to the quality and performance of the
// software is with you. Bruce Eckel and the
// publisher shall not be liable for any damages
// suffered by you or any third party as a result of
// using or distributing software. In no event will
// Bruce Eckel or the publisher be liable for any
// lost revenue, profit or data, or for direct,
// indirect, special, consequential, incidental or
// punitive damages, however caused and regardless of
// the theory of liability, arising out of the use of
// or inability to use software, even if Bruce Eckel
// and the publisher have been advised of the
// possibility of such damages. Should the software
// prove defective, you assume the cost of all
// necessary servicing, repair, or correction. If you
// think you've found an error, please email all
// modified files with loudly commented changes to:
// Bruce@EckelObjects.com. (please use the same
// address for non-code errors found in the book).
//////////////////////////////////////////////////
```

When extracting files from a packed file, the file separator of the system that packed the file is also noted, so it can be replaced with the correct one for the local system.

building a packed file

The first constructor is used to extract a file from the ASCII text version of this book. The calling code (which appears further down in the listing) reads each line in until it finds one that matches the beginning of a listing. At that point, it creates a new **SourceCodeFile** object, passing it the directory name, the first line (which has already been read by the calling code) and the **BufferedReader** object from which to extract the rest of the source code listing.

At this point, you begin to see heavy use of the **String** methods. To extract the file name, the overloaded version of **substring( )** is called that takes the starting offset and just goes to the end of the **String**. This starting index is produced by finding the **length( )** of the **startMarker**. The **trim( )** method removes white space from both ends of the **String**. The first line may also have words after the name of the file; these are detected using the **indexOf( )** method which returns -1 if it cannot find the character you're looking for and the value where the first instance of that character is found if it does. Note there is also an overloaded version of **indexOf( )** that takes a **String** instead of a character.

Once the file name is parsed and stored, the first line is placed into the **contents String** (which is used to hold the entire text of the source code listing). At this point the rest of the lines are read and concatenated into the **contents String**. It's not quite that simple, since certain situations require special handling. First is error checking: if you run into a **startMarker**, it means that no end marker was placed at the end of the listing that's currently being collected. This is an error condition which aborts the program.

The second special case is the **package** keyword. Although Java is a free-form language, this program requires that the **package** keyword be at the beginning of the line, and that it not be placed inside of /* */ style comments. This constraint eliminates the need to do a full parse on the code.

When the **package** keyword is discovered, the package name is extracted by looking for the space at the beginning and the semicolon at the end (note this could also have been performed in a single operation by using the overloaded **substring( )** that takes both the starting and ending indexes). Then the dots in the package name are replaced by the file separator, although an assumption is made here that the file separator is only one character long. This is probably true on all systems, but it's a place to look if there are problems.

The default behavior is to concatenate each line to **contents**, along with the end-of-line string, until the **endMarker** is discovered which indicates that the constructor should terminate. If the end of the file is encountered before the **endMarker** is seen, that's an error.

extracting from a packed file

The second constructor is used to recover the source code files from a packed file. Here, the calling method doesn't have to worry about skipping over the intermediate text. The file just contains all the source-code files, placed end-to-end. Thus, all you need to hand to this constructor is the **BufferedReader** where the information is coming from, and the constructor takes it from there. There is some meta-information, however, at the beginning of each listing, and this is denoted by the **packMarker**. If the **packMarker** isn't there, it means the caller is mistakenly trying to use this constructor where it isn't appropriate.

Once the **packMarker** is found, it is stripped off and the directory name (terminated by a '**#**') and the file name (which goes to the end of the line) are extracted. In both cases, the old separator character is replaced by the one that is current to this machine, using the **String replace( )** method. The old separator is placed at the beginning of the packed file, and you'll see how that is extracted later in the listing.

The rest of the constructor is quite simple. It reads and concatenates each line to the **contents** until the **endMarker** is found.

accessing and writing the listings

The next set of methods are simple accessors: **directory( )**, **filename( )** (notice the method can have the same spelling and capitalization as the field) and **contents( )**, and **hasFile( )** to indicate whether this object contains a file or not (the need for this will be seen later).

The last three methods are concerned with writing this code listing to a file, either a packed file via **writePacked( )** or an actual Java source file via **writeFile( )**. All that **writePacked( )** needs is the **DataOutputStream** which was opened elsewhere and represents the file that's being written. It puts the header information on the first line and then calls **writeBytes( )** to write **contents** in a "universal" format.

When writing the actual Java source file, the file itself must be created. This is done using **IO.psOpen( )**, handing the method a **File** object that contains not just the file name but also the path. But the question is now: does this path exist? The user has the option of placing all the source code directories in a completely different subdirectory, which may not exist. So before each file is written, the **File.mkdirs( )** method is called with the path that you want to write the file into. This will make the entire path, all at once.

containing the entire collection of listings

It is very convenient to organize the listings as subdirectories as the whole collection is built in memory. One reason is another sanity check: as each subdirectory of listings is created, an additional file is added whose name contains the number of files in that directory.

The **DirMap** class produces this effect and demonstrates the concept of a "multimap." This is implemented with a **Hashtable** whose keys are the subdirectories being created and whose values are **Vector** objects containing the **SourceCodeFile** objects in that particular directory. Thus, instead of just mapping a key to a single value, the "multimap" maps a key to a set of values via the associated **Vector**. Although this sounds complex it's actually remarkably straightforward to implement. You'll see that most of the size of the **DirMap** class has to do with writing to files, and not to the "multimap" implementation.

There are two ways you can make a **DirMap**: the default constructor assumes you want to make the directories off of the current one, and the second constructor lets you specify an alternate starting directory path.

The **add( )** method is where quite a bit of dense action occurs. First, the **directory( )** is extracted from the **SourceCodeFile** you want to add, and then the **Hashtable** is examined to see if it contains that key already. If not, a new **Vector** is added to the **Hashtable** and associated with that key. At this point, the **Vector** is there, one way or another, and it is extracted so the **SourceCodeFile** can be added. Because **Vector**s can be easily combined with **Hashtable**s like this, the power of both is amplified.

Writing a packed file involves opening the file to write (as a **DataOutputStream** so the data is universally recoverable) and writing the header information about the old separator on the first line. Next, an **Enumeration** of the **Hashtable** keys is produced and stepped through to select each directory, and fetch the **Vector** associated with that directory so each **SourceCodeFile** in that **Vector** can be written to the packed file.

Writing the Java source files to their directories in **write( )** is almost identical to **writePackedFile( )** since both methods simply call the appropriate method in **SourceCodeFile**. Here, however, the root path is passed into **SourceCodeFile.writeFile( )**, and when all the files have been written the additional file with the name containing the number of files is also written.

the main program

The previously-described classes are used within **CodePackager**. First you see the usage string which is printed out whenever the end user invokes the program incorrectly, along with the **usage( )** method to call it and exit the program. All **main( )** does is determine whether you want to create a packed file or extract from one, then make sure the arguments are correct and call the appropriate method.

When a packed file is created, it's assumed to be made in the current directory so the **DirMap** is created using the default constructor. After the file is opened each line is read and examined for particular conditions:

1. If the line starts with a **[[[** this may be a new chapter and the line may indicate a change of directory if it contains the word "**directory:**" (these are removed from the printed version of the book). If so, the name of the new directory (where all the source files should be placed until the next directory marker is encountered) is extracted and assigned to **directory**.

2. If the line starts with the starting marker for a source code listing, a new **SourceCodeFile** object is created. The constructor reads the rest of the source listing in. The handle that results is directly added to the **DirMap**.

3. If the line starts with the end marker for a source code listing, something has gone wrong, since end markers should only be found by the **SourceCodeFile** constructor.

When extracting a packed file, the extraction may be to the current directory or to an alternate directory, so the **DirMap** object is created accordingly. The file is opened and the first line is read to extract the old file path separator information. Then the input is used to create the first **SourceCodeFile** object, which is added to the **DirMap**. New **SourceCodeFile** objects are created and added as long as they contain a file (the last one created will simply return when it runs out of input and then **hasFile( )** will return false).

checking capitalization style

Although the previous example may come in handy as a guide for some project of your own that involves text processing, this project is directly useful because it performs a style check to make sure that your capitalization conforms to the de-facto Java style. It opens each **.java** file in the current directory and extracts all the class names and identifiers, and then tells you if any of them don't meet the Java style.

To make this program operate correctly, you must first build a class name repository to hold all the class names in the standard Java library. You do this by moving into all the source code subdirectories for the standard Java library and running **ClassScanner** in each subdirectory, providing as arguments the name of the repository file (using the same path and name each time) and the **-a** flag to indicate that the class names should be added to the repository.

To use the program to check your code, you simply run it and hand it the path and name of the repository to use. It will check all the classes and identifiers in the current directory and tell you which ones don't follow the typical Java capitalization style.

You should be aware that the program isn't perfect; there a few times when it will point out what it thinks is a problem but on looking at the code you'll see that nothing needs to be changed. This is a little annoying, but it's still much easier than trying to find all these cases by staring at your code.

Here's the listing, which will be explained immediately following the code:

```
//: ClassScanner.java
// Scans all files in directory for classes
// and identifiers, to check capitalization.
// Assumes properly compiling code listings.
// Doesn't do everything right, but is a very
// useful aid.
import java.io.*;
import java.util.*;

class MultiStringMap extends Hashtable {
  public void add(String key, String value) {
    if(!containsKey(key))
      put(key, new Vector());
    ((Vector)get(key)).addElement(value);
  }
  public Vector getVector(String key) {
    if(!containsKey(key)) {
```

```
      System.err.println(
        "ERROR: cannot find key: " + key);
      System.exit(1);
    }
    return (Vector)get(key);
  }
  public void printValues(PrintStream p) {
    Enumeration k = keys();
    while(k.hasMoreElements()) {
      String oneKey = (String)k.nextElement();
      Vector val = getVector(oneKey);
      for(int i = 0; i < val.size(); i++)
        p.println((String)val.elementAt(i));
    }
  }
}

public class ClassScanner {
  private File path;
  private String[] fileList;
  private Properties classes = new Properties();
  private MultiStringMap
    classMap = new MultiStringMap(),
    identMap = new MultiStringMap();
  private StreamTokenizer in;
  public ClassScanner() {
    path = new File(".");
    fileList = path.list(new JavaFilter());
    for(int i = 0; i < fileList.length; i++) {
      System.out.println(fileList[i]);
      scanListing(fileList[i]);
    }
  }
  void scanListing(String fname) {
    try {
      in = new StreamTokenizer(
          new BufferedReader(
            new FileReader(fname)));
      // Doesn't seem to work:
      // in.slashStarComments(true);
      // in.slashSlashComments(true);
      in.ordinaryChar('/');
      in.ordinaryChar('.');
      in.wordChars('_', '_');
      in.eolIsSignificant(true);
      while(in.nextToken() !=
            StreamTokenizer.TT_EOF) {
        if(in.ttype == '/')
          eatComments();
        else if(in.ttype ==
                StreamTokenizer.TT_WORD) {
          if(in.sval.equals("class") ||
             in.sval.equals("interface")) {
            // Get class name:
              while(in.nextToken() !=
                    StreamTokenizer.TT_EOF
                    && in.ttype !=
                    StreamTokenizer.TT_WORD)
                ;
              classes.put(in.sval, in.sval);
```

```
              classMap.add(fname, in.sval);
          }
          if(in.sval.equals("import") ||
             in.sval.equals("package"))
            discardLine();
          else // It's an identifier or keyword
            identMap.add(fname, in.sval);
        }
      }
    } catch(IOException e) {
      e.printStackTrace();
    }
  }
  void discardLine() {
    try {
      while(in.nextToken() !=
            StreamTokenizer.TT_EOF
            && in.ttype !=
            StreamTokenizer.TT_EOL)
        ; // Throw away tokens to end of line
    } catch(IOException e) {
      e.printStackTrace();
    }
  }
  // StreamTokenizer's comment removal seemed
  // to be broken. This extracts them:
  void eatComments() {
    try {
      if(in.nextToken() !=
         StreamTokenizer.TT_EOF) {
        if(in.ttype == '/')
          discardLine();
        else if(in.ttype != '*')
          in.pushBack();
        else
          while(true) {
            if(in.nextToken() ==
              StreamTokenizer.TT_EOF)
              break;
            if(in.ttype == '*')
              if(in.nextToken() !=
                StreamTokenizer.TT_EOF
                && in.ttype == '/')
                break;
          }
      }
    } catch(IOException e) {
      e.printStackTrace();
    }
  }
  public String[] classNames() {
    String[] result = new String[classes.size()];
    Enumeration e = classes.keys();
    int i = 0;
    while(e.hasMoreElements())
      result[i++] = (String)e.nextElement();
    return result;
  }
  public void checkClassNames() {
    Enumeration files = classMap.keys();
```

```
    while(files.hasMoreElements()) {
      String file = (String)files.nextElement();
      Vector cls = classMap.getVector(file);
      for(int i = 0; i < cls.size(); i++) {
        String className =
          (String)cls.elementAt(i);
        if(Character.isLowerCase(
             className.charAt(0)))
          System.out.println(
            "class capitalization error, file: "
            + file + ", class: "
            + className);
      }
    }
  }
  public void checkIdentNames() {
    Enumeration files = identMap.keys();
    Vector reportSet = new Vector();
    while(files.hasMoreElements()) {
      String file = (String)files.nextElement();
      Vector ids = identMap.getVector(file);
      for(int i = 0; i < ids.size(); i++) {
        String id =
          (String)ids.elementAt(i);
        if(!classes.contains(id)) {
          // Ignore identifiers of length 3 or
          // longer that are all uppercase
          // (probably static final values):
          if(id.length() >= 3 &&
             id.equals(
               id.toUpperCase()))
            continue;
          // Check to see if first char is upper:
          if(Character.isUpperCase(id.charAt(0))){
            if(reportSet.indexOf(file + id)
                == -1){ // Not reported yet
              reportSet.addElement(file + id);
              System.out.println(
                "Ident capitalization error in:"
                + file + ", ident: " + id);
            }
          }
        }
      }
    }
  }
  static final String usage =
    "Usage: \n" +
    "ClassScanner classnames -a\n" +
    "\tAdds all the class names in this \n" +
    "\tdirectory to the repository file \n" +
    "\tcalled 'classnames'\n" +
    "ClassScanner classnames\n" +
    "\tChecks all the java files in this \n" +
    "\tdirectory for capitalization errors, \n" +
    "\tusing the repository file 'classnames'";
  private static void usage() {
    System.err.println(usage);
    System.exit(1);
  }
```

```
  public static void main(String[] args) {
    if(args.length < 1 || args.length > 2)
      usage();
    ClassScanner c = new ClassScanner();
    File old = new File(args[0]);
    if(old.exists()) {
      try {
        // Try to open an existing
        // properties file:
        InputStream oldlist =
          new BufferedInputStream(
            new FileInputStream(old));
        c.classes.load(oldlist);
        oldlist.close();
      } catch(IOException e) {
        System.err.println("Could not open "
          + old + " for reading");
        System.exit(1);
      }
    }
    if(args.length == 1) {
      c.checkClassNames();
      c.checkIdentNames();
    }
    // Write the class names to a repository:
    if(args.length == 2) {
      if(!args[1].equals("-a"))
        usage();
      try {
        BufferedOutputStream out =
          new BufferedOutputStream(
            new FileOutputStream(args[0]));
        c.classes.save(out,
          "Classes found by ClassScanner.java");
        out.close();
      } catch(IOException e) {
        System.err.println(
          "Could not write " + args[0]);
        System.exit(1);
      }
    }
  }
}

