Java™ Application Development on Linux®

Java™ Application Development on Linux®

Carl Albing
Michael Schwarz

Prentice Hall Professional Technical Reference
Boston

Visit us on the Web: www.phptr.com

Library of Congress Cataloging-in-Publication Data:

CIP data on file.

ISBN 0–13–143697-X

Text printed in the United States on recycled paper at Pheonix Color in Hagerstown Maryland.

First printing, November 2004

To my mother, for her love of words.
—Michael

To my wife, Cynthia.
—Carl

Contents

Preface

JAVA AND LINUX

Why another book on Java? Why a book on Java and Linux? Isn't Java a platform-independent system? Aren't there enough books on Java? Can't I learn everything I need to know from the Web?

No doubt, there are a host of Java books on the market. We didn't wake up one morning and say, "You know what the world *really* needs? Another book about Java!" No. What we realized was that there are a couple of "holes" in the Java book market.

First, Linux as a development platform and deployment platform for Java applications has been largely ignored. This is despite the fact that the *nix platform (meaning all UNIX and UNIX-like systems, Linux included) has long been recognized as one of the most programmer-friendly platforms in existence. Those few resources for Java on Linux that exist emphasize tools to the exclusion of the Java language and APIs.

Second, books on the Java language and APIs have focused on pedagogical examples that serve to illustrate the details of the language and its libraries, but very few of these examples are in themselves practically useful, and they tend

to deal only with the issues of writing programs, and not at all with deploying and maintaining them. Anyone who has worked on a major software project, especially a software project that is developed and deployed in a business for a business, knows that designing and coding are only about half of the work involved. Yes, writing Java code is only slightly affected by the development and the deployment platform, but the process of releasing and maintaining such applications is significantly different between platforms.

To address these missing pieces, we decided to cover development and deployment of a Java application that has command-line, GUI, servlet, and enterprise components on a Linux platform. We're writing the guide book we wish we had had when we started writing and deploying Java applications on Linux. We're going to show you a simplistic enterprise application, "from cradle to grave," but along the way cover issues of design process, production environment, setup, administration, and maintenance that few books bother to cover.[1]

If you are considering buying this book and you are wondering if there is any information in here that you can't get for free on the Web, then, no. There is not. In fact, there is little information in any Java or Linux book that is not available for free on the Internet. In fact, in each of our chapters we will tell you where on the Web to find virtually all of the information we present, and then some. And yet books continue to sell, and we have the *chutzpah* to ask you to buy the book. The reason is that Web information is scattered, unorganized, and of highly variable quality. We will be trying to bring all the relevant information together in this book, in a clearly organized manner (and, we would like to believe, at an acceptably high level of quality). We think that has value.

Also, this book is part of the Bruce Perens' Open Source Series. This book is part of the Web literature. And you may freely read it and use it on the Web. We hope this book will be one of those you use on the Web and buy on paper. We don't know about you, but we like to use Web books for reference, but for reading, we like books. We own at least three books that are available for free on the Web: *Thinking in C++*, *Thinking in Java*, and O'Reilly's *Docbook: The Definitive Guide*. We hope that open publishing will be the new model.

---

1. This is not to say this book is without purely pedagogical examples. Especially in Part I we make use of your typical "throwaway" examples and single classes. To try to illustrate the basics with a complete application would obscure and confuse the points being illustrated.

FREE SOFTWARE AND JAVA

GNU/Linux[2] is Free Software. It is Open Source. I don't even want to start the debate on what each term means and which one is "right." One of the two authors of this book is a Free Software advocate, and the other is of a purely laissez-faire attitude towards the question (we won't tell you which, although we invite you to guess). But even with a deliberate decision to cease-fire, the question remains: Is Java Open Source or Free Software?

The answer is mixed. Neither Sun's nor IBM's Java implementations are Open Source or Free Software. You may download and use them for free, but you do not have the source code to them, nor do you have the right to make modifications to them.[3] This book will cover the GNU Compiler for Java, which compiles Java source code to native machine code. The GNU Compiler for Java (**gcj**) is both Open Source and Free Software. It is, however, supporting differing levels of the Java APIs (some packages are current, some are back at 1.1.x levels) and does not fully support the AWT or Swing GUIs.

However, none of this means that you cannot write your own Java programs and release them under a Free Software or Open Source license. So you can certainly develop Free Software in Java. Staunch Free Software partisans (such as Richard Stallman and the Free Software Foundation) would question the wisdom of doing so. Their argument would be that a Free Software product that depends on non-Free tools isn't really Free Software, since to compile, use, or modify it, you need to make use of a proprietary tool.

There is more than one effort to produce a Free Software Java runtime implementation. None of them is "ready for prime time." It would, in our opinion, be a very good thing for Sun to release their SDK and Java Virtual Machine as Free Software. But so far, they have steadily resisted calls to do so.

---

2. This is the only time we will refer to it as "GNU/Linux." See Section 7.3 for the story of why GNU/Linux is the preferred name of some. We understand Stallman and the FSF's position, but "Linux" is much easier on the eyes and ears than "GNU/Linux." And that, not principle, is how names and words go into the language. For better or for worse, "Linux" is the name of the operating system.

3. As we write this, a very public discussion is taking place between Sun, IBM, and Eric Raymond, founder of the Open Source Initiative, about opening Java under some sort of open source license. At this time, no one knows how this will turn out, but it is possible that Java will be Free Software in the future.

The fact, however, that two distinct vendors (Sun and IBM) produce effectively interchangeable development and runtime environments reduces some of the risk that you face when you select a platform available only from a single vendor who does not provide source code.

So, to put the case firmly: Java is free for use, but it is certainly not Free Software as defined in *The GNU Manifesto*[4] or the GNU General Public License.[5] This is a political and philosophical issue of interest only to those aforementioned Free Software partisans. For the rest of us, this has no bearing on Java's technical or business merits. As for us, obviously we like the language or we wouldn't be writing about it.

YOU CAN HELP!

This book is part of the Bruce Perens' Open Source Series. Shortly after this book is published in dead-tree form, it will be on the Web,[6] free for use, redistribution, and modification in compliance with the terms of the Open Publication License,[7] with no options taken. You can immediately create your own version as permitted in that license.

Naturally enough, we plan to maintain our "official" version of the online book, so we encourage you to send suggestions, corrections, extensions, comments, and ideas to us. Please send any such to `javalinux@multitool.net` and we will try to keep our little tome up-to-date so it continues to serve the needs of the Java and Linux development communities.

ACKNOWLEDGMENTS

First off, we naturally wish to thank Mark L. Taub, our acquisitions editor at Prentice Hall PTR, for believing in the book and in open publishing as the way to put it out there. We also want to thank Bruce Perens for lending his name and powers of persuasion to open-content publishing through the Prentice Hall PTR Bruce Peren's Open Source Series. Thanks, too, to Patrick Cash-Peterson

4. `http://www.gnu.org/gnu/manifesto.html`

5. `http://www.gnu.org/copyleft/gpl.html`

6. `http://www.javalinuxbook.com/`

7. `http://www.opencontent.org/openpub/`

and Tyrrell Albaugh, who worked as our in-house production contacts, for all the behind-the-scenes work they did, including overseeing the cover.

In more direct terms of content, we owe major thanks to Kirk Vogen of IBM Consulting in Minneapolis for his article on using SWT with **gcj**, and for his kind help in allowing us to use the ideas he first presented in his IBM developerWorks articles. In more direct terms of content, we owe major thanks to: Kirk Vogen of IBM Consulting in Minneapolis for his article on using SWT with **gcj**, and for his kind help in allowing us to use ideas he first presented in his IBM developerWorks articles; and to Deepak Kumar[8] for graciously allowing us to base our `build.xml` file for EJBs off of a version that he wrote.

Thanks, too, to Andrew Albing for his help in drawing some of our diagrams, and to George Logajan and to Andy Miller for sharing their insights on the more intricate details of Swing.

We also wish to express our great indebtedness to our technical reviewers, especially Andrew Hayes, Steve Huseth, and Dan Moore. A very large thank-you is also due to Alina Kirsanova whose eye for detail, endless patience, and tenacity, and overall talent with proofing, layout, and more added so much refinement and improvement to the book. We are greatful for all their contributions. Any errors or omissions in this text are our fault and certainly not theirs. The book is much stronger for all their efforts.

There are likely many more people we ought to thank, especially those at Prentice Hall PTR, whose names and contributions we may never know, but we do know that this was an effort of many more people than just the authors, and we are grateful to them all.

8. `http://www.roseindia.net/`

Introduction

This book has the unfortunate burden of serving a diverse set of audiences. We realize that this book might appeal to both experienced Java programmers who are new to Linux, and to experienced Linux programmers who are new to Java, with all possible shadings in between.

In addition to balancing these two poles, we are also trying to strike a balance between the size of the book and the range of our topic. Fortunately, there is today quite a range of both book and Web publishing on both Java and Linux, so we are able to do our best within the limits of a book a normal person may lift, and we can make recourse to a number of outside references you might wish to use to supplement our efforts.

WHO SHOULD BUY THIS BOOK

If you are an experienced Java programmer, but quite new to Linux, and you have been looking for information on the tools available to develop and deploy Java applications on Linux systems, this book will provide a lot of useful information.

If you are an experienced Linux user or developer, and you are interested in using the Java language on that platform, this book will guide you through some advanced Java development topics and will present, we hope, some novel uses for familiar Linux and GNU tools.

If you are a rank beginner to either Linux or Java, we still think this book has value, but we would recommend that you use it in conjunction with more introductory books. For a basic introduction to Java and object-oriented programming, we recommend Bruce Eckel's excellent book, *Thinking in Java* (ISBN 0-13-100287-2). For an introduction to Linux and its tools, we can recommend *The Linux Book* by David Elboth (ISBN 0-13-032765-4)[1] as an all-around title. We also list several other books in sections titled Resources throughout this book. Many books we recommend are not actually Linux-specific. Since Linux duplicates (in most respects) a UNIX platform, we do occasionally recommend books that are general to all *nix systems.

If you are a developer, contractor, or MIS development manager with more projects than budget, our book will introduce you to many solid tools that are free of license fees for the development and deployment of production Java applications. We are all being asked to do more with less all the time. In many (but certainly not all) cases, Free and Open Source software is an excellent way to do that.

WHO SHOULD NOT BUY THIS BOOK

Those looking for complete documentation on Java APIs and Linux-based Java application servers will be disappointed. Complete reference material on Free Software and Open Source Software may be found in book form, but it is most certainly out-of-date. And while this is an open-content book, we know full well that we will only be updating it as our "day jobs" permit. In other words, those seeking complete and current reference material should go to the Web.

Those who have a multimillion-dollar budget for applications development will probably be well served by commercial application server products. While we very much believe that Linux and Java on Linux are fully capable of supporting production environments, we recognize that products such as BEAWeblogic and IBM's WebSphere have large support organizations behind them, and

1. Note that we do tend to recommend titles from Pearson Education (our publishers), but that we by no means confine ourselves to that publisher.

(at least for now) a larger base of developers and contracting organizations with staff (variably) experienced in writing and supporting applications in these environments. Please note that you can run these products on Linux systems, and that they are part of the Linux-Java world. Our book does not cover them, however, both because they are well-covered elsewhere, and because we have chosen to emphasize the Free and Open Source tools merely to keep the book small enough to lift, while still covering those tools most in need of well-written supporting documentation.

HOW TO USE THIS BOOK

There are many approaches to a book. Some people like to start with the last chapter to see how it all turns out in the end; others like to start at the front and master each topic before moving on; some read through quickly, then reread for detail; still others prefer to skip around, "cherry picking" topics as whim and fancy strike. We hope this book will work for you, whatever your style.

Each chapter is not really free-standing, nor is it intricately tied to the previous chapters. If we were writing in depth on a single topic we might be able to build chapter by chapter. Instead, we've tackled an immense amount of information in hopes of condensing it down to give a good overview, to give you a glimpse of the possibilities, and to whet your appetite for more. Some chapters will be strongly related to previous chapters; others you may be able to read without having read any of the preceding chapters—it will depend on the topic.

Many Paths

What we're describing below are a few possible paths that you might take through the book, depending on what you bring to the task—your experience and skills, your patience and persistence. We have tried to pack a lot of useful and practical information into these few chapters, distilling down the most important topics for each subject area. We hope that, even for the most experienced of our readers, we still offer, if not some new facts, at least some fresh explanations that might give you new insight into familiar topics.

The Linux Newbie Path

If you are new to Linux, then you'll want to start with the first two chapters. If you are already experienced in Java, feel free to skip Chapter 3, but you may want at least to skim Chapters 4 and 5. You will definitely want to check out Chapter 7 as we are almost sure that it is something you didn't know about.

Chapter 8 is another topic you may not have encountered outside of Linux, although CVS is not limited to Linux environments. Beyond that, it will depend on what else you already know. See what other categories, below, might fit your situation.

The Java Newbie Path

If you are new to Java, then be sure to read Chapters 3 and 4, but if you are not already an experienced programmer you should probably bring along another, more introductory text.

Chapters 5 and 10 will give you some good background for choosing your Java development tools. So many Java projects these days are tied to Ant that you should also cover Chapter 9 if you don't already know the tool.

With your experience in other languages you may have done a lot of unit testing; read about the approach most popular with Java developers in Chapter 13.

The Client-Side Path

Depending on what type of Java development that you hope to do, you may want to concentrate on certain parts of the latter half of the book. Those most interested in the front end or client side should focus on the middle chapters. Of most interest to you will be Chapters 16 and 17. Your client-side emphasis should also include Chapters 18 and 19.

The Server-Side Path

For those with an emphasis on the middle and third tier, or those with a general server emphasis, all of Part IV will be helpful. This is in addition to a solid grounding in the previous chapters in Parts I and II.

The Enterprise Path

The final Part V will discuss enterprise scale software. Such software also typically includes JSP and Servlet software, as covered in Chapters 18, 19, and 20.

For those working at this level, the projects are usually large enough to be staffed with a variety of roles. Even if your role doesn't include the deployment of the software, we encourage you to read these chapters (20 and 24) so as to get some understanding of what is needed and how it fits together.

Now, let's get to work, and discover some of the amazing capabilities available to you when you combine two of the most powerful software trends in the history of computing—Java and Linux.

Part I

Getting Started

Chapter 1

An Embarrassment of Riches: The Linux Environment

The reader is introduced to the vast possibilities of the Linux command line, and excuses are made for its eclecticism.

1.1 What You Will Learn

Some basic shell commands are described in this chapter, especially those related to some common programming tasks. Used as a toolkit, they can be a handy collection of tools for everyday use.

Linux provides an incredible array of such tools, useful for any development effort, Java or otherwise. They will be important not only for the development of your Java code, but for all the hundreds of related housekeeping tasks associated with programming and with managing your development environment. A few tools are described briefly in this chapter, to hint at what can be done and to whet your appetite for more.

We will also describe a command which will help you learn about other commands. Even so, it may be quite worth your while to have another book about UNIX/Linux handy. If there is something you, as a programmer, need

to do on a Linux system, chances are there is already a command (or a sequence of commands) which will do it.

Finally, we will discuss the extent of our remaining ignorance upon finishing the chapter.

Let us take a moment to explain that last comment. As readers of computer books ourselves, we are often frustrated when we discover how lightly a topic has been covered, but particularly so when other parts of the same book are found to fully explore their topics. When only some parts of a book are thorough, you often don't know that you don't know it all. We will introduce some basic shell concepts and commands here, and we may expand on some of these in later chapters, but each of our chapters covers topics that could each fill its own book. Therefore we need to leave out lots of material. We will also let you know when we have left things out because they are off-topic, or because we don't have room. We'll also try to tell you where to look for the rest of the knowledge. We try to sum this up in a final section of each chapter entitled What You Still Don't Know. But we do have a lot of information to impart, so let's get going.

1.2 THE COMMAND LINE: WHAT'S THE BIG DEAL?

One of the revolutionary things that UNIX (and thus Linux) did was to separate operating system commands from the operating system itself. The commands to display files, show the contents of directories, set permissions, and so on were, in the "olden days," an integral part of an operating system. UNIX removed all that from the operating system proper, leaving only a small "kernel" of necessary functionality in the operating system. The rest became executables that lived outside of the operating system and could be changed, enhanced, or even replaced individually by (advanced) users without modifying the operating system. The most significant of these standalone pieces was the command processor itself, called the shell.

The shell is the program that takes command-line input, decides what program(s) you are asking to have run, and then runs those programs. Before there were Graphical User Interfaces, the shell was *the* user interface to UNIX. As more developers began working with UNIX, different shells were developed to provide different features for usability. Now there are several shells to choose from, though the most popular is **bash**. Some BSD/UNIX die hards

still swear by **csh**, a.k.a. the *C-shell*, though most of its best features have been incorporated into **bash**.

> **TIP**
>
> There are actually quite a few shells to choose from, and several editors for entering text. Our recommendation: If you learn only one shell, learn **bash**. If you learn only one editor, learn **vi**. Some basic shell scripting will go a long way to eliminating mundane, repetitive tasks. Some basic **vi** editing will let you do things so much faster than what GUI editors support. (More on editing in Chapter 2.)

Since commands could be developed and deployed apart from the operating system, UNIX and Linux have, over the years, had a wide variety of tools and commands developed for them. In fact, much of what is called *Linux* is really the set of GNU tools which began development as Open Source long before Linux even existed. These tools, while not technically part of the operating system, are written to work atop any UNIX-like operating system and programmers have come to expect them on any Linux system that they use. Some commands and utilities have changed over the years, some are much the same as they first were in the early days of UNIX.

Developers, encouraged by the openness of Open Source (and perhaps having too much free time on their hands) have continued to create new utilities to help them get their job done better/faster/cheaper. That Linux supports such a model has helped it to grow and spread. Thus Linux presents the first time user with a mind-boggling array of commands to try to learn. We will describe a few essential tools and help you learn about more.

1.3 Basic Linux Concepts and Commands

There are some basic Linux commands and concepts that you should know in order to be able to move around comfortably in a Linux filesystem. Check your knowledge of these commands, and if need be, brush up on them. At the end of the chapter, we list some good resources for learning more about these and other commands. Remember, these are commands that you type, not icons for clicking, though the windowing systems will let you set up icons to represent those commands, once you know what syntax to use.

So let's get started. Once you've logged in to your Linux system, regardless of which windowing system you are using—KDE, Gnome, Window Maker, and so on, start up an xterm window by running **xterm** (or even **konsole**) and you'll be ready to type these commands.[1]

1.3.1 Redirecting I/O

The second great accomplishment of UNIX,[2] carried on into its Linux descendants, was the concept of redirecting input and output (I/O). It was based on the concept of a standardized way in which I/O would be done, called *standard I/O*.

1.3.1.1 Standard I/O

A familiar concept to Linux developers is the notion of standard I/O. Virtually every Linux process begins its life with three open file descriptors—standard in, standard out, and standard error. Standard in is the source of input for the process; standard out is the destination of the process' output; and standard error is the destination for error messages. For "old fashioned" command-line applications, these correspond to keyboard input for standard in and the output window or screen for both standard out and error.

A feature of Linux that makes it so adaptable is its ability to redirect its I/O. Programs can be written generically to read from standard in and write to standard out, but then when the user runs the program, he or she can change (or redirect) the source (in) or destination (out) of the I/O. This allows a program to be used in different ways without changing its code.

Redirecting I/O is accomplished on the Linux shell command line by the "<" and ">" characters. Consider the **ls** program which lists the contents of a directory. Here is a sample run of **ls**:

```
$ ls
afile    more.data     zz.top
$
```

1. If you're not using a windowing system, these commands are typed at the shell prompt that you get after you log in. But if you're not using a windowing system, either you're not a beginner (and don't need this introduction) or you can't get your windowing system to work, in which case you may need more help that we can give you here.

2. Yes, we are aware that much of UNIX actually comes from the Multics project, but we credit UNIX with popularizing it.

We can redirect its output to another location, a file, with the ">" character:

```
$ ls > my.files
$
```

The output from the **ls** command no longer appears on the screen (the default location of standard out); it has been redirected to the file `my.files`.

What makes this so powerful a construct (albeit for a very simple example) is the fact that not only was no change to the program required, but the programmer who wrote the **ls** program also did nothing special for I/O. He simply built the program to write to standard out. The shell did the work of redirecting the output. This means that any program invoked by the shell can have its output similarly redirected.

Standard error is another location for output, but it was meant as the destination for error messages. For example, if you try to list the contents of a nonexistent directory, you get an error message:

```
$ ls bogus
ls: bogus: No such file or directory
$
```

If you redirect standard out, nothing changes:

```
$ ls bogus > save.out
ls: bogus: No such file or directory
$
```

That's because the programmer wrote the program to send the message to standard error, not standard out. In the shell (**bash**) we can redirect standard error by preceding the redirect symbol with the number 2, as follows:[3]

```
$ ls bogus 2> save.out
$
```

3. The use of the number 2 comes from an implementation detail: All the I/O descriptors for a UNIX process were kept in an array. The first three elements of the array, numbered 0, 1, and 2, were defined to be the standard in, out, and err, in that order. Thus in the shell you can also redirect standard out by using "1>" as well as the shorter ">".

Note there is no output visible from **ls**. The error message, `ls: bogus: No such file or directory`, has been written to the file `save.out`.

In a similar way standard input (`stdin`) can be redirected from its default source, the keyboard.

As an example, we'll run the `sort` program. Unless you tell it otherwise, `sort` will read from `stdin`—that is, the keyboard. We type a short list of phrases and then type a `^D` (a Control-D) which won't really echo to the screen as we have shown but will tell Linux that it has reached the end of the input. The lines of text are then printed back out, now sorted by the first character of each line. (This is just the tip of the iceberg of what `sort` can do.)

```
$ sort
once upon a time
a small creature
came to live in
the forest.
^D
a small creature
came to live in
once upon a time
the forest.
```

Now let's assume that we already have our text inside a file called `story.txt`. We can use that file as input to the `sort` program by redirecting the input with the "`<`" character. The `sort` doesn't know the difference. Our output is the same:

```
$ sort < story.txt
a small creature
came to live in
once upon a time
the forest.
```

1.3.1.2 Pipes

The output from one command can also be sent directly to the input of another command. Such a connection is called a *pipe*. Linux command-line users also use "pipe" as a verb, describing a sequence of commands as *piping* the output of one command into another. Some examples:

```
$ ls  | wc > wc.fields
$ java MyCommand < data.file | grep -i total > out.put
```

The first example runs **ls**, then pipes its output to the input of the **wc** program. The output of the **wc** command is redirected to the file `wc.fields`. The second example runs **java**, giving it a class file named `MyCommand`. Any input that this command would normally read from keyboard input will be read this time from the file `data.file`. The output from this will be piped into **grep**, and the output from **grep** will be put into `out.put`.

Don't worry about what these commands really do. The point of the example is to show how they connect. This has wonderful implications for developers. You can write your program to read from the keyboard and write to a window, but then, without any change to the program, it can be instructed to read from files and write to files, or be interconnected with other programs.

This leads to a modularization of functions into small, reusable units. Each command can do a simple task, but it can be interconnected with other commands to do more, with each pipeline tailored by the user to do just what is needed. Take **wc** for example. Its job is to count words, lines, and characters in a file. Other commands don't have to provide an option to do this; any time you want to count the lines in your output, just pipe it into **wc**.

1.3.2 The ls Command

The **ls** command is so basic, showing the names of files in a directory. Be sure that you know how to use these options:

- `ls` lists the files in a directory.
- `ls -l` is the long form, showing permissions, ownership, and size.
- `ls -ld` doesn't look inside the directory, so you can see the directory's permissions.
- `ls -lrt` shows the most recently modified files last, so you can see what you've just changed.

1.3.3 Filenames

Filenames in Linux can be quite long and composed of virtually any character. Practically speaking, however, you're much better off if you limit the length to something reasonable, and keep to the alphanumeric characters, period, and the underscore ("_"). That's because almost all the other punctuation characters have a special meaning to the shell, so if you want to type them, you need to *escape* their special meaning, or suffer the results of unintended actions.

Filenames are case sensitive—upper- and lowercase names are different. The files `ReadMe.txt` and `readme.txt` could both be in the same directory; they are distinct files.

Avoid using spaces in filenames, as the shell uses whitespace to delineate between arguments on a command line. You *can* put a blank in a name, but then you always have to put the name in quotes to refer to it in the shell.

To give a filename more visual clues, use a period or an underscore. You can combine several in one filename, too. The filenames `read_me_before_you_begin` or `test.data.for_my_program` may be annoyingly long to type, but they are legal filenames.

> **NOTE**
>
> The period, or "dot," in Linux filenames has no special meaning. If you come from the MS-DOS world, you may think of the period as separating the filename from the extension, as in `myprogrm.bas` where the filename is limited to eight characters and the extension to three characters. Not so in Linux. There is no "extension," it's all just part of the filename.
>
> You will still see names like `delim.c` or `Account.java`, but the `.c` or `.java` are simply the last two characters or the last five characters, respectively, of the filenames. That said, certain programs will insist on those endings for their files. The Java compiler will insist that its source files end in `.java` and will produce files that end in `.class`—but there is no special part of the filename to hold this. This will prove to be very handy, both when you name your files and when you use patterns to search for files (see below).

1.3.4 Permissions

Permissions in Linux are divided into three categories: the *owner* of a file (usually the user who created it), the *group* (a collection of users), and *others*, meaning everyone who is not the owner and not in the group. Any file belongs to a single owner and, simultaneously, to a single group. It has separate read/write/execute permissions for its owner, its group, and all others. If you are the owner of a file, but also a member of the group that owns the file, then the owner permissions are what counts. If you're not the owner, but a member of the group, then the group permissions will control your access to the file. All others get the "other" permissions.

If you think of the three permissions, read/write/execute, as three bits of a binary number, then a permission can be expressed as an octal digit—where the most significant bit represents read permission, the middle bit is write

permission, and the least significant bit is execute permission. If you think of the three categories, user/group/others, as three digits, then you can express the permissions of a file as three octal digits, for example "750". The earliest versions of this command required you to set file permissions this way, by specifying the octal number. Now, although there is a fancier syntax (for example, `g+a`), you can still use the octal numbers in the **chmod** command. See the example below.

The fancier, or more user-friendly, syntax uses letters to represent the various categories and permissions. The three categories of user, group, and other are represented by their first letters: `u`, `g`, and `o`. The permissions are similarly represented by `r`, `w`, and `x`. (OK, we know "x" is not the first letter, but it is a reasonable choice.) For both categories and permissions, the letter `a` stands for "all." Then, to add permissions, use the plus sign (`+`); to remove permissions, use the minus sign (`-`). So `g+a` means "add all permissions to the group category," and `a+r` means "add read permissions to all categories."

Be sure that you know these commands for manipulating permissions:

- **chmod** changes the *mode* of a file, where mode refers to the read/write/execute permissions.
- **chown** changes the owner of a file.[4]
- **chgrp** changes the group owner of a file.

Table 1.1 shows some common uses of these commands.

Table 1.1 Changing permissions

Command	Explanation
`chmod a+r file`	Gives everyone read permission.
`chmod go-w file`	Takes away write permission from group, others.
`chmod u+x file`	Sets up a shell script so you can execute it like a command.
`chmod 600 file`	Sets permission to read and write for the owner but no permissions for anyone else.

4. On Linux the use of this command is restricted to the superuser, or "root."

1.3.5 File Copying

Do you know these commands?

- **mv**
- **cp**
- **ln**

The **mv** command (short for "move") lets you move a file from one place in the hierarchy of files to another—that is, from one directory to another. When you move the file, you can give it a new name. If you move it without putting it in a different directory, well, that's just renaming the file.

- `mv Classy.java Nouveau.java`
- `mv Classy.java /tmp/outamy.way`
- `mv Classx.java Classz.java ..`
- `mv /usr/oldproject/*.java .`

The first example moves `Classy.java` to a new name, `Nouveau.java`, while leaving the file in the same directory.

The second example moves the file named `Classy.java` from the current directory over to the `/tmp` directory and renames it `outamy.way`—unless the file `outamy.way` is an already existing directory. In that case, the file `Classy.java` will end up (still named `Classy.java`) inside the directory `outamy.way`.

The next example just moves the two Java source files up one level, to the parent directory. The ". ." is a feature of every Linux directory. Whenever you create a directory, it gets created with two links already built in: ". ." points to its parent (the directory that contains it), and "." points to the directory itself.

A common question at this point is, "Why does a directory need a reference to itself?" Whatever other reasons there may be, it certainly is a handy shorthand to refer to the current directory. If you need to move a whole lot of files from one directory to another, you can use the "." as your destination. That's the fourth example.

The **cp** command is much like the **mv** command, but the original file is left right where it is. In other words, it copies files instead of moving them. So:

```
cp Classy.java Nouveau.java
```

will make a copy of `Classy.java` named `Nouveau.java`, and:

```
cp Classy.java /tmp
```

will make a copy of `Classy.java` in the `/tmp` directory, and:

```
cp *.java /tmp
```

will put the copies of all the Java sources in the current directory to the `/tmp` directory.

If you run this command,

```
ln Classy.java /tmp
```

you might think that **ln** copies files, too. You will see `Classy.java` in your present working directory and you will see what appears to be a copy of the file in the `/tmp` directory. But if you edit your local copy of `Classy.java` and then look at the "copy" that you made in the `/tmp` directory, you will see the changes that you made to your local file now also appear in the file in the `/tmp` directory.

That's because **ln** doesn't make a copy. It makes a *link*. A link is just another name for the same contents. We will discuss linking in detail later in the book (see Section 6.2.1).

1.3.6 Seeing Stars

We need to describe shell pattern matching for those new to it. It's one of the more powerful things that the shell (the command processor) does for the user—and it makes all the other commands seem that much more powerful.

When you type a command like we did previously:

```
mv /usr/oldproject/*.java .
```

the asterisk character (called a "star" for short) is a shorthand to match *any* characters, which in combination with the `.java` will then match any file in the `/usr/oldproject` directory whose name ends with `.java`.

There are two significant things to remember about this feature. First, the star and the other shell pattern matching characters (described below) do *not* mean the same as the regular expressions in **vi** or other programs or languages. Shell pattern matching is similar in concept, but quite different in specifics.

Second, the pattern matching is done by the shell, the command interpreter, *before* the arguments are handed off to the specific command. Any text with these special characters is replaced, by the shell, with one or more filenames that match the pattern. This means that all the other Linux commands (**mv**, **cp**, **ls**, and so on) never see the special characters—they don't do the pattern matching, the shell does. The shell just hands them a list of filenames.

The significance here is that this functionality is available to any and every command, including shell scripts and Java programs that you write, with no extra effort on your part. It also means that the syntax for specifying multiple files doesn't change between commands—since the commands don't implement that syntax; it's all taken care of in the shell before they ever see it. Any command that can handle multiple filenames on a command line can benefit from this shell feature.

If you're familiar with MS-DOS commands, consider the way pattern matching works (or doesn't work) there. The limited pattern matching you have available for a **dir** command in MS-DOS doesn't work with other commands—unless the programmer who wrote that command also implemented the same pattern matching feature.

What are the other special characters for pattern matching with filenames? Two other constructs worth knowing are the question mark and the square brackets. The "`?`" will match any *single* character.

The `[...]` construct is a bit more complicated. In its simplest form, it matches any of the characters inside; for example, `[abc]` matches any of `a` or `b` or `c`. So `Version[123].java` would match a file called `Version2.java` but not those called `Version12.java` or `VersionC.java`. The pattern `Version*.java` would match all of those. The pattern `Version?.java` would match all except `Version12.java`, since it has two characters where the `?` matches only one.

The brackets can also match a range of characters, as in `[a-z]` or `[0-9]`. If the first character inside the brackets is a "`^`" or a "`!`", then (think "not") the meaning is reversed, and it will match anything but those characters. So `Version[^0-9].java` will match `VersionC.java` but not `Version1.java`. How would you match a "`-`", without it being taken to mean a range? Put it first inside the brackets. How would you match a "`^`" or "`!`" without it being understood as the "not"? *Don't* put it first.

Some sequences are so common that a shorthand syntax is included. Some other sequences are not sequential characters and are not easily expressed as a range, so a shorthand is included for those, too. The syntax for these special

sequences is `[:name:]` where *name* is one of: `alnum`, `alpha`, `ascii`, `blank`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, `xdigit`. The phrase `[:alpha:]` matches any alphabetic character. The phrase `[:punct:]` matches any punctuation character. We think you got the idea.

1.3.6.1 Escape at Last

Of course there are always times when you want the special character to be just that character, without its special meaning to the shell. In that case you need to *escape* the special meaning, either by preceding it with a backslash or by enclosing the expression in single quotes. The commands `rm Account\$1.class` or `rm 'Account$1.class'` would remove the file even though it has a dollar sign in its name (which would normally be interpreted by the shell as a variable). Any character sequence in single quotes is left alone by the shell; no special substitutions are done. Double quotes still do some substitutions inside them, such as shell variable substitution, so if you want literal values, use the single quotes.

> **TIP**
>
> As a general rule, if you are typing a filename which contains something other than alphanumeric characters, underscores, or periods, you probably want to enclose it in single quotes, to avoid any special shell meaning.

1.3.7 File Contents

Let's look at a directory of files. How do you know what's there? We can start with an **ls** to list the names:

```
$ ls
ReadMe.txt   Shift.java  dispColrs  moresrc
Shift.class  anIcon.gif  jam.jar    moresrc.zip
$
```

That lists them alphabetically, top to bottom, then left to right, arranged so as to make the most use of the space while keeping the list in columns. (There are options for other orderings, single column, and so on.)

An `ls` without options only tells us the names, and we can make some guesses based on those names (for example, which file is Java source, and which

is a compiled class file). The long listing `ls -l` will tell us more: permissions, links, owner, group, size (in bytes), and the date of last modification.

```
$ ls -l
total 2414
-rw-r--r--    1 albing   users         132 Jan 22 07:53 ReadMe.txt
-rw-r--r--    1 albing   users         637 Jan 22 07:52 Shift.class
-rw-r--r--    1 albing   users         336 Jan 22 07:55 Shift.java
-rw-r--r--    1 albing   users        1374 Jan 22 07:58 anIcon.gif
-rw-r--r--    1 albing   users        8564 Jan 22 07:59 dispColrs
-rw-r--r--    1 albing   users        1943 Jan 22 08:02 jam.jar
drwxr-xr-x    2 albing   users          48 Jan 22 07:52 moresrc
-rw-r--r--    1 albing   users     2435522 Jan 22 07:56 moresrc.zip
$
```

While **ls** is only looking at the "outside" of files,[5] there is a command that looks at the "inside," the data itself, and based on that, tries to tell you what kind of file it found. The command is called **file**, and it takes as arguments a list of files, so you can give it the name of a single file or you can give it a whole long list of files.

> **NOTE**
>
> Remember what was said about pattern matching in the shell: we can let the shell construct that list of files for us. We can give `file` the list of all the files in our current directory by using the "`*`" on the command line so that the shell does the work of expanding it to the names of all the files in our directory (since any filename will match the star pattern).

```
$ file *
ReadMe.txt:  ASCII text
Shift.class: compiled Java class data, version 45.3
Shift.java:  ASCII Java program text
anIcon.gif:  GIF image data, version 89a, 26 x 26,
dispColrs:   PNG image data, 565 x 465, 8-bit/color RGB, non-interlaced
jam.jar:     Zip archive data, at least v2.0 to extract
moresrc:     directory
moresrc.zip: Zip archive data, at least v1.0 to extract
$
```

5. Technically, `ls` (without arguments) need only read the directory, whereas `ls -l` looks at the contents of the inode in order to get all the other information (permissions, size, and so on), but it doesn't look at the data blocks of the file.

The **file** looks at the first several hundred bytes of the file and does a statistical analysis of the types of characters that it finds there, along with other special information it uses about the formats of certain files.

Three things to note with this output from **file**. First, notice that `dispColrs` was (correctly) identified as a PNG file, even without the `.png` suffix that it would normally have. That was done deliberately to show you that the type of file is based not just on the name but on the actual contents of the file.

Second, notice that the `.jar` file is identified as a ZIP archive. They really do use a identical internal format.

Thirdly, **file** is not foolproof. It's possible to have perfectly valid, compilable Java files that **file** thinks are C++ source, or even just English text. Still, it's a great first guess when you need to figure out what's in a directory.

Now let's look at a file. This simplest way to display its contents is to use **cat**.

```
$ cat Shift.java
import java.io.*;
import java.net.*;
/**
 * The Shift object
 */
public class
Shift
{
  private int val;

  public Shift() { }

  // ... and so on

} // class Shift
```

When a file is longer than a few lines you may want to use **more** or **less** to look at the file.[6] These programs provide a screen's worth of data, then pause

6. Like any open marketplace, the marketplace of ideas and open source software has its "me-too" products. Someone thought they could do even better than **more**, so they wrote a new, improved and largely upward compatible command. They named it **less**, on the minimalist philosophy (with apologies to Dave Barry: "I am not making this up") that "less is more." Nowadays, the **more** is rather passe. The **less** command has more features and has largely replaced it. In fact, on many Linux distributions, **more** is a link to **less**. In the name of full

for your input. You can press the space bar to get the next screen's worth of output. You can type a slash, then a string, and it will search forward for that string. If you have gone farther forward in the file than you wanted, press "`b`" to go backwards.

To find out more about the many, many commands available, press `?` (the question mark) while it's running.

Typical uses for these commands are:

- To view one or more files, for example `more *.java`, where you can type `:n` to skip to the next file.
- To page through long output from a previous pipe of commands, for example, `$ grep Account *.java | more`, which will search (see more on **grep** below) for the string `Account` in all of the files whose names end in `.java` and print out each line that is found—and that output will be paginated by **more**.

If you need only to check the top few lines of a file, use **head**. You can choose how many lines from the front of the file to see with a simple parameter. The command `head -7` will write out the first seven lines, then exit.

If your interest is the last few lines of a file, use **tail**. You can choose how many lines from the end of the file to see; the command `tail -7` will write out the last seven lines of the file. But **tail** has another interesting parameter, `-f`. Though **tail** normally prints its lines and then, having reached the end of file, it quits, the `-f` option tells **tail** to wait after it prints the last few lines and then try again.[7] If some other program is writing to this file, then **tail** will, on its next read, find more data and print it out. It's a great way to watch a log file, for example, `tail -f /tmp/server.log`.

In this mode, **tail** won't end when it reaches the end of file, so when you want it to stop you'll have to manually interrupt it with a `^C` (Control-C— i.e., hold down the Control key and press the C key).

---

disclosure, there is also a paging program called **pg**, the precursor to **more**, but we'll say no more about that.

7. The **less** command has the same feature. If you press "F" while looking at a file, it goes into an identical mode to the `tail -f` command. As is often the case in the wacky world of Linux, there is more than one way to do it.

1.3.8 The grep Command

No discussion of Linux commands would be complete without mentioning **grep**. Grep, an acronym for "generalized regular expression processor," is a tool for searching through the contents of a file. It searches not just for fixed sequences of characters, but can also handle regular expressions.

In its simplest form, `grep myClass *.java` will search for and display all lines from the specified files that contain the string `myClass`. (Recall that the `*.java` expansion is done by the shell, listing all the files that end with `.java`.)

The first parameter to **grep**, `myClass` in the example above, is the string that you want to search for. But the first nonoption parameter to **grep** is considered a regular expression meaning that it can contain special characters for pattern matching to make for more powerful searches (see Section 2.2.3). Some of the most common option parameters for **grep** are listed in Table 1.2.

Here's a quick example:

```
grep println *.java | grep -v System.out
```

It will look for every occurrence of `println` but then exclude those that contain `System.out`. Be aware that while it will exclude lines like

```
System.out.println(msg);
```

it will also exclude lines like this:

```
file.println(msg);    // I'm not using System.out
```

It is, after all, just doing string searches.

Table 1.2 Options for **grep**

Option	Explanation
`-i`	Ignore upper/lower case differences in its matching.
`-l`	Only list the filename, not the actual line that matched.
`-n`	Show the line number where the match was found.
`-v`	Reverses the meaning of the search—shows every line that *does not* match the pattern.

1.3.9 The find Command

If someone compiled a list of the top 10 most useful Linux utilities, **find** would most likely be near the top of the list. But it would also make the top 10 most confusing. Its syntax is very unlike other Linux utilities. It consists of *predicates*—logical expressions that cause actions and have true/false values that determine if the rest of the expression is executed. Confused? If you haven't used **find** before you probably are. We'll try to shed a little light by showing a few examples.

```
find . -name '*frag*' -print
```

This command looks for a file whose name contains `frag`. It starts looking in the current directory and descends into all subdirectories in its search.

```
find /over/there . /tmp/here -name '*frag*.java' -print
```

This command looks for a file that has `frag` in its name and ends with `.java`. It searches for this file starting in three different directories—the current directory ("`.`"), `/over/there`, and `/tmp/here`.

```
find . -name 'My[A-Z]*.java' -exec ls -l '{}' \;
```

Starting in the current directory, this command searches for a file whose name begins with `My` followed by an uppercase alphabetic character followed by anything else, ending with `.java`. When it finds such a file, it will execute a command—in this case, the **ls** command with the `-l` option. The braces are replaced with the name of the file that is found; the "`\;`" indicates to **find** the end of the command.