class JavaFilter implements FilenameFilter {
  public boolean accept(File dir, String name) {
    // Strip path information:
    String f = new File(name).getName();
    return f.trim().endsWith(".java");
  }
} ///:~
```

The class **MultiStringMap** is a tool that allows you to map a group of strings onto each key entry. As in the previous example, it uses a **Hashtable** (this time with inheritance) with the key as the single string that's mapped onto the **Vector** value. The **add( )** method simply checks to see if there's a key already in the **Hashtable**, and if not it puts one there. The **getVector( )** method produces a **Vector** for a particular key, and **printValues( )**, which is primarily useful for debugging, prints out all the values, **Vector** by **Vector**.

To keep life simple, the class names from the standard Java libraries are all put into a **Properties** object (from the standard Java library). Remember that a **Properties** object is a **Hashtable** that only holds **String** objects for both the key and value entries. However, it can be saved to disk and restored from disk in one method call, so it's ideal for the repository of names. However, we only really need a list of names, and a **Hashtable** cannot accept **null** for either it's key or value entry. So the same object will be used for both the key and the value.

For the classes and identifiers that are discovered for the files in a particular directory, two **MultiStringMap**s are used: **classMap** and **identMap**. In addition, when the program starts up it loads the standard class name repository into the **Properties** object called **classes**, and when a new class name is found in the local directory that is also added to **classes** as well as to **classMap**. This way, **classMap** can be used to step through all the classes in the local directory, and **classes** can be used to see if the current token is a class name (which indicates a definition of an object or method is beginning, so grab the next tokens – until a semicolon – and put them into **identMap**).

The default constructor for **ClassScanner** creates a list of file names (using the **JavaFilter** implementation of **FilenameFilter** as described in Chapter 10). Then it calls **scanListing( )** for each file name.

Inside **scanListing( )** the source code file is opened and turned into a **StreamTokenizer**. In the documentation, passing **true** to **slashStarComments( )** and **slashSlashComments( )** is supposed to strip those comments out but this seems to be a bit flawed (it doesn't quite work in Java 1), so those lines are commented out and instead, as you shall see, the comments are extracted by another method. To do this, the '/' must be captured as an ordinary character rather than letting the **StreamTokenizer** absorb it as part of a comment, and the **ordinaryChar** method tells the **StreamTokenizer** to do this. This is also true for dots ('**.**'), since we want to have the method calls pulled apart into individual identifiers. However, the underscore, which is ordinarily treated by **StreamTokenizer** as an individual character, should be left as part of identifiers since it appears in such **static final** values as **TT_EOF** etc., used in this very program. The **wordChars( )** method takes a range of characters you want to add to those that are left inside a token that is being parsed as a word. Finally, when parsing for one-line comments or discarding a line we need to know when an end-of-line occurs, so by calling **eolIsSignificant(true)** the eol will show up rather than being absorbed by the **StreamTokenizer**.

The rest of **scanListing( )** reads and reacts to tokens until the end of the file, signified when **nextToken( )** returns the **final static** value **StreamTokenizer.TT_EOF**.

If the token is a **'/'** this is potentially a comment, and **eatComments( )** is called to deal with it. The only other situation we're interested in here is if it's a word, of which there are some special cases.

If the word is **class** or **interface** then the next token represents a class or interface name, and this is put into **classes** and **classMap**. If the word is **import** or **package**, then we don't want the rest of the line. Anything else must be an identifier (which we're interested in) or a keyword (which we're not, but they're all lowercase anyway so it won't spoil things just to put those in). These are added to **identMap**.

The **discardLine( )** method is a simple tool which just looks for the end of a line. Notice that any time you get a new token, you must check for the end of the file.

The **eatComments( )** method is called whenever a forward slash is encountered in the main parsing loop. However, that doesn't necessarily mean a comment has been found, so the next token must be extracted to see if it's another forward slash (in which case the line is discarded) or an asterisk. But if it's neither of those, it means the token you've just pulled out is needed back in the main parsing loop! Fortunately, the **pushBack( )** method allows you to "push back" the current token onto the input stream so that when the main parsing loop calls **nextToken( )** it will get the one you just pushed back.

For convenience, the **classNames( )** method produces an array of all the names in the **classes** container. This method is not used in the program but is helpful for debugging.

The next two methods are where the actual checking takes place. In **checkClassNames( )**, the class names are extracted from the **classMap** (which, remember, contains only the names in this directory, organized by file name so the file name can be printed along with the errant class name). This is

accomplished by pulling each associated **Vector** and stepping through that, looking to see if the first character is lower case. If so, the appropriate error message is printed.

In **checkIdentNames( )**, a similar approach is taken: each identifier name is extracted from **identMap**. If the name is not in the **classes** list, it is assumed to be an identifier or keyword. A special case is checked: if the identifier length is 3 or more *and* all the characters are uppercase, this identifier is ignored because it's probably a **static final** value like **TT_EOF**. Of course, this is not a perfect algorithm but it assumes you'll eventually notice any all-uppercase identifiers that are out of place.

Rather than reporting every identifier that starts with an uppercase character, this function keeps track of which ones have already been reported in a **Vector** called **reportSet( )**. This treats the **Vector** as a "set" which tells you whether an item is already in the set. The item is produced by concatenating the file name and identifier. If the element isn't in the set, it is added and then the report is made.

The rest of the listing is comprised of **main( )**, which busies itself by handling the command line arguments and figuring out whether you're building a repository of class names from the standard Java library or checking the validity of code you've written. In both cases it makes a **ClassScanner** object.

Whether you're building a repository or using one, you must try to open the existing repository. By making a **File** object and testing for existence, you can decide whether to open the file and **load( )** the **Properties** list **classes** inside **ClassScanner** (the classes from the repository add to rather than overwrite the classes found by the **ClassScanner** constructor). If you only provide one command-line argument it means you just want to perform a check of the class names and identifier names, but if you provide two arguments (the second being "**-a**") you're building a class name repository. In this case an output file is opened and the method **Properties.save( )** is used to write the list to a file along with a string which provides header file information.

a method lookup tool

Chapter 11 introduced the Java 1.1 concept of *reflection* and used that feature to look up methods for a particular class – either the entire list of methods, or a subset that matches a keyword you provide. The magic of this is that it automatically shows you *all* the methods for a class without forcing you to walk up the inheritance hierarchy examining the base classes at each level. Thus it provides a valuable timesaving tool for programming: since most Java methods are verbose and descriptive, you can look for the methods that contain a word and when you find what you think you're looking for, check the online documentation.

However, you hadn't seen the AWT by Chapter 11 so that tool was developed as a command-line application. Here is the more useful GUI version, which dynamically updates the output as you type and also allows you to cut and paste from the output:

```
//: DisplayMethods.java
// Display the methods of any class inside
// a window. Dynamically narrows your search.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.lang.reflect.*;
import java.io.*;

public class DisplayMethods extends Applet {
  Class cl;
  Method m[];
  Constructor ctor[];
  String n[] = new String[0];
  TextField
    name = new TextField(40),
    searchFor = new TextField(30);
  Checkbox strip =
```

```
    new Checkbox("Strip Qualifiers");
  TextArea results = new TextArea(40, 65);
  public void init() {
    strip.setState(true);
    name.addTextListener(new NameL());
    searchFor.addTextListener(new SearchForL());
    strip.addItemListener(new StripL());
    Panel
      top = new Panel(),
      lower = new Panel(),
      p = new Panel();
    top.add(new Label("Qualified class name:"));
    top.add(name);
    lower.add(
      new Label("String to search for:"));
    lower.add(searchFor);
    lower.add(strip);
    p.setLayout(new BorderLayout());
    p.add(top, BorderLayout.NORTH);
    p.add(lower, BorderLayout.SOUTH);
    setLayout(new BorderLayout());
    add(p, BorderLayout.NORTH);
    add(results, BorderLayout.CENTER);
  }
  class NameL implements TextListener {
    public void textValueChanged(TextEvent e) {
      String nm = name.getText().trim();
      if(nm.length() == 0) {
        results.setText("No match");
        n = new String[0];
        return;
      }
      try {
        cl = Class.forName(nm);
      } catch (ClassNotFoundException ex) {
        results.setText("No match");
        return;
      }
      m = cl.getMethods();
      ctor = cl.getConstructors();
      // Convert to an array of Strings:
      n = new String[m.length + ctor.length];
      for(int i = 0; i < m.length; i++)
        n[i] = m[i].toString();
      for(int i = 0; i < ctor.length; i++)
        n[i + m.length] = ctor[i].toString();
      reDisplay();
    }
  }
  void reDisplay() {
    // Create the result set:
    String rs[] = new String[n.length];
    String find = searchFor.getText();
    int j = 0;
    // Select from the list if find exists:
    for (int i = 0; i < n.length; i++) {
      if(find == null)
        rs[j++] = n[i];
      else if(n[i].indexOf(find) != -1)
          rs[j++] = n[i];
```

```
    }
    results.setText("");
    if(strip.getState() == true)
      for (int i = 0; i < j; i++)
        results.append(
          StripQualifiers.strip(rs[i]) + "\n");
    else // Leave qualifiers on
      for (int i = 0; i < j; i++)
        results.append(rs[i] + "\n");
  }
  class StripL implements ItemListener {
    public void itemStateChanged(ItemEvent e) {
      reDisplay();
    }
  }
  class SearchForL implements TextListener {
    public void textValueChanged(TextEvent e) {
      reDisplay();
    }
  }
  static class WL extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
      System.exit(0);
    }
  }
  public static void main(String args[]) {
    DisplayMethods applet = new DisplayMethods();
    Frame aFrame = new Frame("Display Methods");
    aFrame.addWindowListener(new WL());
    aFrame.add(applet, BorderLayout.CENTER);
    aFrame.setSize(500,750);
    applet.init();
    applet.start();
    aFrame.setVisible(true);
  }
}

class StripQualifiers {
  private StreamTokenizer st;
  public StripQualifiers(String qualified) {
      st = new StreamTokenizer(
        new StringReader(qualified));
      st.ordinaryChar(' ');
  }
  public String getNext() {
    String s = null;
    try {
      if(st.nextToken() !=
            StreamTokenizer.TT_EOF) {
        switch(st.ttype) {
          case StreamTokenizer.TT_EOL:
            s = null;
            break;
          case StreamTokenizer.TT_NUMBER:
            s = Double.toString(st.nval);
            break;
          case StreamTokenizer.TT_WORD:
            s = new String(st.sval);
            break;
          default: // single character in ttype
```