The `-name` is called a predicate; it takes a regular expression as an argument. Any file that matches that regular expression pattern is considered `true`, so control passes on to the next predicate—which in the first example is simply `-print` that prints the filename (to standard out) and is always `true` (but since no other predicate follows it in this example, it doesn't matter). Since only the names that match the regular expression cause the `-name` predicate to be `true`, only those names will get printed.

There are other predicates besides `-name`. You can get an entire list by typing `man find` at a command prompt, but Table 1.3 lists a few gems, to give you a taste of what **find** can do.

Let's look at an example to see how they fit together:

Table 1.3 Some **find** predicates

Option	Explanation
`-type d`	Is `true` if the file is a directory.
`-type f`	Is `true` if the file is a plain file (e.g., not a directory).
`-mtime -5`	Is `true` if the file is less than five days old, that is, has been modified within the last five days. A `+5` would mean older than five days and a `5` with no sign means exactly five days.
`-atime -5`	Is `true` if the file was accessed within the last five days. The `+` and `-` mean greater and less than the specified time, as in the previous example.
`-newer myEx.class`	Is `true` if the file is newer than the file `myEx.class`.
`-size +24k`	Is `true` if the file is greater than 24K. The suffix `c` would mean bytes or characters (since `b` stands for 512-byte blocks in this context). The `+` and `-` mean greater and less than the specified size, as in the other examples.

```
$ find . -name '*.java' -mtime +90 -atime +30 -print
./MyExample.java
./old/sample/MyPrev.java
$
```

This command printed out the names of two files that end with `.java` found beneath the current directory. These files hadn't been modified in the last 90 days nor accessed within the last 30 days. The next thing you might want to do is to run this command again adding something at the end to remove these old files.

```
$ find . -name '*.java' -mtime +90 -atime +30 -print -exec rm '{}' \;
./MyExample.java
./old/sample/MyPrev.java
$
```

1.3.10 The Shell Revisited

Most Linux shells—the command interpreters—can be considered programming languages in their own right. That is, they have variables and control structures—`if` statements, `for` loops, and so on. While the syntax can be subtly different between shells, the basic constructs are all there.

Entire books can be—and have been—written on shell programming. (It's one of our favorite subjects to teach.) Programs written in the shell language are often called *shell scripts*. Such scripts can be powerful yet easy to write (once you are familiar with the syntax) and can make you very productive in dealing with all those little housekeeping tasks that accompany program development. All you need to do (dangerous words, no?) is to put commands in a text file and give the file execute permissions. But that's a subject for another day.

Some elements of shell scripting, however, are useful even if you never create a single shell script. Of these, perhaps the most important to know (especially for Java programmers) is how to deal with shell variables.

> **NOTE**
>
> We'll be describing the syntax for **bash**, the default shell on most Linux distributions. The syntax will differ for other shells, but the concepts are largely the same.

Any string of alphanumeric or underscore characters can be used as the name of a variable. By convention shell variables typically use uppercase names—but that is only convention (although it will hold true for most if not all of our examples, too). Since commands in Linux are almost always lowercase, the use of uppercase for shell variables helps them to stand out.

Set the value of a shell variable with the familiar method—the equal sign:

```
$ FILE=/tmp/abc.out
$
```

This has assigned the variable `FILE` the value `/tmp/abc.out`. But to make use of the value that is now in `FILE`, the shell uses syntax that might not be familiar to you: The name must be preceded with a "`$`".

Shell variables can be passed on to other environments if they are *exported*, but they can never be passed back up. To set a shell variable for use by your current shell and every subsequent subshell, export the variable:

```
$ export FILE
$
```

You can combine the assignment of a value with the exporting into one step. Since repeating the export doesn't hurt, you will often see shell scripts use the `export` command every time they do an assignment, as if it were part of the assignment syntax—but you know better.

```
$ export FILE="/tmp/way.out"
$
```

NOTE

The shell uses the dollar sign to distinguish between the variable name and just text of the same letters. Consider the following example:

```
$ echo first > FILE
$ echo second  > TEXT
$ FILE=TEXT
$ cat FILE
first
$
```

The `cat` command will dump the contents of the file named `FILE` to the screen—and you should see `first`. But how would you tell the shell that you want to see the contents of the file whose name you have put in the shell variable `FILE`? For that you need the "`$`":

```
$ cat $FILE
second
$
```

This is a contrived example, but the point is that shell syntax supports arbitrary strings of characters in the command line—some of them are filenames, others are just characters that you want to pass to a program. It needs a way to distinguish those from shell variables. It doesn't have that problem on the assignment because the "`=`" provides the needed clue. To say it in computer science terms, the "`$`" syntax provides the *R-value* of the variable. (Not the insulation R-value, but what you expect when a variable is used on the Right-hand-side of an assignment operator, as opposed to the *L-value* used on the Left-hand-side of an assignment operator.)

There are several shell variables that are already exported because they are used by the shell and other programs. You may need or want to set them to customize your environment. Since they are already exported, you won't need to use the `export` command and can just assign a value, but it doesn't hurt.

The most important shell variable to know is PATH. It defines the directories in the filesystem where the shell will look for programs to execute. When you type a command like **ls** or **javac** the shell will look in all of the directories specified in the PATH variable, in the order specified, until it finds the executable.

```
$ echo $PATH
/usr/local/bin:/usr/bin:/usr/X11R6/bin:/bin:.
$
```

The PATH shown in the example has five directories, separated by colons (":"). (Note the fifth one, the "."; it says to look in the current directory.) Where do you suppose it will find cat? You can look for it yourself by searching in each directory specified in PATH. Or you can use the **which** command:

```
$ which cat
/bin/cat
$
```

Some commands (like exit) don't show up, since they are built into the shell. Others may be aliases—but that opens a whole other topic that we aren't covering here. Just remember that each directory in the PATH variable is examined for the executable you want to run. If you get a *command not found* error, the command may be there, it just may not be on your PATH.

To look at it the other way around: If you want to install a command so that you can execute it from the command line, you can either always type its full pathname, or (a more user-friendly choice) you can set your PATH variable to include the location of the new command's executable.

So where and how do you set PATH? Whenever a shell is started up, it reads some initialization files. These are shell scripts that are read and executed as if they were typed by the user—that is, not in a subshell. Among other actions, they often set values for variables like PATH. If you are using **bash**, look at .bashrc in your home directory.

Shell scripts are just shell commands stored in a file so that you don't need to type the same commands and options over and over. There are two ways to run a shell script. The easiest, often used when testing the script, is

```
$ sh myscript
```

where `myscript` is the name of the file in which you have put your commands. (See Chapter 2 for more on how to do that.) Once you've got a script running the way you'd like, you might want to make its invocation as seamless as any other command. To do that, change its permissions to include the execution permission and then, if the file is located in a place that your `PATH` variable knows about, it will run as a command. Here's an example:

```
$ chmod a+rx myscript
$ mv myscript ${HOME}/bin
$ myscript
... (script runs)
$
```

The file was put into the `bin` directory off of the home directory. That's a common place to put homebrew commands. Just be sure that `$HOME/bin` is in your `PATH`, or edit `.bashrc` and add it.

If you want to parameterize your shell, you'll want to use the variables `$1`, `$2`, and so on which are given the first, second, and so on parameters on the command line that you used to invoke your script. If you type `myscript Account.java` then `$1` will have the value `Account.java` for that invocation of the script.

We don't have the space to go into all that we'd like to about shell programming, but let us leave you with a simple example that can show you some of its power. Used in shell scripts, `for` loops can take a lot of drudgery out of file maintenance. Here's a simple but real example.

Imagine that your project has a naming convention that all Java files associated with the user interface on your project will begin with the letters "UI". Now suppose your boss decides to change that convention to "GUI" but you've already created 200 or more files using the old naming convention. Shell script to the rescue:

```
for i in UI*.java
do
  new="G${i}"
  echo $i ' ==> ' $new
  mv $i $new
done
```

You could just type those commands from the command line—that's the nature of shell syntax. But putting them into a file lets you test out the script without having to type it over and over, and keeps the correct syntax once

you've got it debugged. Assuming we put those commands into a file called `myscript`, here's a sample run:

```
$ myscript
UI_Button.java  ==>  GUI_Button.java
UI_Plovar.java  ==>  GUI_Plovar.java
UI_Screen.java  ==>  GUI_Screen.java
UI_Tofal.java  ==>  GUI_Tofal.java
UI_Unsov.java  ==>  GUI_Unsov.java
...
$
```

Imagine having to rename 200 files. Now imagine having to do that with a point-and-click interface. It could take you all morning. With our shell script, it will be done in seconds.

We can't hope to cover all that we'd like to about shell scripting. Perhaps we have been able to whet your appetite. There are lots of books on the subject of shell programming. We've listed a few at the end of this chapter.

1.3.11 The tar and zip Commands

The **tar** and **zip** commands allow you to pack data into an archive or extract it back. They provide lossless data compression (unlike some image compression algorithms) so that you get back out exactly what you put in, but it can take up less space when archived.[8] Therefore **tar** and **zip** are often used for data backup, archival, and network transmission.

There are three basic actions that you can take with **tar**, and you can specify which action you want with a single letter[9] in the arguments on the command line. You can either

---

8. Well, technically, **tar** doesn't compress the data in the file, but it does provide a certain amount of "compression" by cutting off the tail ends of blocks of data; for example, a file of 37 bytes in its own file takes up 4K of disk space since disk blocks are allocated in "chunks" (not the technical term). When you **tar** together a whole bunch of files, those extra tail-end empty bytes are not used (except in the final block of the TAR file). So, for example, 10 files of 400 bytes could be packed into a single 4K file, instead of the 40K bytes they would occupy on the filesystem. So, while **tar** won't compress the data inside the file (and thus is quite assuredly "lossless") it does result in a smaller file.

9. Linux option strings always start with a "-", right? Yes, except for **tar**. It seems there is always an exception to every rule. The newer versions of **tar** allow the leading minus sign, but can also work without it, for historical compatibility reasons. Early versions of UNIX only had single

- `c`: Create an archive.
- `x`: Extract from an archive.
- `t`: Get a table of contents.

In addition, you'll want to know these options:

- `f`: The next parameter is the filename of the archive.
- `v`: Provide more verbose output.

Using these options, Table 1.4 shows examples of each of the basic functions.

Now let's do the same thing using the **zip** command (Table 1.5). There are actually two commands here—one to compress the files into an archive (**zip**), and the other to reverse the process (**unzip**).

Table 1.4 Examples of the **tar** command

Command	Explanation
`tar tvf packedup.tar`	Gives a table of contents, in long (or verbose) form. Without the `v`, all you get is the filenames; with the `v` you get additional information similar in format to the `ls -l` command.
`tar xvf packedup.tar`	Extracts all the files from the TAR file, creating them according to their specified pathname, assuming your user ID and file permissions allow it. Remove the `v` option if you don't want to see each filename as the file is extracted.
`tar cvf packedup.tar mydir`	Creates a TAR archive named `packedup.tar` from the `mydir` directory and its contents. Remove the `v` option if you don't want to see each filename as the file is added to the archive.

letter options. Newer POSIX versions of UNIX and the GNU tools, which means all flavors of Linux, also support longer full-word options prefixed with a double minus, as in `--extract` instead of `x` or `-x`.

Table 1.5 Examples of the **zip** and **unzip** commands

Command	Explanation
`unzip -l packedup.zip`	Gives a table of contents of the archive with some extra frill around the edges, like a count of the files in the archive.
`unzip packedup.zip`	Extracts all the files from the ZIP file, creating them according to their specified pathname, assuming your user ID and file permissions allow it. Add the quiet option with `-q` if you would like **unzip** not to list each file as it unzips it.
`zip -r packedup mydir`	Creates a ZIP archive named `packedup.zip` from the `mydir` directory and its contents. The `-r` tells **zip** to recursively descend into all the subdirectories, their subdirectories, and so on; otherwise, **zip** will just take the files at the first layer and go no deeper.

> **TIP**
>
> Since TAR and ZIP files can contain absolute as well as relative pathnames, it is a good idea to look at their contents (e.g., `tar tvf file`) before unpacking them, so that you know what is going to be written where.

There are many, many more options for **tar** and **zip** that we are not covering here, but these are the most common in our experience, and they will give you a good start.

The **tar** and **zip** commands are also worth knowing about by a Java developer because of their relationship to *JAR files*. If you are working with Java you will soon run across the notion of a Java ARchive file, or JAR file. They are recognizable by name, ending in `.jar`. Certain Java tools are built to understand the internal format of JAR files. For Enterprise Java (J2EE) there are similar archives known as WAR files and EAR files. The command syntax for dealing with the **jar** command that builds these archives is very similar to the basic commands of **tar**. The internal format of a **jar** is the same as a ZIP file. In fact, most places where you can use a JAR file you can use a ZIP file as well. (You will see more about this when we discuss the standard Java tools in Section 5.11.)

TIP

Here's one more handy example we know you'll use:

```
find . -name '*.java' -print | zip allmysource -@
```

This command starts in the current directory ("`.`") finding every file that ends in `.java` and gives their names to **zip** which will read them from standard in instead of its argument list (told to do so with the `-@` argument) and zip them all into an archive named `allmysource.zip`. To put it simply, it will zip up all your Java source files from the current directory on down.

1.3.12 The man Command

Primitive but handy, the **man** command (short for *manual*) was the early UNIX online manual. While we've come to expect (and ignore) online help, the idea of online manuals was rather revolutionary in the early days of UNIX. In contrast to walls of printed documentation, UNIX provided terse but definitive descriptions of its various commands. When they are done well, these descriptions are an invaluable handy reference. They are not the best way to learn about a command, but they can be a great guide to using the command's options correctly.

The format is simply `man` followed by the name of the command about which you want information. So `man man` will tell you about the **man** command itself.

The most useful option to **man** is the `-k` option. It will do a keyword search in the titles of all the manpages looking for the keyword that you give. Try typing `man -k java` to see what commands are available. The `(1)` means that it's a user command—something that you can type from the shell prompt, as opposed to `(2)` which is a system call or `(3)` which is a C library call. These numbers refer to the original UNIX documentation volumes (volume one was shell commands and so on), and it all fit into a single three ring binder.

TIP

One other way to find out something about a command, if you know the command name already, is to ask the command itself for help. Most commands have either a `-?` or `--help` option. Try `--help` first. If you need to type `-?` either put it in single quotes or type it with a backslash before the question mark, as in `-\?`, since the `?` is a pattern-matching character to the shell.

There are other help systems available, such as **info** and some GUI-based ones. But **man** provides some of the quickest and most terse help when you need to check the syntax of a command or find out if there is an option that does what you need.

1.4 Review

We've looked at commands that will show you where files are in your directory structure, show files' permissions and sizes, change the permissions, show you what is in a file, look for files by searching for strings, and look for files based on names or other properties.

Even so, we've given only the briefest coverage to only a few of the scores of Linux commands worth knowing. Tops among these is the shell, **bash** in our case. Whole books have been written on this subject, and you would do well to have one at hand.

1.5 What You Still Don't Know

The shell is a powerful language in its own right. While you think of it mostly as a command interpreter used for running other commands, it is, in fact, a language, complete with variables, logic and looping constructs. We are not suggesting that you write your application in shell scripts, but you will find it useful for automating many repetitive tasks. There is so much that can be done with shell scripts that we encourage you to read more about this and to talk with other Linux users.

Linux is replete with so many different commands. Some are powerful languages like **awk** and **perl**, others are simple handy utilities like **head**, **tail**, **sort**, **tr**, and **diff**. There are hundreds of other commands that we don't even have time to mention.

1.6 Resources

- Cameron Newham and Bill Rosenblatt, *Learning the Bash Shell*, O'Reilly Associates, ISBN 1565923472.
- Ellie Quigley, *Linux Shells by Example*, 4th ed., Prentice Hall PTR, ISBN 013147572X.

- Rafeeq Rehman and Christopher Paul, *The Linux Development Platform*, Prentice Hall PTR.
- Mark G. Sobell, *A Practical Guide to Linux*, Addison-Wesley, ISBN 0201895498.
- Mark G. Sobell, *A Practical Guide to Red Hat Linux*, Addison-Wesley, ISBN 0201703130.

Chapter 2

An Embarrassment of Riches: Editors

Here the joys of creating and changing text files are introduced, the rudiments of the venerable **vi** editor are presented, and the power of text is exalted.

2.1 What You Will Learn

Readers are encouraged, but not required, to plumb the depths of **vi**. Other text editor choices are briefly covered.

Remember our recommendation: If you learn only one shell, learn **bash**. If you learn only one editor, learn **vi**. Some basic shell scripting will go a long way to eliminating mundane, repetitive tasks. Some basic **vi** editing will let you do things much faster than you can with GUI editors.

2.2 EYE TO EYE WITH VI

Java programs consist of Java classes. Java classes are text files with Java statements and expressions. In order to write a Java program, then, you need to be able to enter text into a file. Sounds simple enough.

With Linux and its GNU tools, you have an amazing array of choices for how to do this. Some are GUI tools not unlike simple word processors. Others, like **vi** and Emacs, predate GUI tools, but provide much the same capability without the luxury (or annoyance) of a mouse or menus.[1]

The editor named **vi** (pronounced as you would spell it: "vee-eye") is one of the most enduring tools in Linux. Its popularity comes from a combination of power and ubiquity—you can find it on virtually every release of UNIX and Linux since 1985. But it is a powerful editor that can do a lot with only a few keystrokes.

There are actually several variants of **vi** from which to choose. Each is someone's attempt to go one better on **vi**, but all retain the same basic syntax and what you learn here will work equally well on any of the **vi** clones. You can choose among

- **elvis**
- **nvi**
- **vim**

Start up **vi** by typing the command name at a shell prompt, followed by the name of the file(s) that you want to edit:

```
$ vi Account.java
```

Keep in mind that **vi** was developed in the days of character-only video screens. Keyboards didn't always have arrow keys or other special characters, which have since been (largely) standardized by the advent of the IBM PC. In that situation, the authors of **vi** had only the alphabetic characters to use for all

1. We realize that **vi** is famous for being difficult to learn and nonintuitive. The UI design of **vi** dates back to earliest cursor-addressable display terminals. User interface design has come a long way since then. The **vi** UI does indeed show its age. But the program refuses to die. Why? Because while simple GUI editors make the easy stuff easy, **vi** makes the hard stuff easy. You can fall back on **pico** or **kate** or other GUI editors if you want, but bear with us. Mastering **vi** really does pay off.

of their commands . . . and did they make good use of those keys! Virtually every letter is used, both lower and upper case, to mean something unique in **vi**. But don't be put off by the large number of commands to learn; they fit some patterns that will make it easy for you to become proficient in a short time by learning a few commands and applying the patterns.

> **NOTE**
>
> If you really can't bear to part with your mouse and menus, try **gvim**. We haven't used it, but we hear that it has support for mice to help with cut and paste and the like. After you learn **vi** and get a little practice, though, you may find that you're never reaching for your mouse any more when you edit.

There are three modes to **vi**: the regular *vi mode*, some extended commands in the *ex mode*, and the *input mode*.

The simplest mode is the input mode. In input mode, every character you type becomes part of the text of the file. It's how you enter the bulk of the text for your Java programs. But **vi** doesn't start up in input mode; you have to "get into" input mode, and then get back out. Once out, you can use other **vi** commands to save the text and exit **vi**. More about those in a bit.

> **NOTE**
>
> Get out of input mode by pressing the Escape key. You can press it more than once, too, just to be sure that you are no longer in input mode. If you are no longer in input mode and you press escape, it will beep at you—with an audible or visual notification, depending on how your terminal window is set to respond.

In both **vim** and **elvis** (two popular **vi** clones) there is a status line at the bottom of the window that will show if you are in input mode. In **vim**, look in the lower left and **elvis**, the lower right. When in input mode, you will see a status word displayed like *insert* or *replace*.

In the vi mode, the default mode that you start in, all the keystrokes are interpreted as commands to the editor. They are *not* displayed as characters. So when we describe a command, such as `dt;`, you can type those three characters but will not see those characters on your screen. Instead you will see some action taken by **vi**—in this case it will delete text from your cursor up to the first semicolon on that line, if any (otherwise it will just beep).

There are several ways to get into input mode, depending on where you want to do the insert. When the file you're editing is completely empty, all these commands are equivalent, but for nonempty files, each command will begin input mode in a different place in the file:

- `i` inserts before the cursor.
- `I` inserts at the beginning of the line.
- `a` appends after the cursor.
- `A` appends at the end of the line.
- `o` "opens" a line for input after the line on which the cursor sits.
- `O` "opens" a line for input before the line on which the cursor sits.

Remember that this is character-based editing, before the days of mice and I-bars. So there is no meta-character for the cursor to show its position *between* two characters in the file. Instead, the cursor sits on top of a character, and thus inserts or appends will happen before or after that character.

Reminder: Get out of input mode by pressing the Escape key.

Next, let's move the cursor around. The simplest way to do that is one character at a time. Using the (lowercase) `h`, `j`, `k`, and `l` keys—notice that they're all in a row on QWERTY keyboards—you have the "arrow" keys for left, down, up, and right. One of the common enhancements for **vi** clones is to include support for the arrow keys on standard PC keyboards. Even so, the convenience of having the motion keys on the "home row" for touch typists can be a great speedup.

- `h` moves left one character.
- `j` moves down one line.
- `k` moves up one line.
- `l` moves right one character; same as a space.

Often, character- or line-at-a-time is too slow. Move to the beginning of the line that you are on with `0` (zero), or to the end of the line with `$`. Move to the top and bottom of the window with `H` (think "High") and `L` (think "Low"). So first type `L` then hold down `j`. To move back in a file, first type `H` then hold down `k`. That gets the display moving down or up respectively.

- `H` ("high") moves to the top line of the window.
- `M` ("middle") moves to the middle line of the window.

- `L` ("low") moves to the bottom line of the window.
- `0` moves to the beginning of the line.
- `$` moves to the end of the line.

This may still be too slow for you, especially if you are working your way through a large file. If you want to page up and down half a page at a time, try `^U` and `^D` (think "up" and "down"). To move a full page at each keystroke, try `^F` and `^B` (think "forward" and "back").

This may still take a while, especially if you want to get to the absolute beginning or end of the file. For those locations, type a three-character sequence, starting with a colon—which will jump your cursor to the status line of the window—then type either zero or the dollar sign, then press the Enter key. For example, `:$`.

So what's with the colon? Just when you thought you were getting the hang of the **vi** keystrokes, this odd pattern appears. It's called *ex mode*, and has to do with the history of **vi** being built atop the **ex** editor. Typing the colon got you back giving commands to **ex**, without the fancier screen-based GUI. (Even editors can have command lines.) There are many powerful search/replace commands that you can do from the ex command line; more on that later. For now, though, remember that you can type the colon, then a line number, then the Enter key and **vi** will position the cursor (and thus what is displayed on the screen) to that line number. The `0` and `$` are just special cases of that more generic way to position your place in the file.

Back to our positioning in the file. Recall that `h`, `j`, `k`, and `l` will move you one unit (char or line) at a time. Now enhance that motion by typing a number first, then the `h`, `j`, `k`, or `l`. So to move five lines up type `5k` (just be sure you use a lowercase letter). You can move way down in a file by typing something like `2000j` which will move down 2,000 lines. If the file doesn't have that many lines, you will find your cursor at the end of the file.

The point here is that almost any **vi** command can be preceded by a count, to repeat it that many times.

A few more navigation tips. Another useful way to move through text is a word at a time. You can move your cursor forward by a word with the letter `w` (for "word"). You can move "back" with the letter `b`. You can move five words at a time with `5w` or `5b`. See?

The definition of "word" to **vi** has to do with alphanumerics separated by whitespace, but also by certain punctuation characters. So to **vi**, the following Java code consists of how many words? Seven.

```
myArraylist.doSomething(magical); // cool
```

From the beginning of the line, you'd need to type `w` seven times (or know to type 7w—but how could you guess seven?) To help out, **vi** uses the uppercase `W` to skip words defined not by punctuation but solely by white space. Think of it as "bigger" words. And of course `B` will go "back" by these bigger words. So on our example line, a single `W` will get you to the start of the comment.

Be sure that you're not just reading these descriptions. Run **vi** on any file that you can find and practice navigating by lines or words or screens. Once you get the hang of it, it can be so much faster than reaching for the mouse and trying to maneuver the cursor into just the right spot between letters.

Sometimes you can see where you want to go based on the characters of text in the document. See that "x"? That's a relatively rare character on any line of text. If you see a character, your cursor can "find" it if you type `f` and then the character you are looking for. So `fx` would search forward on the line for an "x". And `Fx` would search backward from the cursor. To repeat the search, just type a semicolon (`;`).

Searching for a string is another good way to move your way through a file. To search forward, type a slash (`/`), then the characters for which you want to search, and end the string with a second slash and then Enter:[2]

```
/myArrayList/
```

To search backwards (towards the first line of the file) use the question mark rather than the slash to bracket your search string. In either case, to jump to the next occurrence, type `n`, or `27n` to jump to the 27th occurrence. Whether you are searching forward (`/`) or backward (`?`), using uppercase `N` will reverse the direction as it searches for the next occurrence. So, you can search forward with `/myVar/` and then press `n` for each next occurrence forward. If you go too far, just type `N` to back up. Similarly, if you were going backwards looking for an occurrence of a constructor, say something like: `?new HotClass?`; then each `n` will search toward the top of the file, and each `N` will search toward the end of file.

2. The second slash is optional in most **vi** implementations, but used for consistency with the same command in ex mode which has optional suffix characters.

In both cases, when you hit the top or bottom of the file, **vi** will wrap and keep searching from the opposite end, though a warning message will appear in the status bar.

OK, enough navigation. Let's start modifying text.

Copy and paste operations can be done easily on whole lines. Just *yank* and *put* the lines. You can yank a single line or several at a time (e.g., `7y`) and then a single put (`p`) will deposit a copy just after the current line (the line where your cursor sits). If you want to put the text before, not after, the current line, use uppercase `P`.

Go ahead. Try it on the file you're practicing on. It's the best way to get a feel for what we're describing here.

Cut and paste operations involve deleting the lines, not just copying them. This gets us into our third and final kind of syntax in **vi**, the double letter commands. Use `dd` to delete a line. Try it and you will find that the line your cursor is on just got deleted, and the cursor now rests comfortably on the next line. To paste that line back, use the same `p` or `P` that we used for to put the lines that we had copied (a.k.a. "yanked") above.

But why the `dd`? What's with the double letters? Think of "delete" for `d` and then add another letter to describe how much you want to delete—`dw` for "delete word" or `dW` for "delete the bigger words" (see above). So why `dd` for a line? We don't know for a fact, but we suspect that it's just for speed. You can also follow a `d` with the `h`, `j`, `k`, or `l` of our cursor movement, and that will delete either a character or a line in the appropriate direction.

A faster way (one keystroke, not two) to delete a single character is with the `x` key. And of course `5x`, or `27x`, will delete multiple characters. But if you're deleting many characters you will probably get it done faster by deleting "words" (`dw` or `dW`).

Another powerful way to delete text is to delete it from the cursor up *to* a specific character. The sequence `dt;` will delete from the cursor up to (but not including) the semicolon on the current line. If there is no semicolon, **vi** will beep, and no change will be made. To delete *from* the cursor up to and including the semicolon, use `df;`.

Everything you've just learned about delete is also true for change, the `c` in **vi**. You can combine it with itself (`cc`) to change a whole line. You can combine it with `w` to change a word (`cw`), or you can change from the cursor up to the next semicolon (`ct;`), and so on.

Change does the delete, then puts you in input mode. (Notice the reminder on the status line.) Remember, to get out of input mode and back into vi mode, press the Escape key.

Sometimes you've done too much, and you'd like to undo what you've just done. Typing `u` will undo the last change that you made. But here's a difference between the classic **vi** and some of the new, improved versions. In classic **vi**, if you type another `u`, then you are telling **vi** to undo what it just did—which was an undo. So the undo of an undo remakes the change that you had originally made. But in **vim**, "vi improved," typing `u` again and again will just keep undoing previous changes. If you want to undo the undo, in **vim**, you need to type `:redo` and then Enter.

The **vi** editor has shortcuts for helping you to change the indentation of your code. Typing two less-than signs (<<) will shift the line to the left; typing two greater-than signs (>>) will shift the line to the right. Typing a number first and then the less-than or greater-than signs will shift that many lines at once. But how far will they shift? The default is usually set at eight, but you can set it to any value you want. In ex mode you can set all sorts of values and flags, customizing **vi**'s operation. The value we're interested in here is `shiftwidth` which can be abbreviated `sw`. So the command would be `:set sw=4` if you want each shift to move by four characters. For more about this, and how to make it your default, see Section 2.2.4.

Any command that you do may be worth repeating. Say, you just shifted 14 lines and you'd like to shift them further. Or you just deleted five lines, and would like to delete five more. Well, you could just retype the command, but an easier way is just to type the period (`.`) and let **vi** repeat it for you.

2.2.1 Exiting

There are three ways of exiting **vi** (`ZZ`, `:q`, `:q!`) that you should know. The correct one to use depends on whether or not you *want* to save the changes to the file and on whether or not you *have* saved your changes.

- `ZZ` saves and quits in one step (three keystrokes).
- `:w` writes what you've been editing but doesn't quit.
- `:w` *`filename`* writes what you've been editing to a new file named *`filename`*; it will complain (and not write out anything) if the file already exists.

- `:7,.w! filename` writes lines from line 7 up to and including the current line to the named file, clobbering any previous contents (think of the "!" as meaning "and don't argue with me!").
- `:q` quits, provided you've saved your changes (e.g., with `:w`).
- `:q!` quits without saving any changes to the file.
- `:n` doesn't exit **vi**, but moves on to the next file if you started up with more than one file to edit (e.g., `vi Fir.java Pine.java`). When you've reached the last file in the list, you need to quit—for example, with `:q`.

2.2.2 Search and Replace

We've mentioned searching for a string with `/` or `?`, but what about replacing? Once you've located a string with `/`, you can use `cw` or `C` or `R` or other such commands to effect the change. Search for the next occurrence with `n`, and then you can repeat your change (the last `c`, `s`, `r`, and so on) by typing the period "`.`" that will repeat that last substitution, insert, and so on.

But what if you want to make 225 substitutions? Typing `n.n.n.n.n.n.` would get old after a while. Here, the ex mode, like any good command line, comes to the rescue to help with repetitive tasks.

If we want to search and replace all occurrences of one string for another, we can use the command

```
:1,$s/one string/another/
```

Almost all ex commands take an *address range*, that is, the lines of the file over which they will operate. If just one line number is given, the command will operate on that one line. Two numbers, separated by commas, represent the start and end lines—inclusive—of the operation. The first line is line 1, so a 0 as line number would mean "before the first line." The line where the cursor is currently located is just "`.`" (a period). The last line of the file can be represented by the dollar sign (`$`). You can even do some simple math on the addresses—for example, `.+2` meaning the second line in front of the cursor's current line.

> **TIP**
>
> There is a shortcut for the `1,$` address range. Use `%` to mean "all lines"—for example, `%s/one string/another/`.

Here are a few more substitution examples along with an explanation for each.

`.,$s/here/eternity/`
: From here to the end of the file, replace `here` with `eternity`.

`27,$-5s/lost/found/`
: From line 27 to the 5th line prior to the end of the file, replace `lost` with `found`.

`s/here/now/`
: Replace `here` with `now`, on the current line only.

Each line that has a match will do the substitution on only the *first* occurrence of the string. If you want to change all occurrences on those lines, you append a `g` (for "global" substitution) to the end of the command. Consider this snippet of Java:

```
class tryout
{
  int tryout;

  tryout(int startval) { // make a new tryout
    tryout = startval;
  } // tryout constructor

  // a tryout-like resetting
  public void
  setTryout(int toval) {
    tryout = toval;
  }
  // willfindtryoutinhere

} // class tryout
```

`1,$s/tryout/sample/`
: Works as expected except for line 5, where "tryout" appears as the constructor name but also in the comment.

`1,$s/tryout/sample/g`
: Works better (note the trailing `g`). But neither command can deal with "Tryout" in the `setTryout` method name. That's because of the uppercase "T", which doesn't match "tryout".

```
1,$s/Tryout/Sample/g
```

Will make the substitution in that method name.

> **TIP**
>
> Remember to precede these commands with a colon (“:”) to put you into ex mode which puts your cursor on the status bar of the window.

2.2.3 The Joy of Regular Expressions

The substitution command really becomes powerful when you start using regular expressions. Our examples so far have only had plain alphanumeric characters between the slashes of the substitution. But other characters take on special meanings inside the search and replace strings. Table 2.1 shows just a few.

From this small collection we can do some useful things. We show just a few in Table 2.2. All commands begin with `:1,$` to say that the substitution will be attempted from the first through the last line of the file.[3] You could use a smaller range for any of these substitutions, as we discussed above.

Table 2.1 Regular expression character meanings

Character	Meaning
`^`	The beginning of the line.
`$`	The end of the line.
`.`	Any single character.
`*`	Zero or more repetitions of the previous expression.
`+`	One or more repetitions of the previous expression.
`[]`	Any of the characters inside the brackets will match—e.g., `[abc]` matches any of `a`, `b`, `c`. Ranges are allowed too—e.g., `[a-z]`.
`&`	When used on the right-hand side, stands for whatever was found with the search string on the left-hand side (for an example, see Table 2.2).

3. Note that `%` is valid substitute for `1,$`.

Table 2.2 Some useful **vi** substitutions

Command	Explanation
`:1,$s/ *$//`	Removes all (any number of) trailing blanks—that is, looks for zero or more blanks followed immediately by the end of line, and replaces them with nothing (no characters between the last two slashes).
`:1,$s/^.*$/""/`	Puts quotes around the text of each and every line.
`:1,$s/^"//`	Removes the leading quote from any line that starts with one.
`:1,$s/"$//`	Removes the trailing quote from any line that ends with one.

There is so much more that could be said about regular expressions. They are one of the most powerful features for making big changes with few keystrokes. It's an integral part of **sed**, Perl, and other tools. It's in such demand that it has been added to Java for better pattern matching. See Section 2.6 for ways to learn more about regular expressions.

2.2.4 Starting Off Right: `.exrc`

You can preset certain behaviors in **vi** by putting ex commands in a file called `.exrc` in your home directory. Those commands will be read whenever you invoke **vi** and before you begin typing commands.

Here's a simple but useful `.exrc` example:

```
" set my favorite options:
set autoindent shiftwidth=4
set ignorecase
```

As you can see from the example, settings can be combined on one line. Note also that these lines do *not* begin with a colon. A colon *is* needed if you type these lines from within **vi**—because you need to get into ex mode; but since these are assumed to be ex commands (hence the name `.exrc`) they are going straight to the ex side of **vi** and no colon is needed. Comment lines begin with a double quote; the rest of the line is ignored, and doesn't need a matching quote.

The `ignorecase` command tells **vi** to ignore any difference between upper- and lowercase characters when searching for text with the `/` or `?` commands. The single character searches on the current line (`f` and `F`) are not

affected by this setting. The default for **vi** is `noignorecase`, which means case is significant.

The autoindent setting (can be abbreviated `ai`) means that when you do an `o` or `O` to *open* a line after or before (`o` versus `O`) the line on which your cursor rests, **vi** will automatically add whitespace so that the text that you enter begins at the same column where the current line begins.

For example, suppose you are editing an `if` statement like this:

```
if (userBalance < minDaily) {
    userAccount.chargeFees();
}
```

With your cursor on the middle line, if you type an `o` or `O`, the new (blank) line will open with your cursor at the fifth character position, right in line with the "u" of `userAccount`. If you find that your cursor is flush left when you try this, then you need to set autoindent. You can do this from within **vi** by typing `:set ai` or the longer `:set autoindent`. The leading ":" is important—it gets you to ex mode. (Don't forget to press Enter at the end of the command.)

The `shiftwidth` setting tells **vi** how many character positions to move text left or right with each `<<` (left) or `>>` (right) command, as well as when typing `^D` and `^T` in input mode. When typing text in input mode, people often use the Tab key to indent their text, for example inside an `if` or `for` statement. You can do this in **vi**, and the actual tab character will be the character in your text. But if you want tighter indenting, use the `^T` (that's Control-T, "T" for Tab, we suppose) to increase your indent and `^D` to decrease your indent while in input mode. The **vi** editor will automatically compute how much whitespace to use and will put an optimal combination of tabs and spaces as needed to line things up. Alternately, you can have **vi** always expand tabs into spaces and not mix tabs and spaces, but just use spaces, with `set expandtab`. (Remember to add a ":" if you want to type this from the command line.)

2.3 EDITORS GALORE

There are many editors available to a programmer on a Linux system. Some are text-based editors, typically antedating GUI interfaces. Many have graphical interfaces, with mouse-based cut and paste and the like. We will mention several here and encourage you to find the one with which you are comfortable.

The list of choices for editors is quite long. One of the beauties of the Open Source approach is that personal choices like favorite editors aren't squashed by arbitrary decisions: If you want, you can write an editor; others can adopt it.

Test drive a few; try them on for size. Remember that there can be a learning curve to climb. Don't necessarily settle for the easiest to learn—it may not be able to handle all that you'll need it to do, which may cost you more in the long run.

Speaking of editors that aren't easy to learn, we can't discuss editors without a mention of Emacs. To quote the GNU Emacs project home page:[4]

> Emacs is the extensible, customizable, self-documenting real-time display editor. If this seems to be a bit of a mouthful, an easier explanation is Emacs is a text editor and more. At its core is an interpreter for Emacs Lisp ("elisp," for short), a dialect of the Lisp programming language with extensions to support text editing. Some of the features of GNU Emacs include:
>
> - Content sensitive major modes for a wide variety of file types, from plain text to source code to HTML files.
> - Complete online documentation, including a tutorial for new users.
> - Highly extensible through the Emacs Lisp language.
> - Support for many languages and their scripts, including all the European "Latin" scripts, Russian, Greek, Japanese, Chinese, Korean, Thai, Vietnamese, Lao, Ethiopian, and some Indian scripts. (Sorry, Mayan hieroglyphs are not supported.)
> - A large number of extensions which add other functionality. The GNU Emacs distribution includes many extensions; many others are available separately—even a Web browser.

There is another variant of Emacs called XEmacs. It came from the same code base but split over differences both technical and philosophical.[5] Now if you thought that **vi** had a lot of obscure key sequences, you ain't seen nothin' yet. With its Lisp interpreter Emacs is incredibly extensible and powerful, but

4. This is from `http://www.gnu.org/software/emacs/emacs.html#Whatis`.

5. If you want to read more about those differences, and how they came about, see `http://www.xemacs.org/About/XEmacsVsGNUemacs.html`.

has a huge learning curve—which is why we aren't going to cover Emacs or XEmacs at all in this book.

Our favorite editor is still **vi**—in part, we're sure, because we already know it so well. But like any skilled craftsman, even though you may have a favorite hammer or saw that you use on most of your work, you will still have several others ready in your toolkit, and use specialized ones for certain tasks.

2.3.1 Editing Your Pipes (sed, the Stream EDitor)

One important kind of editor available on Linux is the *stream editor*, or **sed**. It allows you to perform editing on the data that comes in on standard in and writes its result to standard out. Similar to the syntax from ex mode in **vi** (and based on the simple **ed** editor), it can be very useful for making changes to large numbers of files in one go.

You can learn much more about **sed** from its manpage or from the book *UNIX AWK and SED Programmer's Interactive Workbook* by Peter Patsis.

2.3.2 Simple Graphical Editors

Linux comes with a wide range of open source software, not all of which is installed on every installation. You may need to use your Linux installation disks to add these programs to your system. Whether it's RedHat's package manager or SuSE's YaST2 or Debian's **apt-get**, most Linux admin interfaces make it easy to add these extra packages. Of course you can also resort to the Web for finding and downloading additional open source software.

Here's a quick listing of some of the many editors that you might find to your liking. The description of each is largely "in its own words," based on the text that the authors supply with their software.

- **jedit** is a cross-platform programmer's text editor written in Java. The Java-based portability seems appealing. This is a very powerful editor and a popular choice.
- **pico** is a small easy to use editor.
- **mbedit** is a multiplatform editor.
- NEdit is a GUI style text editor for workstations with X Window and Motif. NEdit provides all of the standard menu, dialog, editing, mouse support, as well as macro extension language, syntax highlighting, and a lot of other nice features (and extensions for programmers).

- **xcoral** comes up fast; seems well done. Half of the YaST developers swear by it, not only because of the built-in C/C++/Java browser. This editor provides support for C, C++, Java, Perl, Ada, and Fortran programs, as well as LaTeX and HTML documents. With the help of the built-in SMall ANSI C Interpreter (SMAC), **xcoral** can be configured and extended in almost arbitrary ways. Examples can be found in the directory `/usr/lib/xcoral` (or wherever **xcoral** is installed on your system). Further information about **xcoral** and SMAC is available in the detailed online help system (also available in HTML and PostScript format).
- **axe** features multiple windows, multiple buffers, configurable menus and buttons, access to external filters, keyboard macros, comprehensive online help, and more.
- **eddi** is an X editor based on the TiX shell, with syntax highlighting and several other useful features.
- **the**: If you're an IBMer from the heyday of mainframes, perhaps you've used **xedit** from VM/CMS. If so, you might want to check out **the**, whose name is the acronym of "The Hessling Editor."
- JED is an extremely powerful but small Emacs-like editor for programmers that is extensible in a C-like macro language and is able to perform color syntax highlighting. Among the many features: Emacs, WordStar, EDT emulation; C, Fortran, TeX, text editing modes; full undo; Emacs-compatible info reader, and lots more. It claims to be 8-bit clean, so you can even edit binary files.
- Glimmer is the editor formerly known as CodeCommander. It is a full featured code editor with many advanced features, including full scripting integration with either Python or Guile.
- **joe** (Joe's own editor) is a freeware ASCII editor for UNIX. **joe** is similar to most IBM PC text editors. The keyboard shortcuts are similar to WordStar and Turbo C. When **joe** has several files opened at the same time, each file is displayed in its own window. Additionally, **joe** supports shell windows whereby the output of the executed commands is saved in a buffer, automatic filename completion (via Tab), help windows, undo/redo, search and replace using regular expressions.
- gEdit is a small but powerful text editor designed expressly for GNOME. It supports a split-screen mode and plug-ins that make it even more powerful. Developers can develop their own plug-ins if they desire.