```
            s = String.valueOf((char)st.ttype);
          }
        }
      } catch(IOException e) {
        System.out.println(e);
      }
      return s;
    }
    public static String strip(String qualified) {
      StripQualifiers sq =
        new StripQualifiers(qualified);
      String s = "", si;
      while((si = sq.getNext()) != null) {
        int lastDot = si.lastIndexOf('.');
        if(lastDot != -1)
          si = si.substring(lastDot + 1);
        s += si;
      }
      return s;
    }
  } ///:~
```

Some things you've seen before. As with many of the GUI programs in this book, this is created as both an application and an applet. Also, the **StripQualifiers** class is exactly the same as it was in Chapter 11.

The GUI contains a **TextField name** for you to type the fully-qualified class name you want to look up, and another one **searchFor** where you can enter the optional text to search for within the list of methods. The **Checkbox** allows you to say whether you want to use the fully-qualified names in the output or if you want the qualification stripped off. Finally, the results are displayed in a **TextArea**.

You'll notice there are no buttons or other components for you to indicate that you want the search to start. That's because both of the **TextField**s as well as the **Checkbox** are monitored by their listener objects. Whenever you make a change, the list is immediately updated. If you change the text within the **name** field, the new text is captured in **class NameL**. If the text isn't empty, it is used inside **Class.forName( )** to try to look up the class. As you're typing, of course, the name will be incomplete and **Class.forName( )** will fail, which means it throws an exception. This is trapped and the **TextArea** is set to "No match". But as soon as you type in a correct name (capitalization counts), **Class.forName( )** is successful and **getMethods( )** and **getConstructors( )** will return arrays of **Method** and **Constructor** objects, respectively. Each of the objects in these arrays is turned into a **String** with **toString( )** (this produces the complete method or constructor signature) and both lists are combined into a single array of **String** called **n**. The array **n** is a member of **class DisplayMethods** and is used for updating the display whenever **reDisplay( )** is called.

If you change the **Checkbox** or **searchFor** components, their listeners simply call **reDisplay( )**. Inside **reDisplay( )**, a temporary array of **String** is created called **rs** (for "result set"). The result set is either copied directly from **n** if there is no **find** word, or conditionally copied from the **String**s in **n** that contain the **find** word. Finally, the **strip Checkbox** is interrogated to see if the user wants the names to be stripped (the default is "yes"). If so, **StripQualifiers.strip( )** does the job, if not, the list is simply displayed.

In **init( )**, you may think that there's a lot of busy work involved in setting up the layout. In fact, it is possible to lay out the components with less work, but the advantage of using **BorderLayout**s this way is that it allows the user to resize the window and make – in particular – the **TextArea** larger, which means you can resize to allow you to see longer names without scrolling.

You may find that you'll keep this tool running all the time when you're programming, since it provides one of the best "first lines of attack" when you're trying to figure out what method to call.

complexity theory

This program was modified from code originally created by Larry O'Brien, and is based on the "Boids" program created by Craig Reynolds in 1986 to demonstrate an aspect of complexity theory called "emergence."

The goal here is to produce a reasonably lifelike reproduction of flocking or herding behavior in animals by establishing a small set of very simple rules for each animal. Each animal can look at the entire scene and all the other animals in the scene, but it reacts only to a set of nearby "flockmates." The animal moves according to three simple steering behaviors:

1. Separation: avoid crowding local flockmates.
2. Alignment: follow the average heading of local flockmates.
3. Cohesion: move toward the center of the group of local flockmates.

More elaborate models can include obstacles and the ability for the animals to predict collisions and avoid them, so the animals can flow around fixed objects in the environment. In addition, the animals may also be given a goal, which can cause the herd to follow a desired path. For simplicity, obstacle avoidance and goal-seeking is not included in the model presented here.

Emergence means that, despite the limited nature of computers and the simplicity of the steering rules, the result seems quite realistic. That is, remarkably lifelike behavior "emerges" from this simple model.

The program is presented as a combined application/applet:

```
//: FieldOBeasts.java
// Demonstration of complexity theory; simulates
// herding behavior in animals. Adapted from
// a program by Larry O'Brien lobrien@msn.com
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.util.*;

class Beast {
  int
    x, y,            // Screen position
    currentSpeed;    // Pixels per second
  float currentDirection;  // Radians
  Color color;      // Fill color
  FieldOBeasts field; // Where the beasts roam
  static final int gsize = 10; // Graphic size

  public Beast(FieldOBeasts f, int x, int y,
      float cD, int cS, Color c) {
    field = f;
    this.x = x;
    this.y = y;
    currentDirection = cD;
    currentSpeed = cS;
    color = c;
  }
  public void step() {
    // You move based on those within your sight:
    Vector seen = field.beastListInSector(this);
    // If you're not out in front
    if(seen.size() > 0) {
```

```
      // Gather data on those you see
      int totalSpeed = 0;
      float totalBearing = 0.0f;
      float distanceToNearest = 100000.0f;
      Beast nearestBeast =
        (Beast)seen.elementAt(0);
      Enumeration e = seen.elements();
      while(e.hasMoreElements()) {
        Beast aBeast = (Beast) e.nextElement();
        totalSpeed += aBeast.currentSpeed;
        float bearing =
          aBeast.bearingFromPointAlongAxis(
            x, y, currentDirection);
        totalBearing += bearing;
        float distanceToBeast =
          aBeast.distanceFromPoint(x, y);
        if(distanceToBeast < distanceToNearest) {
          nearestBeast = aBeast;
          distanceToNearest = distanceToBeast;
        }
      }
      // Rule 1: Match average speed of those
      // in the list:
      currentSpeed = totalSpeed / seen.size();
      // Rule 2: Move towards the perceived
      // center of gravity of the herd:
      currentDirection =
        totalBearing / seen.size();
      // Rule 3: Maintain a minimum distance
      // from those around you:
      if(distanceToNearest <=
         field.minimumDistance){
        currentDirection =
          nearestBeast.currentDirection;
        currentSpeed = nearestBeast.currentSpeed;
        if(currentSpeed > field.maxSpeed) {
          currentSpeed = field.maxSpeed;
        }
      }
    }
    else {  // You are in front, so slow down
      currentSpeed =
        (int)(currentSpeed * field.decayRate);
    }
    // So move already!
    x += (int)(Math.cos(currentDirection)
               * currentSpeed);
    y += (int)(Math.sin(currentDirection)
               * currentSpeed);
    x %= field.xExtent;
    y %= field.yExtent;
    if(x < 0)
      x += field.xExtent;
    if(y < 0)
      y += field.yExtent;
  }
  public float bearingFromPointAlongAxis (
      int originX, int originY, float axis) {
    // Returns angle of _this_ in the coordinate
    // system specified bearing in world
```

```
    // coordinate system
    try {
      double bearingInRadians =
        Math.atan(
          (this.y - originY) /
          (this.x - originX));
      // Inverse tan has two solutions, so you
      // have to correct for other quarters:
      if(x < originX) {
        if(y < originY) {
          bearingInRadians += - (float)Math.PI;
        }
        else {
          bearingInRadians =
            (float)Math.PI - bearingInRadians;
        }
      }
      // Just subtract the axis (in radians):
      return (float) (axis - bearingInRadians);
    } catch(ArithmeticException aE) {
      // Divide by 0 error possible on this
      if(x > originX) {
          return 0;
      }
      else
        return (float) Math.PI;
    }
  }
  public float distanceFromPoint(int x1, int y1){
    return (float) Math.sqrt(
      Math.pow(x1 - x, 2) +
      Math.pow(y1 - y, 2));
  }
  public Point position() {
    return new Point(x, y);
  }
  // Beasts know how to draw themselves:
  public void draw(Graphics g) {
    g.setColor(color);
    int directionInDegrees = (int)(
      (currentDirection * 360) / (2 * Math.PI));
    int startAngle = directionInDegrees -
      FieldOBeasts.halfFieldOfView;
    int endAngle = 90;
    g.fillArc(x, y, gsize, gsize,
      startAngle, endAngle);
  }
}

public class FieldOBeasts extends Applet
    implements Runnable {
  private Vector beasts;
  static float
    fieldOfView =
      (float) (Math.PI / 4), // In radians
    // Deceleration % per second:
    decayRate = 1.0f,
    minimumDistance = 10f; // In pixels
  static int
    halfFieldOfView = (int)(
```

```
      (fieldOfView * 360) / (2 * Math.PI)),
    xExtent,
    yExtent,
    numBeasts = 50,
    maxSpeed = 20; // Pixels/second
  boolean UniqueColors = true;
  Thread thisThread;
  int delay = 25;
  public FieldOBeasts(int xExtent, int yExtent) {
    this.xExtent = xExtent;
    this.yExtent = yExtent;
    beasts =
      makeBeastVector(numBeasts, UniqueColors);
    // Now start the beasts a-rovin':
    thisThread = new Thread(this);
    thisThread.start();
  }
  public void run() {
    while(true) {
      for(int i = 0; i < beasts.size(); i++){
        Beast b = (Beast) beasts.elementAt(i);
        b.step();
      }
      try {
        thisThread.sleep(delay);
      } catch(InterruptedException ex){}
      repaint(); // Otherwise it won't update
    }
  }
  Vector makeBeastVector(
      int quantity, boolean uniqueColors) {
    Vector newBeasts = new Vector();
    Random generator = new Random();
    // Only used if UniqueColors is on
    double cubeRootOfBeastNumber =
      Math.pow((double) numBeasts, 1.0 / 3.0);
    float colorCubeStepSize =
      (float) (1.0 / cubeRootOfBeastNumber);
    float r = 0.0f;
    float g = 0.0f;
    float b = 0.0f;
    for(int i = 0; i < quantity; i++) {
      int x =
        (int) (generator.nextFloat() * xExtent);
      if(x > xExtent - Beast.gsize)
        x -= Beast.gsize;
      int y =
        (int) (generator.nextFloat() * yExtent);
      if(y > yExtent - Beast.gsize)
        y -= Beast.gsize;
      float direction = (float)(
        generator.nextFloat() * 2 * Math.PI);
      int speed = (int)(
        generator.nextFloat() * (float)maxSpeed);
      if(uniqueColors) {
        r += colorCubeStepSize;
        if(r > 1.0) {
          r -= 1.0f;
          g += colorCubeStepSize;
          if( g > 1.0) {
```

```
            g -= 1.0f;
            b += colorCubeStepSize;
            if(b > 1.0)
              b -= 1.0f;
          }
        }
      }
      newBeasts.addElement(
        new Beast(this, x, y, direction, speed,
          new Color(r,g,b)));
    }
    return newBeasts;
  }
  public Vector beastListInSector(Beast viewer) {
    Vector output = new Vector();
    Enumeration e = beasts.elements();
    Beast aBeast = (Beast)beasts.elementAt(0);
    int counter = 0;
    while(e.hasMoreElements()) {
      aBeast = (Beast) e.nextElement();
      if(aBeast != viewer) {
        Point p = aBeast.position();
        Point v = viewer.position();
        float bearing =
          aBeast.bearingFromPointAlongAxis(
            v.x, v.y, viewer.currentDirection);
        if(Math.abs(bearing) < fieldOfView / 2)
         output.addElement(aBeast);
      }
    }
    return output;
  }
  public void paint(Graphics g)  {
    Enumeration e = beasts.elements();
    while(e.hasMoreElements()) {
      ((Beast)e.nextElement()).draw(g);
    }
  }
  static class WL extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
      System.exit(0);
    }
  }
  public static void main(String args[])   {
    FieldOBeasts field =
      new FieldOBeasts(640,480);
    Frame frame = new Frame("Field 'O Beasts");
    // Optionally use a command-line argument
    // for the sleep time:
    if(args.length >= 1)
      field.delay = Integer.parseInt(args[0]);
    frame.addWindowListener(new WL());
    frame.add(field, BorderLayout.CENTER);
    frame.setSize(640,480);
    field.init();
    field.start();
    frame.setVisible(true);
  }
} ///:~
```

Although this isn't a perfect reproduction of the behavior in Craig Reynold's "Boids" example, it exhibits its own fascinating characteristics, which you can modify by adjusting the numbers. You can find out more about the modeling of flocking behavior and see a spectacular 3-D version of Boids at Craig Reynold's page http://www.hmt.com/cwr/boids.html.

summary

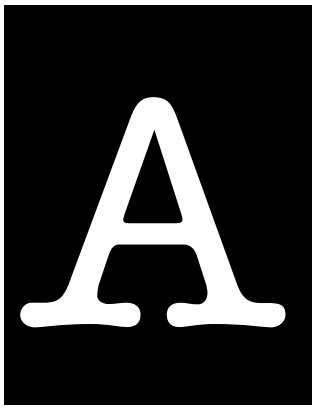
This chapter shows some of the more sophisticated things you can do with Java. It also makes the point that while Java must certainly have its limits, those limits are primarily relegated to performance (when the text-processing programs were written, for example, C++ versions were much faster – this may be mainly due to an inefficient implementation of the IO library, which should change in time). The limits of Java *do not* seem to be in the area of expressiveness. Not only does it seem possible to express just about everything you can imagine, Java seems oriented towards making that expression easy to write and read, so you don't run into the wall of complexity that often occurs with languages that are more trivial to use than Java (at least they seem that way, at first). Of course, the AWT remains the one area where expressing yourself can be a bit frustrating. Well, at least it's cross-platform.

exercises

1. (Term project) Taking **FieldOBeasts.java** as a starting point, build an automobile traffic simulation system.

2. (Term project) Using **ClassScanner.java** as a starting point, build a tool that points out methods and fields that are defined but never used.

3. (Term project) Using JDBC, build a contact management program using a flat-file database containing names, addresses, phone numbers, email addresses, etc. You should be able to easily add new names to the database. When typing in the name to be looked up, use automatic name completion as shown in **VLookup.java** in Chapter 15.



A: using non-Java code

This appendix contains pointers to other resources for connecting Java to non-Java code.

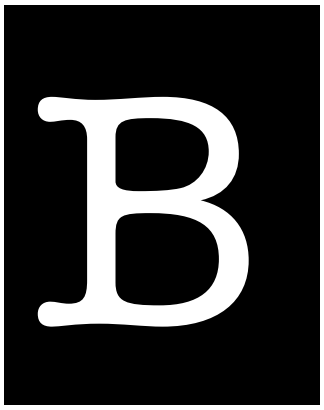
In Chapter 15, the section titled “a Web application” contains an example of connecting to non-Java code using standard input and output.

native methods

For information about programming native methods in Java 1.1, see the electronic version of *The Java Tutorial* by Campione and Walrath at http://www.Javasoft.com.

CORBA

CORBA is designed to support computing with distributed objects, but to accomplish this it also allows you to program in any language that has CORBA support. Thus, you can write code in one language and connect it to Java via CORBA. Full details can be found in *Client/Server Programming with Java and CORBA* by Orfali & Harkey (John Wiley & Sons, 1997). The official source for technical references is the Object Management Group (OMG) Web site: http://www.omg.org.



B: comparing C++ and Java

As a C++ programmer you already have the basic idea of object-oriented programming, and the syntax of Java no doubt looks very familiar to you. This makes sense since Java was derived from C++.

However, there are a surprising number of differences between C++ and Java. These differences are intended to be significant improvements, and if you understand the differences you'll see why Java is such a beneficial programming language. This appendix takes you through the important features that make Java distinct from C++.

1. The biggest potential stumbling block is speed: interpreted Java runs something like 20 times slower than C. Nothing prevents the Java language from being compiled and there are just-in-time compilers appearing at this writing which offer significant speed-ups. It is not inconcievable that full native compilers will appear for the more popular platforms, but without those there are classes of problems that will be insoluble with Java because of the speed issue.

2. Java has both kinds of comments like C++ does.

3. Everything must be in a class. There are no global functions or global data. If you want the equivalent of globals, make **static** methods and **static** data within a class. There are no structs or enumerations or unions. Only classes.

4. All method definitions are defined in the body of the class. Thus, in C++ it would look like all the functions are inlined, but they're not (inlines are noted later).

5. Class definitions are roughly the same form in Java as in C++, but there's no closing semicolon. There are no class declarations of the form **class foo;** only class definitions.
```
class aType {
  void aFunction( ) { /* method body */ }
}
```

6. There's no scope resolution operator **::** in Java. Java uses the dot for everything, but can get away with it since you can only define elements within a class. Even the method definitions must always occur within a class so there is no need for scope resolution there, either. One place where you'll notice the difference is in the calling of **static** methods: you say **ClassName.methodName( );**. In addition, **package** names are established using the dot, and to perform the equivalent of a C++ **#include** you use the **import** keyword. For example: **import java.awt.*;**.

7. Java, like C++, has primitive types for efficient access. In Java, these are **boolean**, **char**, **byte** , **short**, **int**, **long**, **float**, and **double**. All the primitive types have specified sizes that are machine-independent for portability (this must have some impact on performance, varying with the machine). Type-checking and type requirements are much tighter in Java. For example:
A. Conditional expressions can only be **boolean**, not integral
B. The result of an expression like X + Y must be used; you can't just say "X + Y" for the side effect.

8. The **char** type uses the international 16-bit Unicode character set, so it can automatically represent most national characters.

9. Static quoted strings are automatically converted into **String** objects. There is no independent static character array string as there is in C/C++.

10. Java adds the triple right shift **>>>** to act as a "logical" right shift by inserting zeroes at the top end; the **>>** inserts the sign bit as it shifts (an "arithmetic" shift).

11. Arrays are quite different in Java. There's a read-only **length** member that tells you how big the array is, and run-time checking throws an exception if you go out of bounds. All arrays are created on the heap, and you can assign one array to another (the array handle is simply copied). The array identifier itself is a first-class object, with all the methods commonly available to all other objects.

12. All non-primitive types can only be created using **new**. There's no equivalent to creating class objects "on the stack" as in C++. All primitive types can only be created directly, without **new**. There are wrapper classes for all primitive classes so you can create equivalent heap-based objects with **new**. (Arrays of primitives are a special case: they can be allocated via aggregate initialization as in C++, or by using **new**).

13. No forward references are necessary in Java. If you want to use a class or a method before it is defined in, you simply use it – the compiler ensures that the appropriate definition exists. Thus you don't have any of the forward referencing issues that you do in C++.

14. Java has no preprocessor. If you want to use classes in another library, you say **import** and the name of the library. There are no preprocessor-like macros.

15. Java uses packages in place of namespaces. The name issue is taken care of by (1) putting everything in a class and (2) a facility called "packages" that performs the equivalent namespace breakup for class names. Packages also collect library

components under a single library name. You simply **import** a package and the compiler takes care of the rest.

16. Object handles defined as class members are automatically initialized to **null**. Initialization of primitive class data members is guaranteed in Java; if you don't explicitly initialize them they get a default value (a zero or equivalent). You can initialize them directly when you define them in the class, or you can do it in the constructor. The syntax makes more sense than C++, and is consistent for **static** and non-**static** members alike. You don't need to externally define storage for **static** members like you do in C++.

17. There are no Java pointers in the sense of C and C++. When you create an object with **new**, you get back a reference (which I've been calling a *handle* in this book). For example:
```
String s = new String("howdy");
```
However, unlike C++ references that must be initialized when created and cannot be rebound to a different location, Java references don't have to be bound at the point of creation, and they can be rebound at will, which eliminates part of the need for pointers. The other reason for pointers is to select any place in memory (which makes them unsafe, which is why Java doesn't support them). Pointers are often seen as an efficient way to move through an array of primitive variables; Java arrays allow you to do that in a safer fashion. The final solution for pointer problems is native methods (discussed in Appendix A). Passing pointers to methods isn't a problem since there are no global functions, only classes, and you can pass references to class objects.
The Java language promoters initially said "no pointers!" but when many programmers questioned "how can you work without pointers?" they began saying "restricted pointers." You can make up your mind whether it's "really" a pointer or not. In any event, there's no pointer *arithmetic*.

18. Java has constructors, similar to constructors in C++. You get a default constructor if you don't define one, and if you define a non-default constructor, there's no automatic default constructor defined for you, just like C++. There are no copy-constructors, since all arguments are passed by reference.

19. There are no destructors in Java. There is no "scope" of a variable per se, to indicate when the object's lifetime is ended – the lifetime of an object is determined instead by the garbage collector. There is a **finalize( )** method that's a member of each class, like a destructor, but **finalize( )** is called by the garbage collector and is only supposed to be responsible for releasing resources. If you need something done at a specific point, you must create a special method and call it, not rely upon **finalize( )**. Put another way, all objects in C++ will be (or rather, should be) destroyed, but not all objects in Java are garbage collected. Because Java doesn't support destructors, you must be careful to create a cleanup method if necessary, and to explicitly call all the cleanup methods for the base class and member objects in your class.

20. Java has method overloading that works virtually identically to C++ function overloading.

21. Java does not support default arguments.

22. There's no **goto** in Java. The one unconditional jump mechanism is the **break** *label* or **continue** *label*, which is used to jump out of the middle of multiply-nested loops.

23. Java uses a singly-rooted hierarchy, so all objects are ultimately inherited from the root class **Object**. In C++ you can start a new inheritance tree anywhere, so you end up with a forest of trees. In Java you get a single ultimate hierarchy. This can seem restrictive, but it gives a great deal of power since you know that every object is guaranteed to have at least the **Object** interface. C++ appears to be the only OO language that does not impose a singly-rooted hierarchy.

24. Java has no templates or other implementation of parameterized types. There is a set of containers: **Vector**, **Stack** and **Hashtable** that hold **Object** references, and through which you can satisfy your container needs, but these containers are not designed for efficiency

like the C++ Standard Template Library (STL). For a more complete set of containers, there's a freely-available library called the *Generic Collection Library for Java* (www.ObjectSpace.com) which shows signs of eventually being incorporated into the standard Java language.

25. Garbage collection means memory leaks are much harder to cause in Java, but not impossible. However, many memory leaks and resouce leaks may be tracked to a badly written **finalize( )** or not releasing a resource at the end of the block where it is allocated (a place where a destructor would certainly come in handy). The garbage collector is a huge improvement over C++, and makes a lot of programming problems simply vanish. It may make Java unsuitable for solving a small subset of problems that cannot tolerate a garbage collector, but the advantage of a garbage collector seems to greatly outweigh this potential drawback.

26. Java has built in multithreading support. There's a **Thread** class that you inherit to create a new thread (you override the **run( )** method). Mutual exclusion occurs at the level of objects using the **synchronized** keyword as a type qualifier for methods. Only one thread may use a **synchronized** method of a particular object an any one time. Put another way, when a **synchronized** method is entered, it first "locks" the object against any other **synchronized** method using that object, and only "unlocks" the object upon exiting the method. There are no explicit locks; they happen automatically. You're still responsible for implementing more sophisticated synchronization between threads by creating your own "monitor" class.
Recursive **synchronized** methods work correctly. Time slicing is not guaranteed between equal priority threads.

27. Instead of controlling blocks of declarations like C++ does, access specifiers (**public**, **private** and **protected**) are placed on each definition for each member of a class. Without an explicit access specifier, the element defaults to "friendly," which means it is accessible to other elements in the same package (equivalent to them all being friends) but inaccessible outside the package. The class, and each method within the class, has an access specifier to determine whether it's visible outside the file. Sometimes the **private** keyword is used less in Java because "friendly" access is often more useful than excluding access from other classes in the same package (however, with multithreading the proper use of **private** is essential). The Java **protected** keyword means "accessible to inheritors *and* to others in this package." There is no equivalent to the C++ **protected** keyword which means "accessible *only* to inheritors" (**private protected** used to do this, but it was removed).

28. Nested classes. In C++, nesting a class is an aid to name hiding and code organization (but C++ namespaces eliminate the need for name hiding). Java packaging provides the equivalence of namespaces, so that isn't an issue. Java 1.1 has *inner classes* which look just like nested classes. However, an object of an inner class secretly keeps a handle to the object of the outer class that was involved in the creation of the inner-class object. This means that the inner-class object may access members of the outer-class object without qualification, as if those members belonged directly to the inner-class object. This provides a much more elegant solution to the problem of callbacks, solved with pointers to members in C++.

29. Because of inner classes described in the previous point, there are no pointers to members in Java.

30. No **inline** methods. The Java compiler may decide on its own to inline a method, but you don't have much control over this. You may suggest inlining in Java by using the **final** keyword for a method. However, **inline** functions are only suggestions to the C++ compiler, as well.

31. Inheritance in Java has the same effect as in C++, but the syntax is different. Java uses the **extends** keyword to indicate inheritance from a base class, and the **super** keyword to specify methods to be called in the base class that have the same name as the method you're in (however, the **super** keyword in Java only allows you to access methods in the

parent class, one level up in the hierarchy. Base-class scoping in C++ allows you to access methods that are deeper in the hierarchy). The base-class constructor is also called using the **super** keyword. As mentioned before, all classes are ultimately, automatically inherited from **Object**. There's no explicit constructor initializer list like in C++ but the compiler forces you to perform all base-class initialization at the beginning of the constructor body and it won't let you perform these later in the body. Member initialization is guaranteed through a combination of automatic initialization and exceptions for uninitialized object handles.