- **fte** is an editor "with many features and simple usage for X11 and console."
- **e3** is a very tiny editor (only .07MB) that offers many different modes such as **vi**, Emacs, and WordStar. The default mode is WordStar.
- **asedit** is a simple ASCII text editor for X11/Motif. Mouse support, dialog boxes, hypertext online help, undo/redo. Named for its author, Andrzej Stochniol.

2.4 REVIEW

We've given a good foundation for using **vi**—a set of commands that will help you with much of your daily editing. While not as pretty as a GUI tool, **vi** can be much more productive once you get familiar with the commands. Start with some basic commands, then refer to this chapter or another **vi** resource and learn a new keystroke each week. By next year, you'll be a master at **vi**—and incredibly productive at producing code.

We also described several other editors available under Linux. If you know them already, or are wed to your mouse, then try one on for size. The choice of an editor can be as much about personality and "fit" as it is a technical choice.

2.5 WHAT YOU STILL DON'T KNOW

There is still a lot more to learn about regular expressions. They may take a bit of practice, but it is a skill that can be used in a variety of contexts, in a variety of languages.

2.6 RESOURCES

- Rafeeq Rehman and Christopher Paul, *The Linux Development Platform*, Prentice Hall PTR, especially Chapter 2 on editors **vim**, Emacs, and **jed**.
- Peter Patsis, *UNIX AWK and SED Programmer's Interactive Workbook*, Prentice Hall PTR, ISBN 0130826758.

Chapter 3

An Experienced Programmer's Introduction to Java

Here the reader is rapidly acquainted with the manner in which Java implements the OO (Object-Oriented) concepts. The language's statements are unceremoniously presented. Much deference is paid to other texts in print and on the Web, since this is well traveled ground. We then present a simple sample Java application that will be used throughout the rest of this introductory part as an example that can be easily built in all of the Java environments available for Linux.

3.1 What You Will Learn

- Java syntax and semantics for the familiar (to an experienced programmer) programming constructs.
- How Java implements the OO: buzzwords of inheritance, encapsulation, and polymorphism.
- How Java deals with the absence of C++-style multiple inheritance.

- Why the absence of templates in Java is not as crippling as a C++ programmer might suppose.
- How `final` is better than `virtual` and how interfaces are often better than multiple inheritance.

This is going to be a whirlwind tour. Our book assumes that you already know programming in general, and have had some exposure to OO programming. We are going to distill into a single chapter material that comprises significant portions of other books. In particular, if there are concepts here that you are not already familiar with, look at Chapters 1–9 of Bruce Eckel's wonderful book, *Thinking in Java*, 3rd ed., published by Prentice Hall PTR (ISBN 0-131-00287-2). It is, genuinely, one of the best books on the market for learning the Java language and the design principles Java embodies.

If you are somewhat new to programming, but technically quite adept (maybe a system administrator or database administrator with little formal programming background), you may want to supplement your reading with a book that, unlike Eckel's, is targeted more toward the novice programmer. We like *Java Software Solutions: Foundations of Program Design*, 3rd ed., by John Lewis and William Loftus, Addison-Wesley, 2003 (ISBN 0-201-78129-8). It will introduce the concepts behind the programming constructs, whereas we will assume that you know these concepts so we can focus only on the Java syntax.

3.2 FUNDAMENTAL LANGUAGE ELEMENTS

Before the object-oriented structures, Java (like C) has a small number of fundamental statements and (again, like C and C++) some fundamental "nonobject" data types.[1]

1. The existence of these nonobject data types is another thing that brings up criticism of the Java language. Since Java does not have C++'s operator overloading features, you cannot use objects in standard algebraic expressions. I'm not sure if the inclusion of scalar classes was motivated by speed, or by the lack of operator overloading. Whatever the reason, like any other design compromise, it has both advantages and disadvantages, as we shall see throughout the book.

3.2.1 Scalar Types

Java has a number of built-in scalar (in this context, nonobject) types. We discuss these below.

3.2.1.1 Integer Types

Java defines four integer types—`byte`, `short`, `int`, and `long`. Unlike some languages, Java defines the precision, that is, the bit size, of these types.

- `byte`: 8 bits
- `short`: 16 bits
- `int`: 32 bits
- `long`: 64 bits

For Java, the goal is "compile once, run anywhere." Defining that `int` means 32 bits—everywhere—helps to achieve this goal. By contrast, when C language was first defined, its goal was different: to be available quickly on a variety of architectures, not to produce object code that would be portable between architectures. Thus, for C, the choice was up to the compiler developer to choose a size that was most "natural" (i.e., convenient) for that particular architecture.[2] This would make it easiest on the compiler writer. It succeeded—C was an easy language to implement, and it spread widely.

Back to Java. Note that all these values are signed. Java has no "unsigned" type.

Note also that `byte` is listed here. It can be treated as a numeric value, and calculations performed on it. Note especially that it is a *signed* number (i.e., values range from –128 to 127 and not from 0 to 255). Be careful when promoting a `byte` to an `int` (or other numeric value). Java will sign-extend on the promotion. If the value in the `byte` variable was a character (e.g., an ASCII value) then you really wanted it treated like an unsigned value. To assign such a value to an `int` you'll need to mask off the upper bits, as in Example 3.1.

You may never encounter such a situation, but if you are ever working with bytes (e.g., byte arrays) and start to mess with the individual bytes, don't say we didn't warn you.

---

2. In fact, C's only rule is that a `short int` will not be longer than an `int` and a `long` will not be shorter than an `int`. It is both ANSI and K&R compatible for all integer types in a C compiler to be the same size!

Example 3.1 Coercing a `byte` to `int` as if the `byte` were unsigned

```
byte c;
int ival;
...
ival = ((int) c) && 0xFF;  // explicit cast needed
```

3.2.1.2 Floating Point Types

Java provides two different precisions of floating point numbers. They are:

- `float`: 32 bits
- `double`: 64 bits

The `float` type is not very useful at that precision, so `double` is much more commonly seen. For other situations where precision is important, but you can spare some cycles, consider the `BigDecimal` and `BigInteger` object classes.

Java floating point numbers are specified to follow the IEEE floating point standard, IEEE 754.

3.2.1.3 Other Types

Java also has a `boolean` type, along with constants `true` and `false`. In Java, unlike C/C++, `boolean` values are a distinct type, and do not convert to numeric types. For example, it is common in C to write:

```
if (strlen(instr)) {
 strcpy(buffer, instr);
}
```

In this case, the integer result of `strlen()` is used as a `boolean`, where `0` is `false` and any other value is `true`. This doesn't work in Java. The expression must be of a `boolean` type.

Java also has a `char` type, which is *not* the same as a `byte`. The `char` is a character, and in Java, characters are represented using Unicode (UTF-16). They take two bytes each.

For more discussion on the differences between `byte`s and `char`s and about Unicode, read the Java tutorial on the `java.sun.com` Web site or visit `www.unicode.org`, the international standard's Web site.

3.2.1.4 Operators

Before we move on to the topic of arrays (which are sort of a hybrid scalar/object type in Java), let's spend a moment on the operators that can be used in expressions (Table 3.1). Most deal with numeric or boolean operands. For completeness, we'll include the operators that deal exclusively with arrays (the "`[]`") and classes ("`.`", `new`, and `instanceof`), even though we haven't discussed them yet.

Operators listed on the same row in the table have the same precedence. Operators with the same precedence, except for the unary operators, group from left to right. Unary operators group from right to left.

3.2.1.5 Arrays

Example 3.2 demonstrates the array syntax in Java.

Example 3.2 Example array syntax

```
int [] oned = new int[35];                // array = new type[size]
int alta [] = {1, 3, 5, 14, 11, 6, 24};  // alternative syntax plus
                                          // initialization
int j=0;

for(int i=0; i<35; i++) {
  oned[i] = valcomp(i, prop, alta[j]);    // array[index]
  if (++j > alta.length) {                // array.length
    j = 0;
  }
}
```

The array can be declared with the `[]` on either side of the variable name. While our example uses the primitive type `int`, array syntax looks just the same for any objects.

Note that in Java, one doesn't declare the size of the array. It's only in creating the array with a `new` that the array gets created to a particular size. (The `{...}` syntax is really just a special compiler construct for what is essentially a `new` followed by an assignment of values.)

Multidimensional arrays follow the syntax of simple arrays, but with additional adjacent square brackets, as shown in Example 3.3.

Table 3.1 Arithmetic and logical Java operators in order of precedence

Operators	Explanation
`[]  .`	array indexing, member reference
`-  ++ -- ! ~`	unary operators: negate, increment, decrement, logical-not, bitwise-not
`(`*`type`*`) new`	coercion, or casting to a different type; creating a new object
`* / %`	multiplication, division, remainder
`+ -`	addition, subtraction
`<< >> >>>`	shift-left, shift-right-sign-extend, shift-right-zero-fill
`< > <= >= instanceof`	less-than, greater-than, less-or-equal, greater-or-equal, comparing object types
`== !=`	equal, not-equal
`&`	bitwise-and (boolean for boolean operands with no short-circuit)*
`^`	bitwise-xor (with boolean operands it is a boolean-xor)**
`|`	bitwise-or (boolean for boolean operands with no short-circuit)*
`&&`	logical-and (with short-circuit)*
`||`	logical-or (with short-circuit)*
`?:`	Inline `if` expression, e.g., `a ? b : c` says, if `a` is `true`, then the value is `b`, else it is `c`.
`= += -= *= /= %= <<= >>= >>>= &= ^= |=`	Assignment; those with an operator, as in `a op= b` will perform the operation `a op b` then assign the result back to `a`.

* In Java there are two ways to do a boolean AND operation: using `&` or `&&`. Remember that for "a AND b", if either is `false`, then the result is `false`. That means that if "a" is `false`, there is no need to evaluate "b" because it will not affect the result. Skipping the evaluation of "b" in this case is called *short-circuiting*. Java will use short-circuit evaluation when using the `&&` operator, but not `&`. The same applies to the OR operators `||` and `|` where Java can short-circuit on a `true` evaluation of the first operand for `||`. This is an important distinction when "a" and "b" are not just simple variable references but rather method calls or other complex expressions, especially ones with side effects.

** XOR is exclusive or, where the result of "a XOR b" is `true` if "a" or "b" is `true`, but not both. For bitwise operands, "a" and "b" refer here to bits in the operand; for boolean operands it is the one value. Examples: `5^6` is `3`; `true^false` is `true` but `true^true` is `false`.

Example 3.3 Example two-dimensional array syntax

```
int [][] ragtag = new int[35][10];

for (int i=0; i<35; i++) {
  for (int j=0; j<10; j++) {
    ragtag[i][j] = i*j;
  } // next j
} // next i
```

Multidimensional arrays are built as arrays of arrays. Therefore, we can actually allocate it in a piecemeal fashion and have *ragged-edged* arrays, where each row has a different number of columns, as shown in Example 3.4.

Example 3.4 Ragged two-dimensional array syntax

```
int [][] ragtag = new int[17][];

for (int i=0; i<17; i++) {
  ragtag[i] = new int[10+i];
} // next i

for (int i=0; i<17; i++) {
  System.out.println("ragtag["+i+"] is "+ragtag[i].length+" long.");
} // next i
```

For a fuller discussion of arrays, see Chapter 9 of Eckel or Chapter 6 of Lewis&Loftus.

3.2.2 Object Types

The real power in Java, or any object-oriented language, comes not from the scalar types, cool operators, or powerful control statements it provides (see below), but from its *objects*.

Object-oriented programming is a relatively recent innovation in software design and development. Objects are meant to embody the real world in a more natural way; they give us a way to describe, in our programs, the real-world objects with which we deal. If you are programming a business application, think of real-world business objects such as orders, customers, employees,

addresses, and so on. Java is an object-oriented programming language and thus has some significant syntax related to OO concepts.

If you are new to object-oriented programming, be sure to read Chapter 1 of Eckel's *Thinking in Java*.

In Java, we define a *class* to represent the objects about which we want to program. A class consists of the data and the methods to operate on that data. When we create a new instance of some class, that instance is an object of that type of class. Example 3.5 shows a simple class.

Example 3.5 Simple class

```
class
PairInt
{
  // data
  int i;
  int j;

  // constructors
  PairInt() { i=0; j=0; }
  PairInt(int ival, int jval) { i=ival; j=jval; }

  // methods
  setI(int val) { i=val; }
  setJ(int val) { j=val; }
  int getI() { return i; }
  int getJ() { return j; }
}
```

Note that this class defines both data (`i`, `j`) and methods (`setI()`, `getJ()`, and so on). We put all this into a file named `PairInt.java` to match the name of the class definition.

If some other Java code wanted to create and use a `PairInt` object, it would create it with the `new` keyword followed by a call to a constructor (Example 3.6).

This example shows only a snippet of code, not the entire `PairInt` class. That class, though, would likely reside in its own source file (named for its class name). In Java you normally create lots of files, one for each class. When it's

Example 3.6 Using simple class

```
// declare a reference to one:
PairInt twovals;

// now create one:
twovals = new PairInt(5, 4);

// we can also declare and create in one step:
PairInt twothers = new PairInt(7, 11);
```

time to run the program, its various classes are loaded as needed. We'll discuss grouping classes together and how Java locates them in Section 3.3.1.

In Java, each source file contains one class and the file is named after that class. It is possible to define *inner* classes located inside another class definition and thus inside its file, but that introduces other complexities that we wish to avoid discussing at this point. Most importantly, an inner class has access to even the private members of the enclosing class. (Read more about inner classes in any of the Java books that we recommend at the end of this chapter.)

For each of the class methods, class data declarations, and the class itself, Java has syntax to limit the *scope*, or visibility, of those pieces. The examples above didn't include those keywords—that is, they took the default values. Usually you'll want to specify something. See Section 3.4.1.

3.2.2.1 Objects as References

So far we have not explained something important about object type variables. These variables can all be thought of as pointers or references to an object. When you declare a variable of an object type, what you are declaring is a variable that is capable of referring to an object of that type. When declared, it does not point at anything. It has a value of *null* and any attempt to use it will result in a null pointer *exception* (more on those later).

Before an object variable might be used, it must be made to refer to an instance of an object. This is done by assignment. You can assign an existing object, or you can use the `new` operator.

Any new class will have a *constructor*, that is, a method whose name is the name of the class. There can be many different constructors for the same class,

each with unique types of parameters. For example, the `String` class has many different constructors, including one which constructs a new `String` from a different `String` and another that constructs a new `String` from an array of bytes.

```
String strbystr = new String(oldstr);
String strbyarr = new String(myByteArray);
```

3.2.2.2 Strings

One of the most commonly used classes is the `String` class. It comes already defined as part of Java and has some special syntax for initialization which makes it look familiar. Whereas other objects need a `new` keyword and a constructor call, a `String` object can be created and initialized with the intuitive double quotes, as in:

```
String xyz="this is the stringtext";
```

The compiler also makes a special allowance for `String`s with respect to the plus sign (+). It can be used to concatenate two `String`s into a third, new `String`.

```
String phrase = "That is"
String fullsent = phrase + " all.";
```

It is worth noting that `String`s do not change—they are *immutable*. When you assign a `String` the value of one `String` plus another, there's a lot of `String` object creation going on behind the scenes. If you need to do a lot of concatenation of `String`s, say inside loops, then you should look into the use of the `StringBuffer` object. See Appendix A of *Thinking in Java*, 3rd ed., the section titled Overloading "+" and the `StringBuffer`, for a full discussion of the tradeoffs here.

There are a variety of methods for `String`—ones that will let you make substrings, search for substrings at the start or end or anywhere in the string, or check for equality of two strings.

Table 3.2 shows some of the most useful methods associated with `String` objects.

Table 3.2 Useful `String` methods

Return type	Method	Description
`int`	`length()`	Returns the length, i.e. number of characters, in the `String`.
`boolean`	`equals(Object obj)`	Returns `true` if the object is a `String` object and is equal to the `String`. (Aside: the argument takes a generic `Object` type rather than only a `String` object because it's meant to override the `equals()` method in the class `Object` of which `String` is a descendant.) This is the way to compare two `String`s to see if they are both holding the same sequence of characters. Using `stringA == stringB` will only tell you if `stringA` and `stringB` are referencing the same object (pointing to the same location in memory). What you typically want is `stringA.equals(stringB)`.

`boolean`	`equalsIgnoreCase(String str)`	Similar to `equals()`, but this one only allows a `String` parameter, and it ignores the upper/lower case distinction between letters. For example:
		`String sample = "abcdefg";`
		`String sample2 = "AbCdEfG";`
		`sample.equalsIgnoreCase(sample2)`
		returns `true`.

`String`	`toLowerCase()`	Returns a string with all characters converted to lowercase.
`String`	`toUpperCase()`	Returns a string with all characters converted to uppercase.
`boolean`	`startsWith(String substr)`	Returns `true` if the `String` starts with the given substring.
`boolean`	`endsWith(String substr)`	Returns `true` if the `String` ends with the given substring.
`String`	`substring(int index)`	Returns a string starting at position `index` to the end of the `String`.

Table 3.2 *(Continued)*

Return type	Method	Description
`String`	`substring(int first, int last)`	Returns a string starting at position `first` and up to, but not including, character position `last`. If `last` is greater than the length of the `String`, or `last` is less than `first`, it throws an `IndexOutOfBounds` exception.

3.2.2.3 Other Classes: Reading Javadoc

Java comes with a huge collection of existing classes for you to use. The simplest ones are just wrappers for the primitive classes. There is an `int` primitive data type, but Java provides an `Integer` class, so that you can have an integer as an object. Similarly, there are classes for `Long`, `Float`, `Boolean`, and so on. Such classes aren't nearly as interesting as the myriad other classes that come with Java. These others provide objects for doing I/O, networking, 2D and 3D graphics, graphical user interfaces (GUIs), and distributed computing. Java provides ready-to-use classes for strings, math functions, and for special kinds of data structures like trees and sets and hash tables. There are classes to help you with the manipulation of HTML, XML, and SQL, as well as classes for sound, music, and video. All these objects can be yours to use and enjoy if you just learn the magic of reading *Javadoc*—online documentation for Java classes. The documentation for all these classes is viewed with a Web browser. (In a following chapter we'll describe how you can make Javadoc documents for the classes that you write, too.)

The online version of the API documentation can be found at

```
http://java.sun.com/j2se/1.4.2/docs/api/
```

for Java 1.4.2. (Similarly, put 1.5.1 or whatever version you want at the appropriate place in the URL.) When displayed, it shows a three-frame page, as seen in Figure 3.1, except that we've overlaid the image with three labels: A, B, and C.

The upper left frame of the Javadoc display, the area labeled with A in our figure, lists all the packages that are part of Java 2 Standard Edition (J2SE). While there are many other packages of classes available for Java, these classes are the standard ones available without any other class libraries, with no

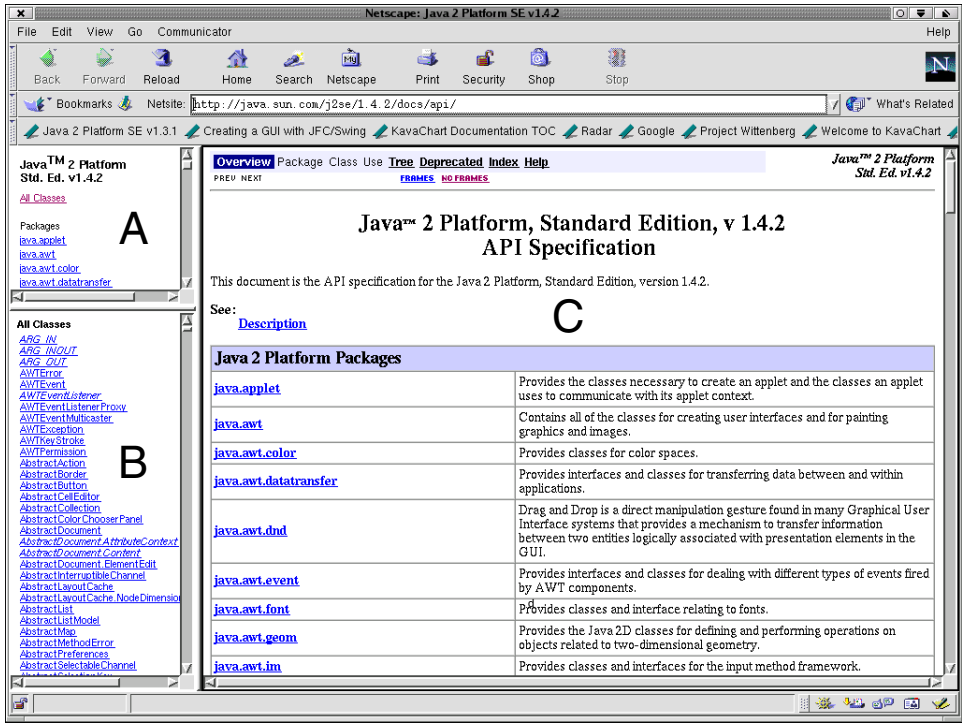

Figure 3.1 The three frames of a Javadoc page

additional downloads necessary. Other classes are documented in the same way—with Javadoc—but they are downloaded and displayed separately.

Frame B initially lists all the classes and interfaces available in all of the packages. When you select a package in A, B will display only those interfaces and classes that are part of the chosen package.

Frame C starts out with a list and description of all packages. Once you have selected a package in A, C will show the overview of that package, showing its classes and interfaces with descriptions.

But C is most often used to display the detailed description of a class. Choose a class or interface in B and you will see C filled with its description—some opening information followed by a list of the visible members of that class, followed by the possible constructors for that class and all the methods in that class (Figure 3.2). Each method is shown with its parameters and a one-sentence description. Clicking on the method name will open a fuller description (Figure 3.3).

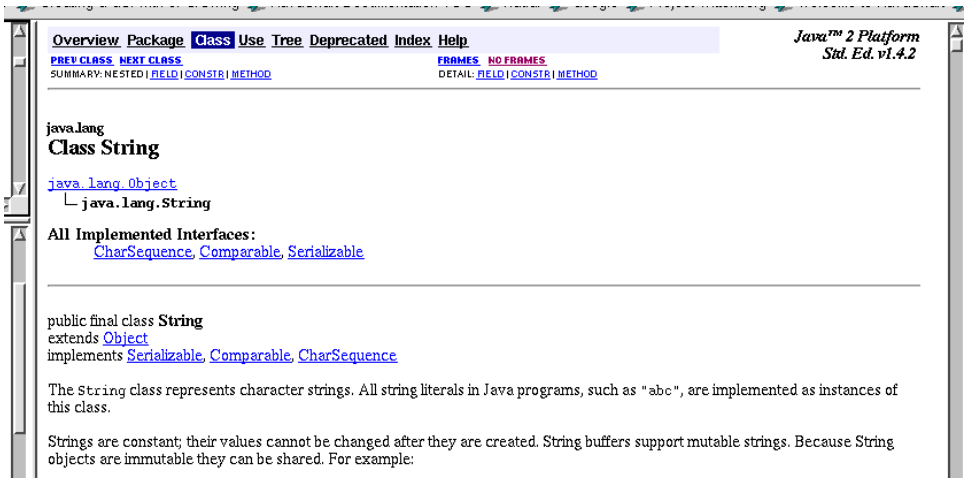

Figure 3.2 Javadoc display of class information

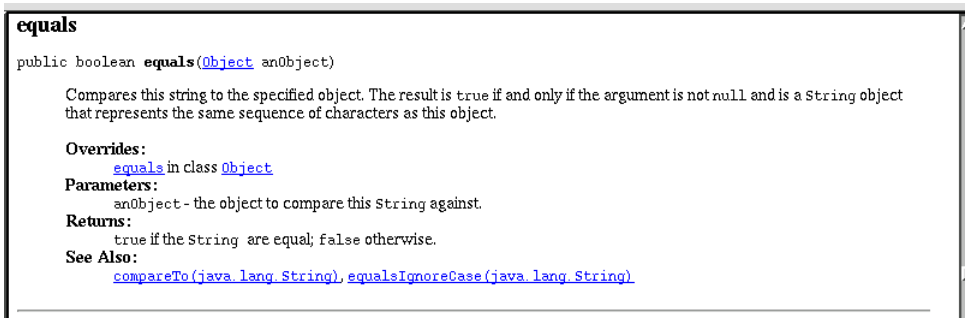

Figure 3.3 Javadoc display of a single method

Since you will likely be referencing the Javadoc pages regularly, you may want to download a copy to your hard drive. From the same page on the `java.sun.com` Web site where you can download the Java SDK you can also download the API documentation.

If you agree to the licensing terms, you will download a large ZIP file. Installing the documentation, then, is just a matter of unzipping the file—but it's best if you put it in a sensible location. If you have installed your Java SDK into a location like `/usr/local/java` then `cd` into that directory and unzip the file that you downloaded. Assuming that you saved the downloaded file into `/tmp`, a good place to put temporary files, and assuming that you have installed your version of Java into `/usr/local/java` and that you have write

permission in that directory (check the permissions with `ls -ld .`) then you can run these commands:

```
$ cd /usr/local/java
$ unzip -q /tmp/j2sdk-1_4_2-doc.zip
```

There may be quite a pause (tens of seconds) while it unzips everything. The **unzip** command will spew out a huge list of filenames as it unpacks them unless you use the `-q` option ("quiet") on the command line (which we did, to avoid all that). The files are all unzipped into a directory named `docs`. So now you can point your browser to

```
file:///usr/local/java/docs/api/index.html
```

Now you have your own local copy for quick reference, regardless of how busy the network or Sun's Web site gets. Be sure to bookmark this page; you'll want to reference it often. It's your best source of information about all the standard Java2 classes.

3.2.3 Statements

This section is not intended to be a formal presentation of Java syntactic elements.[3] Our purpose here is merely to show you the Java way to express common programming constructs. You will find that these are fundamentally similar to the analogous statements in C and C++. For much more detail on these subjects, see Chapter 3 of *Thinking in Java* by Bruce Eckel.

Like C, Java has a very small set of statements. Most constructs are actually *expressions*. Most operations are either assignments or method calls. Those few statements that are not expressions fall into two broad categories:

- Conditional execution statements
- Loop control statements

By the way, you may have already noticed one of the two kinds of *comments* that Java supports. They are like the C/C++ comments—a pair of slashes (`//`) marks a comment from there to the end of the line, and a block

3. For those so inclined, Sun has a BNF language grammar (`http://java.sun.com/docs/books/jls/second_edition/html/syntax.doc.html`) on their Web site, and the Lewis and Loftus book, Appendix L, has a good set of syntax diagrams.

comment consists of everything from the opening `/*` to the closing `*/` sequence.

3.2.3.1 Conditional Execution

An experienced programmer probably only needs to see examples of `if` and other such statements to learn them. It's only a matter of syntax. Java breaks no new ground here; it adds no new semantics to conditional execution constructs.

The `if-else` statement. The `if` can take a single statement without any braces, but we always use the braces as a matter of good style (Example 3.7).

Example 3.7 A compound Java `if-else` statement

```
if (x < 0) {
    y = z + progo;
} else if (x > 5) {
    y = z + hmron;
    mylon.grebzob();
} else {
    y = z + engrom;
    mylon.kuggle();
}
```

> **TIP**
>
> An important thing to remember about the Java `if` statement (and all other conditional tests, such as `while`, `do-while`, and `for`) is that, unlike C/C++, its expression needs to evaluate to a `boolean`. In C/C++, numeric expressions are valid, any nonzero value being considered `true`, but not so in Java.

The `switch` statement. For a multiway branch Java, like C/C++, has a `switch` statement, though the Java version of `switch` is a bit more restrictive. Example 3.8 shows the syntax.

In Java, the expression in the `switch` statement must evaluate to either an `int` or a `char`. Even `short` and `long` are not allowed.

As in C/C++, be sure to put the `break` statement at the end of each `case`, or else control will flow right into the next `case`. Sometimes this is the desired behavior—but if you ever do that deliberately, be sure to add a comment.

Example 3.8 A `switch` statement in Java

```
switch (rval*k+zval)
{
  case 0:
    mylon.reset();
    break;
  case 1:
  case 4:
    // matches either 1 or 4
    y = zval+engrom;
    mylon.kuggle(y);
    break;
  default:
    // all other values end up here
    System.out.println("Unexpected value.");
    break;
}
```

The `default` case is where control goes when no other `case` matches the expression. It is optional—you don't need to have one among your switch cases. Its location is also arbitrary; it could come first, but by convention programmers put it last in the sequence of cases, as a visual "catch all."

> **TIP**
>
> For whichever `case` is last (typically `default`), the ending `break` is redundant because control will continue outside the `break`—but we show it here in the example, and use it ourselves in our code. Why? Well, code gets edited—for bug fixes and for feature additions. It is especially important to use `break` in all the `case`s in `switch` statements that have no `default` case, but even in those that do, we keep the `break` to avoid forgetting it, should another `case` ever be added or this last one relocated. We recommend that you do the same.

3.2.3.2 Looping and Related Statements

The `while` statement. Like the `while` construct in other computer languages, the expression inside the parentheses is evaluated, and if `true`, the statement following it is executed. Then the expression is evaluated again, and if still `true`, the looped statement is again executed. This continues until the expression evaluates to `false` (Example 3.9).

Example 3.9 A Java `while` statement

```
while (greble != null)
{
  greble.glib();
  greble = treempl.morph();
}
```

Technically, the `while` statement consists of the expression and a single statement, but that single statement can be replaced by a set of statements enclosed in braces (you know, the characters `{` and `}`). We will always use braces, even if there is only one statement in our `while` loop. Experience has shown that it's a safer practice that leads to code that is easier to maintain. Just treat it as if the braces were required syntax, and you'll never forget to add them when you add a second statement to a loop.

The `do-while` loop. To put the terminating check at the bottom of the loop, use `do-while` as shown in Example 3.10. Notice the need for the terminating semicolon after the expression.

Example 3.10 A Java `do-while` statement

```
do {

  greble.morph();
  xrof = treempl.glib();

} while (xrof == null);
```

Die-hard Pascal programmers should note that Java has no `repeat-until` statement. Sorry. Of course the logic of an `until(condition)` is equivalent to `do-while(!condition)`.

The `for` loop. The `for` loop in Java is very similar to C/C++. It consists of three parts (Example 3.11):

- The initializing expression, done up front before the loop begins
- The conditional expression for terminating the loop

- The expression that gets executed at the end of each loop iteration, just prior to retesting the conditional

Example 3.11 A Java `for` loop

```
for (i = 0; i < 8; i++) {
  System.out.println(i);
}
```

Unlike C/C++, Java doesn't have the comma operator for use within arbitrary expressions, but the comma is supported as special syntax in Java `for` loops. It makes it possible to have multiple initializers in the opening of the `for` loop and multiple expressions in the portion repeated at each iteration of the loop. The result is much the same—you can initialize and increment multiple variables or objects in your `for` loop.

More formally, the full syntax of the `for` loop can be described with following meta-language as shown in Example 3.12 (where the `[]*` means "zero or more repetitions of").

Example 3.12 Java `for` loop syntax

```
for ( before [, before]* ; exit_condition ; each_time [, each_time]* )
  statement
```

The biggest difference between C and Java `for` loops, however, is that Java allows you to declare one or more variables of a single type in the initializing expression of the `for` loop (Example 3.13). Such a variable's scope is the `for` loop itself, so don't declare a variable there if you want to reference it outside the loop. It is a very handy construct, however, for enumerators, iterators, and simple counters.

Example 3.13 A Java `for` loop with local index

```
for (int i = 0; i < 8; i++) {
  System.out.println(i);
}
```

As in the `if` and `while` statements, the braces are optional when only a single statement is involved, but good practice compels us always to use the braces. Additional code can easily be added without messing up the logic—should one forget, at that point, the need to add braces.

Speaking of the `while` loop: When do you use a `for` and when do you use a `while` loop? The big advantage of the `for` loop is its readability. It consolidates the loop control logic into a single place—within the parentheses. Anyone reading your code can see at once what variable(s) are being used to control how many times the loop executes and what needs to be done on each iteration (e.g., just increment `i`). If no initialization is needed before starting the loop, or if the increment happens indirectly as part of what goes on in the body of the loop, then you might as well use a `while` loop. But when the initialization and iteration parts can be clearly spelled out, use the `for` loop for the sake of the next programmer who might be reading your code.

The `for` loop with iterators. As of Java 5.0, there is additional syntax for a `for` loop. It is meant to provide a useful shorthand when looping over the members of an iterator.[4] So what's an iterator? Well, it has to do with collections. Uh, oh, we're surrounded by undefined terms. One step at a time, here. Java has a whole bunch (we won't say "collection," it's a loaded term) of utility classes that come with it. We mentioned these classes in our discussion of Javadoc. While not part of the language syntax, some of these classes are so useful that you will see them throughout many, if not most, Java programs.

Collection is a generic term (in fact, it's a Java interface) for several classes that allow you to group similar objects together. It covers such classes as `Lists`, `LinkedLists`, `Hashtables`, `Sets`, and the like. They are implementations of all those things that you (should have) learned in a Data Structures course in school. Typically you want to add (and sometimes remove) members from a collection, and you may also want to look something up in the collection. (If you're new to collections, think "array," as they are a simple and familiar type of collection.) Sometimes, though, you don't want just one item from the collection, but you want to look at all of the objects in the collection, one at a time. The generic way to do that, the way that hides the specifics of what kind of collection you have (linked list, or array, or map) is called an iterator.[5]

---

4. This feature is related to the topic of templates and generics. See Section 3.5.

5. The earliest versions of Java used an object called an `Enumeration`. It does much the same thing as an iterator, but with somewhat clumsier method names. Iterators also allow for a

The purpose of an iterator, then, is to step through a collection one item at a time. Example 3.14 shows a collection being built from the arguments on the command line. Then two iterators are used to step through the collection

Example 3.14 Using iterators

```
import java.util.*;

public class
Iter8
{
  public static void
  main(String [] args)
  {
    // create a new (empty) ArrayList
    ArrayList al = new ArrayList();

    // fill the ArrayList with args
    for(int i = 0; i < args.length; i++) {
      al.add(args[i]);
    }

    // use the iterator in the while loop
    Iterator itr1 = al.iterator();

    while(itr1.hasNext()) {
      String onearg;
      onearg = (String) (itr1.next());
      System.out.println("arg=" + onearg);
    }

    // define and use the iterator in the for loop:
    for(Iterator itr2 = al.iterator(); itr2.hasNext(); ) {
      String onearg;
      onearg = (String) (itr2.next());
      System.out.println("arg=" + onearg);
    }

  } // main

} // Iter8
```

`remove()` method, something that `Enumeration` doesn't support. The `Enumeration` class is still around, but less frequently used. It is only available from certain older utility classes.

and print the objects in the collection to the command window. The first iterator uses the `while` loop, the second one uses a `for` loop, but they both do the same thing.

As of Java 5.0, there is another way to work your way through a collection, one that requires less type casting, but more importantly one that can enforce the type of objects at compile time.

Notice in Example 3.14 that the result of the `next()` is coerced into type `String`. That's because everything coming from the iterator (via the `next()` method) comes to us as a generic object. That way an iterator can handle any type of object, but that also means that it is up to the application program to know what type should be coming back from the iterator. Any typecasting error won't be found until runtime.

With the syntax added in 5.0, not only is there a shorthand in the `for` loop for looping with an iterator. There is also syntax to tell the compiler explicitly what type of objects you are putting into your collection or array so that the compiler can enforce that type.

Example 3.15 may help to make this clearer.

Example 3.15 Using a `for` loop iterator

```
import java.util.*;

public class
Foreign
{
  public static void
  main(String [] args)
  {
    List <String> loa = Arrays.asList(args);

    System.out.println("size=" + loa.size());

    for(String str : loa) {
      System.out.println("arg=" + str);
    }

  } // main
} // Foreign
```

Here we build a `List` from the arguments supplied on the command line. Notice the type name inside of angle brackets (less-than and greater-than signs).

This is the new syntax that tells the compiler that we are putting `String`s into the `List`. The compiler will enforce that and give a compile time error if we try to add any other type to the `List`.

Now we come to the `for` loop. Read it as "for `str` in `loa`" or "for `String` values of `str` iterating over `loa`." We will get an iterator working out of sight that will iterate over the values of `loa`, our `List`. The values (the result of the `next()` method) will be put in the `String` variable `str`. So we can use `str` inside the body of the loop, with it taking on successive values from the collection.

Let's describe the syntax, then, as

```
for ( SomeType variable : SomeCollectionVariable ) {
}
```

which will define `variable` to be of type `SomeType` and then iterate over the `SomeCollectionVariable`. Each iteration will execute the body of the loop, with the variable set to the `next()` value from the iterator. If the collection is empty, the body of the loop will not be executed.

This variation of the `for` loop works for arrays as well as for these new typed collections. The syntax for arrays is the same. Example 3.16 will echo the arguments on the command line, but without loading up a `List` like we did in our previous example.

Example 3.16 A `for` loop iterator for arrays

```
import java.util.*;

public class
Forn
{
  public static void
  main(String [] args)
  {
    for(String str : args) {
      System.out.println("arg="+str);
    }

  } // main
} // Forn
```

The `break` and `continue` statements. There are two statements that will change the course of execution of the `while`, `do-while`, and `for` loops from within the loop. A `continue` will cause execution to skip the rest of the body of the loop and go on to the next iteration. With a `for` loop, this means executing the iteration expression, and then executing the test-for-termination expression. With the `while` and `do-while` loops, this means just going to the test expression.

You can quit out of the loop entirely with the `break` statement. Execution continues on the next statement after the loop.

3.2.3.3 *The `return` statement*

There is one more statement that we need to cover. The `return` statement is optionally followed by an expression. Execution of the current method ends at once upon executing `return`, and the expression is used as the return value of the method. Obviously, the type of the expression must match the return type of the method. If the method is void, there should be no return expression.

3.2.4 Error Handling, Java Style

Errors in Java are handled through *exceptions*. In some circumstances, the Java runtime will *throw* an exception, for example, when you reference a null pointer. Methods you write may also throw exceptions. This is quite similar to C++. But Java exceptions are classes. They descend from `Object`, and you can write your own classes that *extend* an existing exception. By so doing, you can carry up to the handler any information you would like. But we're getting ahead of ourselves here. Let's first describe the basics of exceptions, how to catch them, how to pass them along, and so forth.

In other programming languages a lot of code can be spent checking return codes of function or subroutine calls. If A calls B and B calls C and C calls D, then at each step the return value of the called function should be checked to see if the call succeeded. If not, something should be done about the error—though that "something" is usually just returning the error code to the next level up. So function C checks D's return value, and if in error, returns an error code for B to check. B in turn looks for an error returned from C and returns an error code to A. In a sense, the error checking in B and C is superfluous. Its only purpose is to pass the error from its origin in D to the function that has some logic to deal with the error—in our example that's A.

Java provides the `try/catch/throw` mechanism for more sophisticated error handling. It avoids a lot of unnecessary checking and passing on of errors. The only parts of a Java program that need to deal with an error are those that know what to do with it.

The `throw` in Java is really just a nonlocal "goto"—it will branch the execution of your program to a location which can be quite far away from the method where the exception was thrown. But it does so in a very structured and well-defined manner.

In our simple example of A calling B calling C calling D, D implemented as a Java method can *throw* an exception when it runs into an error. Control will pass to the first enclosing block of code on the call stack that contains a `catch` for that kind of exception. So A can have code that will *catch* an exception, and B and C need not have any error handling code at all. Example 3.17 demonstrates the syntax.

Example 3.17 A simple `try/catch` block

```
try {
  for (i = 0; i < max; i++) {
    someobj.methodB(param1, i);
  }
} catch (Exception e) {
  // do the error handling here:
  System.out.println("Error encountered. Try again.");
}
// continues execution here after successful completion
// but also after the catch if an error occurs
```

In the example, if any of the calls to `methodB()` in the `for` loop go awry—that is, anywhere inside `methodB()` or whatever methods it may call an exception is thrown (and assuming those called methods don't have their own `try/catch` blocks), then control is passed up to the `catch` clause in our example. The `for` loop is exited unfinished, and execution continues first with the `catch` clause and then with the statements after the `catch`.

How does an error get thrown in the first place? One simply creates an `Exception` object and then throws the exception (Example 3.18).

Example 3.18 Throwing an `Exception`, step by step

```
Exception ex = new Exception("Bad News");
throw ex;
```

Since there is little point in keeping the reference to the object for the local method—execution is about to leave the local method—there is no need to declare a local variable to hold the exception. Instead, we can create the exception and throw it all in one step (Example 3.19).