```
public class Foo extends Bar {
  public Foo(String msg) {
    super(msg); // Calls base constructor
  }
  public baz(int i) { // Override
    super.baz(i); // Calls base method
  }
}
```

32. Inheritance in Java doesn't change the protection level of the members in the base class. You cannot specify **public**, **private** or **protected** inheritance in Java as you can in C++. Also, overridden methods in a derived class cannot reduce the access of the method in the base class. For example, if a method is **public** in the base class and you override it, your overridden method must also be **public** (the compiler ensures this).

33. Java provides the **interface** keyword which creates the equivalent of an abstract base class filled with abstract methods and with no data members. This makes a clear distinction between something designed to be just an interface versus an extension of existing functionality using the **extends** keyword. It's worth noting that the **abstract** keyword produces a similar effect, in that you can't create an object of that class. However, an **abstract** class *may* contain abstract methods but it can also contain implementations, so it is restricted to single inheritance. Together with interfaces, this scheme prevents the need for some mechanism like virtual base classes in C++.
To create a version of the **interface** that can be instantiated, you use the **implements** keyword, whose syntax looks like inheritance:

```
public interface Face {
  public void smile();
}
public class Baz extends Bar implements Face {
  public void smile( ) {
    System.out.println("a warm smile");
  }
}
```

34. There's no **virtual** keyword in Java because all non-**static** methods always use dynamic binding. In Java, the programmer doesn't have to decide whether or not to use dynamic binding. The reason **virtual** exists in C++ is so you can leave it off for a slight increase in efficiency when you're tuning for performance (or, put another way, "if you don't use it you don't pay for it"), but this often results in confusion and unpleasant surprises. The **final** keyword provides some latitude for efficiency tuning – it tells the compiler that this method may not be overridden, and thus that it can be statically bound (and made inline, thus using the equivalent of a C++ non-**virtual** call). These optimizations are up to the compiler.

35. Java doesn't provide multiple inheritance (MI), at least not in the same sense C++ does. Like **protected**, MI seems like a good idea but you only really know you need it when you are face to face with the right design problem. Since Java uses a singly-rooted hierarchy, you'll probably run into fewer situations where MI is necessary. The aforementioned **interface** keyword takes care of combining multiple interfaces.

36. Run-time type identification functionality is quite similar to C++. To get information about handle **X** you can say, for example:

```
X.getClass().getName();
```

To perform a type-safe downcast you say:
`derived d = (derived)base;`
just like an old-style C cast. The compiler automatically invokes the dynamic casting mechanism without requiring extra syntax. Although this doesn't have the benefit of easy location of casts as in C++ "new casts," Java checks usage and throws exceptions so it doesn't allow bad casts like C++ does.

37. Exception handling in Java is different because there are no destructors. A **finally** clause can be added to force execution of statements that perform necessary cleanup. All exceptions in Java are inherited from the base class **Throwable**, so you're guaranteed a common interface.
```
public void f(Obj b) throws IOException {
  myresource mr = b.createResource();
  try {
    mr.UseResource();
  } catch (MyException e) {
    // handle my exception
  } catch (Throwable e) {
    // handle all other exceptions
  } finally {
    mr.dispose(); // special cleanup
  }
}
```

38. Exception specifications in Java are vastly superior to those in C++. Instead of the C++ approach of calling a function at run-time when the wrong exception is thrown, Java exception specifications are checked and enforced at compile-time. In addition, overridden methods must conform to the exception specification of the base-class version of that method: they can throw the specified exceptions, or exceptions derived from those. This provides much more robust exception-handling code.

39. There is method overloading, but no operator overloading in Java. The **String** class does use the **+** and **+=** operators to concatenate strings and **String** expressions use automatic type conversion, but that's a special built-in case.

40. The **const** issues in C++ are avoided in Java by convention. You only pass handles to objects, and local copies are never made for you automatically. If you want the equivalent of C++'s pass-by-value, you call **clone( )** to produce a local copy of the argument (although the **clone( )** mechanism is somewhat poorly designed). There's no copy-constructor that's automatically called.
To create a compile-time constant value, you say:
```
static final int SIZE = 255;
static final int BSIZE = 8 * SIZE;
```

41. Because of security issues, programming an "application" is quite different from programming an "applet." A significant issue is that an applet won't let you write to disk, because that would allow a program downloaded from an unknown machine to trash your disk. This changes somewhat with Java 1.1 digital signing, which allows you to unequivocally *know* everyone that wrote all the programs that have special access to your system (one of which may have trashed your disk; you still have to figure out which one and what to do about it…).

42. Since Java can be too restrictive in some cases, you may be prevented from doing important tasks like directly accessing hardware. Java solves this with *native methods* that allow you to call a function written in another language (currently only C/C++ are supported). Thus you can always solve a platform-specific problem (in a relatively non-portable fashion, but then that code is isolated). Applets cannot call native methods, only applications.

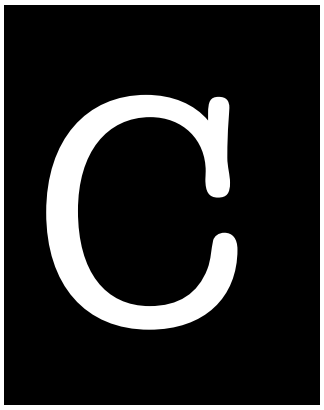
43. Java has built-in support for comment documentation, so the source code file can also contain its own documentation, which is stripped out and reformatted into HTML using a separate program. This is a boon for documentation maintenance and use.

44. Java contains standard libraries for solving specific tasks. C++ relies on non-standard third-party libraries. These tasks include (or will soon include):
    – Networking
    – Database Connection (via JDBC)
    – Multithreading
    – Distributed Objects (via RMI and CORBA)
    – Compression
    – Commerce
    The availability and standard nature of these libraries allow for more rapid application development.

45. Java 1.1 includes the JavaBeans standard, which is a way to create components that can be used in visual programming environments. This promotes visual components that can be used under all vendor's development environments. Since you aren't tied to a particular vendor's design for visual components, this should result in greater selection and availability of components. In addition, the design for JavaBeans is simpler for programmers to understand; vendor-specific component frameworks tend to involve a steeper learning curve.

46. If the access to a Java handle fails, an exception is thrown. This test doesn't have to occur right before the use of a handle; the Java specification just says that the exception must somehow be thrown. Many C++ runtime systems can also throw exceptions for bad pointers.

47. Generally, Java is more robust, via:
    – Object handles initialized to **null** (a keyword)
    – Handles are always checked and exceptions are thrown for failures
    – All array accesses are checked for bounds violations
    – Automatic garbage collection prevents memory leaks
    – Clean, relatively fool-proof exception handling
    – Simple language support for multi-threading
    – Bytecode verification of network applets



C: Java programming guidelines

This appendix is incomplete.

This appendix contains suggestions to help guide you while performing low-level program design, and also while writing code.

1. Capitalize the first letter of class names. The first letter of fields, methods and objects (handles) should be lowercase. All identifiers should run their words together, and capitalize the first letter of all intermediate words. For example
   ThisIsAClassName
   thisIsAMethodOrFieldName
   Capitalize all the letters of **static final** identifiers.
   Packages are a special case: they are all lowercase letters, even for intermediate words.

2. For each class you create, consider including a **main( )** that contains code to test that class. You don't need to remove the test code to use the class in a project, and if you make any changes you can easily re-run the tests.

3. If your class requires any cleanup when the client programmer is finished with the object, place the cleanup code in a single, well defined method. In addition, place a **boolean** flag in the class to indicate whether the object has been cleaned up or not. In the **finalize( )** method for the class, check to make sure the object has been cleaned up, and throw a class derived from **RuntimeException** if it hasn't, to indicate a programming error.

4. When overriding **finalize( )** during inheritance, remember to call **super.finalize( )** (this is not necessary if **Object** is your immediate superclass). You should call **super.finalize( )** as the *last* act of your overridden **finalize( )** rather than the first, to ensure that base-class components are still valid if you need them.

5. Constructors, exceptions, and **finalize( )**.
   You'll generally want to re-throw any exceptions that you catch in a constructor if it

causes failure of the creation of that object, so the caller doesn't blindly continue, thinking that the object was created correctly.

6. When you are creating a fixed-size collection of objects, transfer them to an array (especially if you're returning this collection from a method). This way you get the benefit of the array's compile-time type checking and the recipient of the array may not have to cast the objects in the array to use them.

7. Choose **interface**s over **abstract** classes. If you know something is going to be a base class, your first choice should be to make it an **interface**, and only if you're forced to have method definitions or member variables should you change to an **abstract** class.

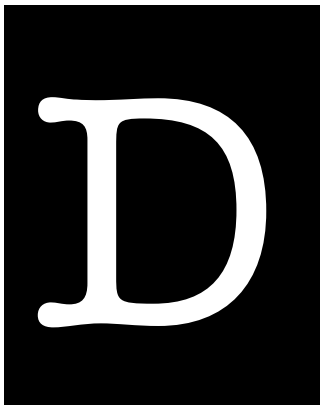
8. Inside constructors, do only what is necessary to set the object into the proper state. Actively avoid calling other methods (except for **final** methods) since those methods may be overridden to produce unexpected results during construction (see Chapter 7 for details).

9. Choose composition first when creating new classes from existing classes. Only if inheritance is required by your design should it be used. If you use inheritance where composition will work, your designs will become needlessly complicated.

10. Use inheritance to express differences in behavior, and member variables to express variations in state.

11. To avoid a highly frustrating experience, make sure there's only one class of each name anywhere in your classpath. Otherwise the compiler can find the identically-named other class first, and report error messages that make no sense. If you suspect you are having a classpath problem, try looking for **.class** files with the same names at each of the starting points in your classpath.

12. When using the event "adapters" in the new Java 1.1 AWT, there's a particularly easy pitfall you can encounter. If you override one of the adapter methods and you don't quite get the spelling right, you'll end up adding a new method rather than overriding an existing method. However, this is perfectly legal and so you won't get any error message from the compiler or run-time system – your code simply won't work correctly.



D: a bit about garbage collection

It's hard to believe that Java could possibly be as fast or faster than C++.

This assertion has yet to be proven to my satisfaction. However, I've begun to see that many of my doubts about speed come from early implementations that were not particularly efficient so there was no model at which to point and explain how Java can be fast.

Part of the way I've thought about speed has come from being cloistered with the C++ model. C++ is very focused on everything happening statically, at compile time, so that the run-time image of the program is very small and fast. C++ is also based very heavily on the C model, primarily for backwards compatibility, but sometimes simply because it worked a particular way in C so it was the easiest approach in C++. One of the most important cases is the way memory is managed in C and C++, and this has to do with one of my more fundamental assertions about why Java must be slow: in Java, all objects must be created on the heap.