Example 3.19 Throwing an `Exception`, one step

```
throw new Exception("Bad News");
```

`Exception` is an object, and as such it can be extended. So we can create our own unique kinds of exceptions to differentiate all sorts of error conditions. Moreover, as objects, exceptions can contain any data that we might want to pass back to the calling methods to provide better diagnosis and recovery.

The `try/catch` block can catch different kinds of exceptions much like cases in a `switch/case` statement, though with different syntax (Example 3.20).

Notice that each `catch` has to declare the type of each exception and provide a local variable to hold a reference to that exception. Then method calls can be made on that exception or references to any of its publicly available data can be made.

Remember how we created an exception (`new Exception("message")`)? That message can be retrieved from the exception with the `toString()` method, as shown in that example. The method `printStackTrace()` is also available to print out the sequence of method calls that led up to the creation of the exception (Example 3.21).

The exception's stack trace is read top to bottom showing the most recently called module first. Our example shows that the exception occurred (i.e., was constructed) on line 6 of the class named `InnerMost`, inside a method named `doOtherStuff()`. The `doOtherStuff()` method was called from inside the class `MidModule`—on line 7—in a method named `doStuff()`. In turn, `doStuff()` had been called by `doSomething()`, at line 11 inside

Example 3.20 Catching different kinds of exceptions

```
try {
  for (i = 0; i < max; i++) {
    someobj.methodB(param1, i);
  } // next i

} catch (SpecialException sp) {
    System.out.println(sp.whatWentWrong());

} catch (AlternateException alt) {
    alt.attemptRepair(param1);

} catch (Exception e) {
    // do the error handling here:
    System.out.println(e.toString());
    e.printStackTrace();
}
// continues execution here after any catch
```

Example 3.21 Output from `printStackTrace()`

```
java.lang.Exception: Error in the fraberstam.
    at InnerMost.doOtherStuff(InnerMost.java:6)
    at MidModule.doStuff(MidModule.java:7)
    at AnotherClass.doSomething(AnotherClass.java:11)
    at ExceptExample.main(ExceptExample.java:14)
```

`AnotherClass`, which itself had been called from line 14 in the `ExceptExample` class' `main()` method.

We want to mention one more piece of syntax for the `try/catch` block. Since execution may never get to all of the statements in a `try` block (the exception may make it jump out to a `catch` block), there is a need, sometimes, for some statements to be executed regardless of whether all the `try` code completed successfully. (One example might be the need to close an I/O connection.) For this we can add a `finally` clause after the last `catch` block. The code in the `finally` block will be executed (only once) after the `try` or after the `catch`—even if the path of execution is about to leave because of throwing an exception (Example 3.22).

Example 3.22 Use of a `finally` clause

```
try {
  for (i = 0; i < max; i++) {
    someobj.methodB(param1, i);
  } // next i

} catch (SpecialException sp) {
    System.out.println(sp.whatWentWrong());

} catch (AlternateException alt) {
    alt.attemptRepair(param1);
    throw alt;    // pass it on

} catch (Exception e) {
    // do the error handling here:
    System.out.println(e.toString());
    e.printStackTrace();

} finally {
    // Continue execution here after any catch
    // or after a try with no exceptions.
    // It will even execute after the AlternateException
    // before the throw takes execution away from here.
    gone = true;
    someobj = null;
}
```

3.2.5 `print()`,`println()`,`printf()`

We've already used `println()` in several examples, and assumed that you can figure out what it's doing from the way we have used it. Without going whole-hog into an explanation of Java I/O and its various classes, we'd like to say a little more about the three various output methods on a `PrintStream` object.[6]

Two of the methods, `print()` and `println()`, are almost identical. They differ only in that the latter one appends a newline (hence the `ln`) at the end of its output, thereby also flushing the output. They expect a `String` as their only argument, so when you want to output more than one thing, you add the `String`s together, as in:

6. The mention of the `PrintStream` object was meant to be a hint, to tell you that you can find out more about this sort of thing on the Javadoc pages for the `PrintStream` object.

```
System.out.println("The answer is "+val);
```

"But what if `val` is not a `String`?" we hear you asking. Don't worry, the Java compiler is smart enough to know, that when you are adding with a `String` argument it must convert the other argument to a `String`, too. So for any `Object`, it will implicitly call its `toString()` method. For any primitive type (e.g., `int` or `boolean`), the compiler will convert it to a `String`, too.

The third of the three output methods, `printf()`, sounds very familiar to C/C++ programmers, but be warned:

- It is only available in Java 5.0[7] and after.
- It is similar but not identical to the C/C++ version.

Perhaps the most significant enhancement to `printf()` is its additional syntax for dealing with internationalization. It's all well and good to translate your `String`s to a foreign language, but in doing so you may need to change the word order and thus the order of the arguments to `printf()`. For example, the French tend to put the adjective *after* rather than *before* the noun (as we do in English). We say *"the red balloon"* and they say *"le balloon rouge."* If your program had `String`s for adjective and noun, then a `printf()` like this:

```
String format = "the %s %s\n";
System.out.printf(format, adjective, noun);
```

wouldn't work if you translate just the format `String`:

```
String format = "le %s %s\n";
System.out.printf(format, noun, adjective);
```

You'd like to be able to do the translation without changing the code in your program.[8] With the Java version of `printf()`, there is syntax for specifying which argument corresponds to which format field in the format string. It uses

---

7. Remember, you'll need the `-source 5.0` option on the command line.

8. Java has good support for internationalization, another topic for which we don't have the time. The ability to translate the strings without otherwise modifying the program is a crucial part to internationalization, and the `printf()` in Java 5.0 is certainly a help in this regard. In a similar vein, the Eclipse IDE, covered in Chapter 10, includes a feature to take all string constants and convert them to external properties at a stroke, making internationalization much easier to do.

a number followed by a dollar sign as part of the format field. This may be easier to explain by example; our French translation, switching the order in which the arguments are used, would be as follows:

```
String format = "le %2$s %1$s\n";
System.out.printf(format, noun, adjective);
```

The format field `%2$s` says to use the second argument from the argument list—in this case, adjective—as the string that gets formatted here. Similarly, the format field `%1$s` says to use the first argument. In effect, the arguments get reversed without having to change the call to `println()`, only by translating the format `String`. Since such translations are often done in external files, rather than by assignment statements like we did for our example, it means that such external files can be translated without modifying the source to move arguments around.

This kind of argument specification can also be used to repeat an argument multiple times in a format string. This can be useful in formatting `Date` objects, where you use the same argument for each of the different pieces that make up a date—day, month, and so on. Each has its own format, but they can be combined by repeating the same argument for each piece. One format field formats the month, the next format field formats the day, and so on. Again an example may make it easier to see:

```
import java.util.Date;
Date today = new Date();
System.out.printf("%1$tm / %1$td / %1$ty\n", today);
```

The previous statement uses the single argument, `today`, and formats it in three different ways, first giving the month, then the day of the month, then the year. The `t` format indicates a date/time format. There are several suffixes for it that specify parts of a date, a few of which are used in the example.[9]

> **NOTE**
>
> Don't forget the trailing `\n` at the end of the format string, if you want the output to be a line by itself.

9. There are many more, familiar to C/C++ UNIX/Linux/POSIX programmers who have used the `strftime()` library call.

The details for all the different format fields can be found in the Javadoc for the `java.util.Formatter` class, a class that is used by `printf()` to do its formatting, but one that you can also use by itself (C programmers: think "sprintf").

In order to implement `printf()` for Java, the language also had to be extended to allow for method calls with a varying number of arguments. So as of Java 5.0, a method's argument list can be declared like this:

```
methodName(Type ... arglist)
```

This results in a method declaration which takes as its argument an array named `arglist` of values of type `Type`. That is, it is much the same as if you declared `methodName(Type [] arglist)` except that now the compiler will let you call the method with a varying number of arguments and it will load up the arguments into the array before calling the method. One other implication of this is that if you have a declaration like this:

```
varOut(String ... slist)
```

then you can't, in the same class, also have one like this:

```
varOut(String [] alist)
```

because the former is just a compiler alias for the latter.

> **TIP**
>
> We recommend that you avoid methods with variable argument list length. You lose the compile-time checking on the number of arguments that you supply (since it can vary). Often the type of the arguments in the list will be `Object`, the most general type, to allow anything to be passed in. This, too, circumvents type checking of arguments, and can lead to runtime class-cast exceptions and other problems. Methods with variable argument list length are often a lazy approach, but were necessary to make `printf()` work, and for that we are grateful.

3.3 USING (AND MAKING) JAVA APIS

With every class you write, you define a name—the name of the class. But what if someone else has already used that name? Java programming should encourage reuse of existing code, so how do you keep straight which names are available?

This is a *namespace* issue—who can use which names. A classic way to solve a namespace issue is to divide the namespace up into domains. On the Internet, host names are sectioned off into domains, so that I can have a host named `Pluto` or `www` and so can lots of others—because each host is *qualified* by its domain (e.g., `myco.com`). Thus `www.myco.com` isn't confused with `www.otherco.com` or `www.hisgroup.org`. Each host is named `www`, but each is unique because of the qualifying domain.

Java solves the problem in much the same way, but with the names in the reverse order. Think of the "host" as the class name; the "domain" name, used to sort out identical host names, is, in Java parlance, the *package name*. When you see a name like `com.myco.finapp.Account`, that can be a Java package `com.myco.finapp` qualifying a class named `Account`.

Beyond just keeping the namespace clean, Java packages serve another important function. They let you group together similar classes and interfaces to control access to them. Classes within the same package can access each others' members and methods even when they are not declared `public`, provided they are not declared to be `private`. This level of intimacy, sometimes called *package protection*, means that you should group classes together that are related, but avoid grouping too many classes together. It's tempting just to put all your classes for a project into the same package, for example, `com.myco.ourproject`, but you will provide better safety and perhaps promote better reuse by grouping them into several smaller packages, for example, `com.myco.util`, `com.myco.financial`, and `com.myco.gui`.

3.3.1 The `package` Statement

So how do you make a Java class part of a package? It's easy—you just put, as the first (noncomment) line of the file, the `package` statement, naming the package to which you want this class to belong. So if you want your `Account` class to be part of the `com.myco.financial` package, your Java code would look as shown in Example 3.23.

Example 3.23 Use of a `package` statement

```
package com.myco.financial;

public class
Account
{
  // ...
}
```

Making a class part of a package is easy. What's tricky is putting the class file in the right location so that Java can find it.

Think of the current directory as the root of your package tree. Each part of a package name represents a directory from that point on down. So if you have a package named `com.myco.financial` then you'll need a directory named `com` and within that a directory named `myco` and within that a directory named `financial`. Inside that `financial` directory you can put your `Account.class` file.

When Java runs a class file, it will look in all the directories named in the `CLASSPATH` environment variable. Check its current value:

```
$ echo $CLASSPATH

$
```

If it's empty, as in this example, then the only place where it will look for your classes will be the current directory. That's a handy default, because it is just what you want when your class file has no `package` statement in it. With no `package` statement, your class becomes part of the *unnamed package*. That's fine for simple sample programs, but for serious application development you'll want to use packages.

Let's assume that you're in your home directory, `/home/joeuser`, and beneath that you have a `com` directory and beneath that a `myco` directory with two subdirectories `financial` and `util`. Then with your classes in those lower level directories, you can run your Java program from the home directory. If you want to run it from any arbitrary directory (e.g., `/tmp` or `/home/joeuser/alt`) then you need to set `CLASSPATH` so it can find this package tree. Try:

```
$ export CLASSPATH="/home/joeuser"
$
```

Now Java knows where to look to find classes of the `com.myco.financial` and `com.myco.util` packages.

3.3.2 The `import` Statement

Once we have put our classes into packages, we have to use that package's name when we refer to those classes—unless we use the `import` statement.

Continuing our example, if we want to declare a reference to an `Account` object, but `Account` is now part of `com.myco.financial`, then we could refer to it with its full name, as in:

```
com.myco.financial.Account =
    new com.myco.financial.Account(user, number);
```

which admittedly is a lot more cumbersome than just:

```
Account = new Account(user, number);
```

To avoid the unnecessarily long names, Java has `import` statements. They are put at the beginning of the class file, outside the class definition, just after any `package` statement. In an `import` statement, you can name a class with its full name, to avoid having to use the full name all the time. So our example becomes:

```
import com.myco.financial.Account;
// ...
Account = new Account(user, number);
```

If you have several classes from that package that you want to reference, you can name them all with a "`*`", and you can have multiple different `import` statements, as in:

```
import java.util.*;
import com.myco.financial.*;
// ...
Account = new Account(user, number);
```

Here are a few things to remember about `import` statements. First, they don't bring in any new code into the class. While their syntax and placement

is reminiscent of the C/C++ `include` preprocessor directive, their function is not the same. An `import` statement does not include any new code; it only aids name resolution. Secondly, the "`*`" can only be used at the end of the package name; it is not a true wildcard in the regular expression sense of the word. Thirdly, every class has what is in effect an implicit `import java.lang.*` so that you don't need to put one there. References to `String` or `System` or other core language classes can be made without the need for either the `import` statement or the fully qualified name (except as described in the next paragraph).

If you need to use two different classes that have the same name but come from different packages, you will still need to refer to them by their full names; `import` can't help you here. As an example of this, consider the two classes `java.util.Date` and `java.sql.Date` (though with any luck you won't need to refer to both of them within the same class).

3.4 Encapsulation, Inheritance, and Polymorphism

The classic troika of OOP buzzwords is "encapsulation, inheritance, and polymorphism." How does Java do each of these things?

3.4.1 Encapsulation

Encapsulation is the grouping of data and algorithms together into units, and it's also about hiding implementation details that are not important to the users of the unit. The basic unit of encapsulation is the class. All Java code exists in classes. A class is declared with the `class` keyword (Example 3.24).

Example 3.24 A sample Java class declaration that doesn't actually do anything useful

```
public class
Sample
{
  private int id;

  public void method()
  {
    System.out.println(id);
  }
}
```

3.4.2 Inheritance

Inheritance is how a class places itself in a hierarchy of existing classes. In Java, each class inherits from exactly one existing class. A class names the class from which it inherits with the `extends` keyword. We said a Java class inherits from exactly one class, and yet our Example 3.24 doesn't contain the `extends` keyword. What gives?

If a class doesn't explicitly extend an existing class, it implicitly extends the "root" Java class, `Object`. The `Object` class has some interesting features, and the fact that all Java classes directly or indirectly extend `Object` has interesting consequences that we will explore later.

Persons coming to Java from another object-oriented language whose name shall remain C++ might wonder about multiple inheritance. Java has the concept of *interfaces*. An interface is like a class, except that it may not contain data[10] and may contain only method definitions, without any implementation (Example 3.25). An interface may not be *instantiated* (created with the `new` operator),[11] so how do you make use of interfaces? Well, a class extends exactly one existing base class, but it may *implement* any number of interfaces by using the `implements` keyword (Example 3.26).

Example 3.25 An interface

```
public interface
Identifiable
{
  public int getID();
}
```

As you can see, a class that implements an interface must provide an implementation of all the methods defined in the interface. We said that an interface cannot be instantiated, but you can declare a variable of type `Identifiable`

10. Actually, an interface may contain `final static` data, but since we haven't introduced these concepts yet, just pretend interfaces cannot contain data for now and we'll put the lie to it later.

11. Although you can do something that looks suspiciously like it with *anonymous inner classes*—but since we haven't introduced these concepts yet, just pretend that you cannot instantiate an interface; you will see such use later.

Example 3.26 A class that implements an interface

```
class
Sample
  implements Identifiable
{
  private int id;

  public void method()
  {
    System.out.println(id);
  }

  public int getID()
  {
    return id;
  }
}
```

and assign an instance of the `Sample` class to it. In fact, you could assign an instance of any class that implements the `Identifiable` interface to it.

Interfaces may also have an `extends` keyword. In other words, an interface may inherit from an existing interface. This may be useful if you know you will want to use methods from both the extended and the base interface without having to cast the object reference. Otherwise extending an interface is unnecessary since a given class may implement as many interfaces as desired.

3.4.2.1 Inheritance and Encapsulation

Encapsulation and inheritance are related to one another and are controlled by *access modifiers* on classes, data members, and methods. Let's spend a little time talking about these modifiers and what they mean.

The access modifier keywords are `public`, `private`, and `protected`. When a data member or method is `private`, it can only be accessed or called from within this specific class. Neither classes that extend this class, nor classes outside this class may access such a data member or call such a method. However, one instance of a class can access `private` members of another instance of the same class. We don't encourage such use.

When a data member or method is marked `protected`, however, the only classes that can access such a data member or method are either 1) classes that extend this class and their descendants, or 2) other classes in this package (even

if they don't extend this class). Classes in other packages (unless they extend this class) can not get at such members.

A `public` data member or method may be accessed by any code in any class.

What if you do not put an access specifier on? Then the item (data member, method, or class) has package visibility. Such an item is accessible to any other class within the package, but no further. Not even derived classes, unless they are in the same package, are allowed to see it.[12]

In terms of how restrictive the access is, you can think of the terms in order of decreasing strictness as:

- `private`
- (package)[13]
- `protected`
- `public`

> **TIP**
>
> Beginner Java programmers often declare everything as `public`, so that they can ignore such issues. But then they get the OO religion, and having experienced reliability issues (others messing with their variables) they go to the other extreme and declare `private` as much as possible. The problem here is that they often don't know how others will want to reuse their code. Restricting everything to `private` makes reuse more narrow. We prefer using `private` for data members but `protected` for those internal helper methods that you might otherwise make `private`; this hides your implementation from most other classes while allowing someone to override your methods, effectively providing a way for them to override your implementation. Allow those who would build on your work the ability to do so without having to reimplement.

Here is a simple example of each type of declaration:

12. If you are a C++ programmer, the following description may mean something to you (if not, skip this): All classes within a package are essentially "friends."

13. Remember there is no keyword for package level protection, rather it is the absence of a keyword that denotes this level of protection. We had to write something in that space on the page so you'd know what we're talking about.

```
private String hidn;
String pkgstr;
protected String protstr;
public String wideOpen;
```

3.4.2.2 The `static` statement

Another keyword qualifier on declarations that we need to describe is the `static` keyword. When a variable is declared `static` then there is only one instance of that variable shared among all instances of the class. Since the variable exists apart from a particular instance of the class, one refers to it with the class name followed by a dot followed by the variable name, as in `System.out`.

Similarly, methods can be declared `static` as well. This also means that you don't need an instance of the class to call them, just the class name, as in `System.getProperties()`.

Now with Java 5.0, you don't even need the class name, provided that you have a `static import` statement at the top of your class, for example:

```
import static java.lang.System.*;
```

3.4.2.3 The `final` statement

Another way that `static` is often seen is in conjunction with the `final` keyword. When a variable is declared `final` then a value can be assigned to it once, but never changed. This can make for good constants.

Since `public` will make the variable visible to all other classes, `static` will make it a class variable (available without an instance of that class), and `final` will keep it from being altered (even though it is publicly available), then combining all of those gives us a good declaration for a constant, for example:

```
public static void long lightSpeed = 186000;    // mps
```

New to Java 5.0 is the explicit creation of enumerated types. Prior to 5.0, programmers would often use `static final` constants even when the particular value was unimportant, as a way to provide compile-time checking of the use of the constant values. Here is an example of a declaration of a set of enumerated values:

```
enum WallMods { DOOR, WINDOW, VENT, GLASSBLOCK };
```

TIP

A common technique used with `public static final` constants is to put them in an interface definition. (This is the exception to the rule that interfaces define method signatures but contain no data.) When a class wants to use one or more of those constants, it is declared to implement that interface:

```
public MyClass
  extends BigClass
  implements Comparable, LotsaConstants
{
...
}
```

In defining `MyClass` we have declared that it implements `LotsaConstants` (not a name that we recommend you using). That makes all the constants that we have defined inside the `LotsaConstants` interface available to the `MyClass` class. Since classes can implement many different interfaces, this doesn't interfere with the use of other "real" interfaces, such as `Comparable`.

WARNING

The keyword `enum` is new to Java 5.0, so older programs that may have used `enum` as a variable name and will now cause an error when recompiled for Java 5.0.

The `enum` will look very familiar to C/C++ programmers, but there are some important differences. In C/C++ the values of the `enum` elements are, in reality, integer values. Not so in Java. Here they are their own type, but can be converted to a `String` via the `toString()` method, with a value that matches the name, for easy reading and debugging.

Enumerated values can be used in `==` comparisons since they will be defined only once (like other `static final` constants) and it would only be references that are passed around. They would be referenced by the name of the enumeration followed by dot followed by the particular value (e.g., `WallMods.WINDOW`) and used as an object. (We have used uppercase for the names not out of any syntax requirement, but only to follow the typical naming convention for constants.)

3.4.3 Polymorphism

Polymorphism (from the Greek *poly* meaning "many" and *morph* meaning "shape") refers to the language's ability to deal with objects of many different "shapes," that is, classes, as if they were all the same. We have already seen that Java does this via the `extends` and `implements` keywords. You can define an interface and then define two classes that both implement this interface.