In C++, creating objects on the stack is very fast because when you enter a particular scope the stack pointer is moved down once to allocate storage for all the stack-based objects created in that scope, and when you leave the scope (after all the local destructors have been called) the stack pointer is moved up once. However, creating heap objects in C++ is typically much slower because it's based on the C concept of a heap as a big pool of memory which (and this is essential) must be recycled. When you call **delete** in C++ the released memory leaves a hole in the heap and so when you call **new**, the storage allocation mechanism must go seeking to try to fit the storage for your object into any existing holes in the heap, or else you'll rapidly run out of heap storage. However, searching for available pieces of memory is why allocating heap storage has such a performance impact in C++, so it's far faster to create stack-based objects.

Again, because so much of C++ is based on doing everything at compile-time this makes sense. But in Java there are certain places where things happen more dynamically and it changes the model. When it comes to creating objects, it turns out that the garbage collector can have a significant impact on the speed of object creation. This sounds a bit odd at first – that storage release affects storage allocation – but it's the way some JVMs work and it means that allocating storage for heap objects in Java can be nearly as fast as creating storage on the stack in C++.

You can think of the C++ heap (and a slow implementation of a Java heap) as a yard where each object stakes out its own piece of turf. This real estate can become available sometime later and must be reused. In some JVMs, the Java heap is quite different; it's more like a conveyor belt which moves forward every time you allocate a new object. This means that object storage allocation is remarkably rapid. The "heap pointer" is simply moved forward into virgin territory, so it's effectively the same as C++'s stack allocation (of course, there's a little extra overhead for bookkeeping but it's nothing like searching for storage).

Now you might observe that the heap isn't in fact a conveyor belt and if you treat it that way you'll eventually start paging memory a lot (which is a big performance hit) and later run out. But the trick is that the garbage collector steps in and while it collects the garbage, it compacts all the objects in the heap so that you've effectively moved the "heap pointer" closer to the beginning of the conveyor belt and further away from a page fault. So the garbage collector rearranges things and makes it possible for the high-speed, infinite-free-heap model to be used while allocating storage.

To understand how this works, you need to get a little better idea of the way the different garbage collector (GC) schemes work. A very simple but very slow GC technique is reference counting. This means that each object contains a reference counter, and every time a handle is attached to an object the reference count is increased. Every time a handle goes out of scope or is set to **null**, the reference count is decreased. Thus, managing reference counts is a small but constant overhead that happens throughout the lifetime of your program. The garbage collector moves through the entire list of objects and when it finds one with a reference count of zero it releases that storage. The one complication is that if objects circularly refer to each other they can have non-zero reference counts while still being garbage. Locating such self-referential groups requires significant extra work on the part of the garbage collector. Reference counting is commonly used to explain one kind of garbage collection but it doesn't seem to be used in any JVM implementations.

In faster schemes, garbage collection is not based on reference counting. Instead, it is based on the idea that any non-dead object must ultimately be traceable back to a handle that lives either on the stack or in static storage. The chain may go through several layers of objects. Thus, if you start in the stack and the static storage area and walk through all the handles you'll find all the live objects. For each handle that you find you must trace into the object that it points to, and then follow all the handles in *that* object, tracing into the objects they point to, etc., until you've moved through the entire Web that originated with the handle on the stack or in static storage. Each object that you move through must still be alive. Note that there is no problem with detached self-referential groups – these are simply not found, and are therefore automatically garbage.

In the approach described here, the JVM uses an *adaptive* garbage-collection scheme, and what it does with the live objects that it locates depends on the variant currently being used. One of these variants is *stop-and-copy*. This means that, for reasons that will become apparent, the program is first stopped (this is not a background collection scheme). Then, each live object that is found is copied from one heap to another, leaving behind all the garbage. In addition, as the objects are copied into the new heap they are packed end-to-end, thus compacting the new heap (and allowing new storage to simply be reeled off the end as previously described).

Of course, when an object is moved from one place to another, the handles to that object must be changed. The handle that comes from tracing to the object from the heap or the static storage area can be changed right away, but there may be other handles pointing to this object that will be encountered later during the "walk," and these are fixed up as they are found (you could imagine a hash table mapping old addresses to new ones).

There are two issues that make copy collectors inefficient. The first is the idea that you have two heaps and you slosh all the memory back and forth between these two separate heaps, meaning you must maintain twice as much memory as you actually need. Some JVMs deal with this by allocating the heap in chunks as needed and simply copying from one chunk to another.

The second issue is the copying itself. Once your program becomes stable it may be generating little or no garbage. Despite that, a copy collector will still copy all the memory from one place to another, which is wasteful. To prevent this, some JVMs detect that no new garbage is being generated and switch to a different scheme (this is the "adaptive" part). This other scheme is called *mark and sweep*,

and it's what Sun's JVM uses all the time. For general use mark and sweep is fairly slow, but when you know you're generating little or no garbage it's very fast.

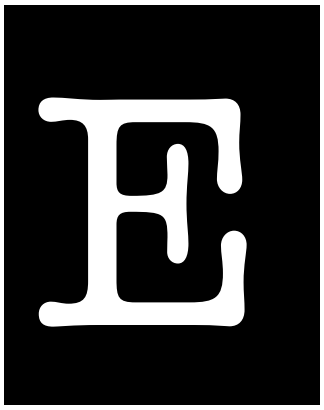
Mark and sweep follows the same logic of starting from the stack and static storage and tracing through all the handles to find live objects. However, each time it finds a live object that object is marked by setting a flag in it, but the object isn't collected yet. Only when the marking process is finished does the sweep occur. During the sweep, the dead objects are released. However, no copying happens so if the collector chooses to compact a fragmented heap it does so by shuffling objects around.

The "stop-and-copy" refers to the idea that this type of garbage collection is *not* done in the background; instead, the program is stopped while the GC occurs. In the Sun literature you'll find many references to garbage collection as a low-priority background process, but it turns out that this was a theoretical experiment that didn't work out. In practice the Sun garbage collector is run when memory gets low. In addition, mark-and-sweep requires that the program be stopped.

As previously mentioned, in the JVM described here memory is allocated in big blocks. If you allocate a large object, it gets its own block. Strict stop-and-copy requires copying every live object from the source heap to a new heap before you could free the old one, which translates to lots of memory. With blocks the GC can typically use dead blocks to copy objects to as it collects. Each block has a *generation count* to keep track of whether it's alive or not. In the normal case, only the blocks created since the last GC are compacted, all other blocks just get their generation count bumped if they have been referenced from somewhere. This handles the normal case of lots of short-lived temporary objects. Periodically, a full sweep is made – large objects are still not copied (just get their generation count bumped) and blocks containing small objects are copied and compacted. The JVM monitors the efficiency of GC and if it becomes a waste of time because all objects are long-lived then it switches to mark-and-sweep. Similarly, the JVM keeps track of how successful mark-and-sweep is and if the heap starts to become fragmented it switches back to stop-and-copy. This is where the "adaptive" part comes in, so you end up with a mouthful: "adaptive generational stop-and-copy mark-and-sweep."

There are a number of additional speedups possible in a JVM. An especially important one involves the operation of the loader and Just-In-Time (JIT) compiler. When a class must be loaded (typically, the first time you want to create an object of that class), the **.class** file is located and the byte codes for that class are brought into memory. At this point one approach is to simply JIT all the code, but this has two drawbacks: it takes a little more time which, compounded throughout the life of the program, can add up, and it increases the size of the executable (byte codes are significantly more compact than expanded JIT code) and thus may cause paging which definitely slows down a program. An alternative approach is *lazy evaluation* which means that the code is not JIT compiled until necessary. Thus, code that never gets executed may never get JIT compiled.

Because JVMs are external to browsers, you might expect that you could benefit from the speedups of some JVMs while using any browser. Unfortunately, JVMs don't currently interoperate with different browsers. Thus, to get the benefits of a particular JVM you must either use the browser with that JVM built in or run standalone Java applications.



E: recommended reading

Java Network Programming, by Elliote Rusty Harold, O'Reilly 1997.

Core Java, *2nd Edition*, by Cornell & Horstmann, Prentice-Hall 1997.

Inside Java, by Siyan & Weaver, (New Riders press, 1997). Not as tightly written and edited as it could be, but reasonably useful for finding out about the new Java 1.1 features.

Java Database Programming with JDBC, by Patel & Moss (Coriolis Books, 1997). An early book dedicated to JDBC.

Client/Server Programming with Java and CORBA by Orfali & Harkey (John Wiley & Sons, 1997). Compares the different approaches to distributed computing.

Design Patterns, by Gamma, Helm, Johnson & Vlissides (Addison-Wesley 1995). The seminal book that started the patterns movement in programming.