Remember our `Sample` class (Example 3.26). We'll now define another class, `Employee`, which also implements the `Identifiable` interface (Example 3.27).

Example 3.27 The `Employee` class

```
class
Employee
  extends Person
  implements Identifiable
{
  private int empl_id;

  public int getID()
  {
    return empl_id;
  }
}
```

Notice that the same method, `getID()`, is implemented in the `Employee` class, but that the field from which it gets the ID value is a different field. That's implementation-specific—the interface defines only the methods that can be called but not their internal implementation. The `Employee` class not only implements `Identifiable`, but it also extends the `Person` class, so we better show you what our example `Person` class looks like (Example 3.28).

To make a really useful `Person` class would take a lot more code than we need for our example. The important part for our example is only that it is quite different from the `Sample` class we saw earlier.

Example 3.29 demonstrates the use of polymorphism. We only show some small relevant snippets of code; there would be a lot more code for this to become an entire, complete example. Don't be distracted by the constructors; we made up some new ones just for this example, that aren't in the class definitions above. Can you see where the polymorphism is at work?

Example 3.28 The `Person` class

```
class
Person
{
  String name;
  Address addr;

  public
  Person(String name, Address addr)
  {
    this.name = name;
    this.addr = addr;

  } // constructor

  // ... lots more code is here

  public String getName()
  {
    return name;
  }
}
```

Example 3.29 An example use of polymorphism

```
//...
Sample labwork = new Sample(petridish);
Employee tech = new Employee(newguy, 27);
Identifiable stuff;

//...
if (mode) {
    stuff = labwork;
} else {
    stuff = tech;
}
id = stuff.getID();
```

The key point here is when the call is made to `getID()`. The compiler can't know at compile time which object will be referred to by `stuff`, so it doesn't know whose `getID()` method will be called. But don't worry—it works

this all out at runtime. That's polymorphism—Java can deal with these different objects while you, the programmer, can describe them in a generalized way.

One other related keyword should be mentioned here, `abstract`. When one declares a class as an abstract class, then the class itself is an incomplete definition. With an abstract class you define all the data of the class but need only write method declarations, not necessarily all the code for the methods. This makes abstract classes similar to interfaces, but in an abstract class, some of the methods can be fully written out.

If you'd like to know more about polymorphism, "late binding," and more of this aspect of Java, read Chapter 7 of Eckel's *Thinking in Java*. There is an extensive example there with much more detail than we can cover here.

3.5 O, TEMPLATES! WHERE ART THOU?

Programmers familiar with C++ may be wondering how in the world an OOP language without templates can be useful.

> **NOTE**
>
> Actually, something very much like templates is available in Java 5.0.[14] A new feature, which Sun calls *generics*, looks an awful lot like C++ templates (including similar syntax). It provides compile-time type checking and implicit casting when retrieving objects from a generic container.

Speaking as programmers who worked with C++ before it had templates, we can sympathize. Java's previous lack of true templates does impose some limits on generic programming, but not as much as one might think. Remember that unlike C++, all Java classes inherit from exactly one base class, and that if no base class is specified, they extend the `Object` class. This means that every single Java class either directly or indirectly extends `Object`, and thus all Java classes are instances of `Object`. So if you need, for example, to implement a container, you can guarantee that it can contain any Java class by implementing a container for the `Object` type. Java also has runtime type identification features that are more than a match for anything C++ has, plus it has type-safe

14. Java 5.0 will only be out by the time this book is completed.

downcasting[15] so that in the worst case scenario, your program has a nice, clean type exception. You simply do not get the kind of "mystery bugs" that you can get in C++ when you miscast an object.[16]

Thanks to interfaces and a true single object hierarchy, many of the uses of C++ templates go away. We doubt very much that you will miss them. In many cases, such as STL algorithms and other functional programming implementations, you can use interfaces to produce similar results.

Critics of the Java language have a point when they complain that all the type casting of class references in order to expose desired interfaces tends to produce code that violates object-oriented principles. The fact that a class or interface implements all these other named interfaces is hard-coded all over the place in an application's code. Such critics say this is a bad thing, because it violates encapsulation and implementation hiding. These critics have a point. If you find yourself frequently downcasting object references, consider using the Java 5.0 generics, or try to find another way to code what you want to do. There may be a better way. In defense of the original Java approach (before generics), all casts are runtime type safe. An exception is thrown if a class reference is improperly cast. In C++, if you miscast a pointer, it assumes you meant it. Java certainly can be awkward, but errors will get caught. Sometimes that is more important.

3.6 VIRTUALLY FINAL

One difficulty anyone writing about Java faces is whether or not to assume your readers are familiar with C++. In this chapter, we have tried to help those with C++ experience without requiring such knowledge. But it is in the inevitable comparisons between those languages that many subtle Java features are best discussed. We promised you that we would talk about the relative merits of `virtual` (a C++ concept) and `final` (a Java concept). To do that, we have to assume some knowledge of C++. So, let's reverse the pattern and talk about the

15. Don't worry if this is all gibberish to you right now. We will revisit these topics in detail when we come upon them in the course of our sample project.

16. Actually, we're being a bit optimistic here. While Java programs are not subject to many mystery bugs, the Java Virtual Machines that run Java code are written in traditional languages, and there have been VMs with bugs. Time and again we see that there is no "silver bullet." But in our experience, Java comes close. So very close.

straight Java facts so we can let the non-C++ folks move on while we go a little deeper with you C++'ers.

In Java, a method or a class may be declared `final`. A method that is declared `final` may not be overridden in classes that extend the class containing the `final` implementation. A class that is declared `final` may not be extended at all.

Now, the comparisons to C++ require us to talk about a language feature that does not exist at all in Java. In C++, unless a method is declared `virtual`, when a class is used by reference to a base class (for example, when using `Employee` as a `Person`), the base class version of the method is called. If the method is declared virtual, the version of the method called is the version for the type of `Person` referenced (in this case, `Employee`). In Java, all methods are virtual. There is no such keyword in Java.

3.7 A Useful Simple Application

We will use the sample application shown in Example 3.30 in other sections of this book to illustrate the use of Java tools. This example is so simple (a single class) that it doesn't demonstrate the object-oriented aspect of development, but it does make use of some APIs that take advantage of it. We will not walk you through this application right here, but present it as a listing of a complete Java class. Not all of the APIs used in this example will be explained, so you may want to refer to the Javadoc pages for explanations of object types or method calls that don't seem obvious.

3.8 Review

We've taken a very quick look at the syntax of Java statements, classes, and interfaces. Much of the syntax is very reminiscent of C, though Java's object-oriented features differ in significant ways from C++. We looked at how to put Java classes into packages, and at the implications of this for locating the `.class` files.

We also showed what the HTML-based Javadoc documentation looks like. These HTML pages will likely be a handy reference for you as you design and write your Java code.

Example 3.30 Single class example: `FetchURL`

```
import java.net.*;
import java.io.*;

public class FetchURL {
  private URL requestedURL;

  public FetchURL(String urlName)
  {
    try {
       requestedURL = new URL(urlName);
    } catch (Exception e) {
       e.printStackTrace();
    }
  }

  public String toString()
  {
    String rc = "";
    String line;
    BufferedReader rdr;

    try {
        rdr = new BufferedReader(
          new InputStreamReader(
          requestedURL.openConnection().getInputStream()
          )
        );

        while ((line = rdr.readLine()) != null)
        {
          rc = rc + line + "\n";
        }
    } catch (Exception e) {
        e.printStackTrace();
        rc = null;
    }

    return rc;
  }
```

```
  public static void main(String[] args)
  {
    int i;
    FetchURL f;

    for (i = 0; i < args.length; i++)
    {
      System.out.println(args[i] + ":");
      System.out.println(new FetchURL(args[i]));
    }
  }
}
```

3.9 WHAT YOU STILL DON'T KNOW

We have deliberately avoided file I/O. For Java, it is a multilayered and complex topic—and with version 1.4 of Java, there is a whole new set of additional classes (`java.nio.*`) to consider. We refer you instead to Chapter 11 of Eckel's *Thinking in Java*.

There are also a few Java keywords that we have not yet discussed, notably `synchronize`.

Even if you know all the Java syntax, it may still take a while to get familiar with the way that syntax is typically put to use. Experience and reading other people's Java code will be your best teachers—but don't assume that a particular approach is good just because someone else uses it; much new code has been written in the last several years as people have learned Java. Be sure it's a style worth imitating, and if you find a better way to do it, use it.

3.10 RESOURCES

- Bruce Eckel, *Thinking in Java*.
- Cay S. Horstmann and Gary Cornell, *Core Java 2: Volume 1 Fundamentals*, especially Chapter 3.
- John Lewis and William Loftus, *Java Software Solutions*.
- The Sun Microsystems Java Tutorial.[17]

17. `http://java.sun.com/docs/books/tutorial/index.html`

3.11 EXERCISES

1. Write a simple class with a `main()` method that prints out the arguments supplied on the command line used to invoke it. First use a `for` loop to do this, then a `while` loop, then a `do-while`. What differences do you notice? Which do you find most amenable for this task?
2. Modify the previous class to quit echoing its arguments should it encounter an argument of length 5. (You can tell the length of a `String` object with the `length()` method, e.g., `mystr.length()`.) Did you use `break`, or `continue`, or some other mechanism?

Chapter 4

Where Am I? Execution Context

Java claims—and rightly so—to be a "compile once, run anywhere" language. But when a program starts, that "anywhere" is now a specific somewhere. When running a Java application on Linux, or any environment for that matter, the question arises, "Where am I?" (context, environment, familiar landmarks). What can a Java program find out about its environment? In particular, on the Linux platform, (how) can we get at:

- Command-line parameters?
- The current shell's environment variables?
- The current working directory?
- The location of data files?

The answers to these questions will depend on what kind of Java application you are creating, and just how portable you want your application to be.

4.1 WHAT YOU WILL LEARN

We'll show you how Java provides access to the command-line parameters and environment variables. We'll also discuss the Java `RunTime` and `Property` classes. Java's use of the standard input/output/error streams is also briefly covered, along with an introduction to those concepts. We'll end with a short word on portability concerns.

4.2 A SIMPLE START

The most basic external information that a program may use is the information supplied on its invocation—simple parameters or arguments, such as filenames or options, that can direct its running and make it a more flexible tool. Let's start with getting at that information from a Java program.

4.2.1 Command-Line Arguments

When a program is run from the command line, more than just the program name can be supplied. Here are some examples:

```
$ javac Hi.java
$ mv Acct.java core/Account.java
$ ls -l
```

In the first example, we invoked a program called **javac** and gave it the parameter `Hi.java`, the name of the file containing the Java program that we want **javac** to compile to Java byte code. (We've got a whole chapter on how to set up and run the Java compiler, see Chapter 5.) The **mv** got two command-line arguments, `Acct.java` and `core/Account.java`, which look a lot like pathnames. The **ls** command has one argument, `-l`, which in Linux usually indicates, by its leading minus sign, that it is an option for altering the behavior of the command. (In this case it produces the "long" version of the directory listing.)

Even point-and-click GUIs allow such parameters to be supplied, though often not visible to the user. In KDE, one can create a new desktop icon that is a link to an application. Such an icon has a property sheet that lists, on the **Execute** tab, the command to be run, including any parameters.

In Java, the parameters supplied on the command line are available to the `main()` method of a Java class. The signature for this method is:

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

From within `main()`, the various parameters are available as the elements of the array of `String`s. The class in Example 4.1 will display those parameters when the program is run.

Example 4.1 Java program to dump command-line arguments

```
/*
 * simple command-line parameter displayer
 */

public class
CLine
{
  public static void
  main(String [] args)
  {
    for (int i = 0; i < args.length; i++)
    {
      System.out.println(args[i]);
    }

  } // main

} // class CLine
```

We compile and run the example, providing a few command-line parameters:

```
$ javac CLine.java
$ java CLine hello world file.txt blue
hello
world
file.txt
blue
$
```

Not all classes will have `main()` methods, but any can. Even if several classes in a package have `main()` methods, that is not a problem. Which one will be the "main" `main()`? It's the class we specified when we invoked our program. In Example 4.1, the `main()` that is executed is the one in the `CLine`

class. Even if `CLine` used other classes (it does—`String` is a class) it doesn't matter if those other classes have `main()` methods or not.

4.2.2 Unit Testing Made Easy

Why all the fuss about `main()` and command-line parameters? Such `main()` methods are a handy way to provide unit tests for a class. The tests can be controlled by the command-line parameters. By testing each class you can reduce the time to integrate the parts of an application. Furthermore, a set of unit tests can be built up (e.g., as shell scripts) to provide automated regression tests for the entire project. As a more rigorous and systematic approach to unit testing, we discuss **junit** in Chapter 13.

4.3 THE `System` CLASS

The Java `System` class provides some of the answers to questions about our environment. What follows is not an exhaustive discussion of all the methods in the `System` class, but only of those areas that touch on our specific focus—input/output (I/O) and environment variables.

Be aware that all of the methods in the `System` class are `static`. Therefore you never need to (and you can't) call a constructor on `System`. You just use the "class name, dot, method name" syntax to call the method (e.g., `System.getProperties()`). Similarly, the accessible fields in `System` are all `static`, so for some of the I/O-related methods you use the "class name, dot, field name, dot, method name" syntax (e.g., `System.out.println()`). As of Java 5.0, you can shorten this, by using a `static import`, that is:

```
import static java.lang.System.*;
```

Then in your other references you can leave off `System`, for example, `getProperties()` and `out.println()`.

4.3.1 Java and Standard I/O

Java adopted the UNIX concept of standard I/O (see Section 1.3.1.1). The Linux file descriptors are available to a running Java program as I/O streams via the `System` class. The `System` class contains three `public static` fields

named `in`, `out`, and `err`. You've probably already seen `out` in Java programs with statements like this:

```
System.out.println("Hello, world.");
```

You can also write:

```
System.err.println("Error message here\n");
```

and

```
BufferedReader in = new BufferedReader(new
                        InputStreamReader(System.in));
while ((line = in.readLine()) != null) {
...
}
```

Java parallels Linux nicely on I/O descriptors. If you redirect any of those file descriptors from the shell command line when you execute a Java program, then that redirected I/O is available to your Java application—with no additional work on your part.

In the example above, if you have `System.in` all wrapped up into a `BufferedReader` from which your program is reading lines, then you can run that program as:

```
$ java MyCode
```

and it will read input as you type it on your keyboard. This may be how you test your program, but when you put this program to its intended use, you may want it to be able to read from a file. This you can do without any change to the program—thanks to file descriptors, input streams, and redirecting input, for example:

```
$ java MyCode < file2
```

which will let the same Java program read from the file named `file2` rather than from keyboard.

Your Java program can also set the values of `System.in`, `System.out`, and `System.err` as it executes, to change their destinations.

One common example is changing the destination of `System.out`, the typical recipient of debugging or logging messages. Say you've created a class

or even a whole package of classes that write log messages to `System.out` (e.g., `System.out.println("some message")`). Now you realize that you'd like the output to go somewhere else.

You could redirect standard `out`, as in:

```
$ java SomeClass > log
```

but that requires the user to remember to redirect the output every time the program is invoked. That's fine for testing, or if the output is intended to go to a different place each time it is invoked. But, in this example, we always want the output to go to the same location.

Without changing any of the `System.out.println()` statements, all the messages can be sent to a new location by reassigning the `System.out` print stream. The `System` class has a *setter* for `out`—that is, a method which will let you set a new value for `out`. In your Java program, open the new destination file and give this to the `System` class:

```
PrintStream ps = new PrintStream("pathname");
System.setOut(ps);
```

It will be used from that point forward in the execution of this program as its `out` output stream.

> **CAUTION**
>
> Changing standard `out` (or `in`, or `err`) will make the change for all classes from here on in this invocation of the Java runtime—they are `static` fields of the one `System` class. Since this is so serious a move, the Java Security Manager (see Section 5.8.4.2) provides a check for `setIO` to see if the Java program is allowed to make such changes. If such a security manager is in place and you are not allowed to make such changes, an exception (`SecurityException`) will be thrown. Note also that the permission applies to setting any of the fields; it doesn't divide the permission into setting one (e.g., `out`) but not another (e.g., `in`).

4.3.2 Environment Variables

When Linux programs are run they have the open file descriptors described above. They also carry with them a list of "name=value" pairs called their *environment*. These environment variables allow for context to be shared among

several successively executed programs. Some examples of environment variables are:

- `USER` is the name you used to log in.
- `HOME` is the directory where you start when you log in.
- `PATH` is the list of directories searched for executable files.

To see the environment variables defined in your current shell, type `env` at the command prompt:

```
$ env
HOME=/home/user01
USER=user01
PATH=/bin:/usr/bin:/usr/local/bin:/home/user01/bin
...
$
```

The names of environment variables, sometimes referred to as *shell variables*, are traditionally uppercase, though that is only a convention. The variable names are treated in a case sensitive fashion (e.g., `Home != HOME`).

You can set environment variables for use in the current shell with a simple assignment statement:

```
$ VAR=value
```

That will set the value for the duration of this shell, but *not* for any of its subprocesses. Since running another program is a subprocess, such an assignment won't be visible in your running program. Instead, you can export the variable so that it is carried forward to all subprocesses:[1]

```
$ export VAR=value
```

4.3.3 Java and Environment Variables

If these environment variables are available to all Linux processes, then how do we get at them from a Java program? Well, we can't do it quite as directly as you might think. In previous (1.2 and older) versions of Java, the `System` class

1. If you are using **csh** (the C-shell, another Linux command-line interpreter), then the syntax is slightly different. Instead of `export name=value` use `setenv name value` (note the different keyword and no equal sign).

Example 4.2 Java program to dump environment variables

```
/*
 * simple environment examiner
 */
import java.util.*;

public class
AllEnv
{
  public static void
  main(String [] args)
  {
    Properties props = java.lang.System.getProperties();
    for (Enumeration enm = props.propertyNames(); enm.hasMoreElements();)
    {
      String key = (String) enm.nextElement();
      System.out.print(key);
      System.out.print(" = ");
      System.out.println(props.getProperty(key));
    }

  } // main

} // class AllEnv
```

had a `getenv()` method. Its argument was a `String` name of an environment variable and it returned the environment variable's value as a `String`. This has been deprecated. In fact, an attempt to use `getenv()` in more recent versions of Java will result in an exception. Sun decided that this was too platform-specific; not all platforms have environment variables.

Now (Java 1.3 and beyond) the preferred approach is to use the `getProperties()` and `getProperty()` methods of the `System` class. How are these different from the `getenv()` approach? To a Linux developer, `getenv()` was easy and straightforward—just not very portable. To accommodate other systems, Java defines a set of properties that are reasonable to expect to be defined on any system, and provides a Java property name for each one.

To see the entire list, call the `getProperties()` method. It returns a `Properties` class, which is an extension of the `Hashtable` class. From this class you can get an `Enumeration` of the names, as Example 4.2 demonstrates.

Now compile and run this example:

```
$ javac AllEnv.java
$ java AllEnv
```

and you will get a long list of properties—in no particular order. They are kept in a hashtable and thus not sorted. Of course it would be easier to use this list if they were sorted. Linux to the rescue.

```
$ java AllEnv | sort
```

It's often in simple little steps like this that one begins to see the power of Linux. In Linux, not every desirable feature has to be crammed into every possible place where it might be used. Instead, features can be written once and connected to one another as needed. Here what we need is to have the list of properties sorted. We don't need to worry that our class didn't sort its output. In Linux we just connect the standard output of the Java program with a sort utility that Linux provides.

So what are all these properties? Many of them have to do with Java-related information (`java.version`, and so on), but a few are more general. Those that parallel the typical Linux environment variables are:

- `file.separator` is the file separator ("`/`" on Linux).
- `path.separator` is the path separator ("`:`" on Linux).
- `line.separator` is the line separator ("`\n`" on Linux).
- `user.name` is the user's account name.
- `user.home` is the user's home directory.
- `user.dir` is the user's current working directory.

But that leaves out so many environment variables, especially the application-specific ones (e.g., `CVSROOT`). How would a Java program get at these?

Because of this new, more portable way to describe the environment, there is no easy way to get at other environment variables. There are a few approaches, but they are all indirect.

First, you can add to the properties list by defining new properties on the command line when invoking the program, for example:

```
$ java -Dkey=value AllEnv
```

You can list several properties on the line by repeating the `-D` parameter:

```
$ java -DHOME=/home/mydir -DALT=other -DETC="so forth" AllEnv
```

Instead of typing those values, you'd probably want to let the Linux shell put in the values from its environment. So you'd use shell variables, for example:

```
$ java -DHOME="${HOME}" -DALT="${ALT}" -DETC="${ETC}" AllEnv
```

assuming that `HOME`, `ALT`, and `ETC` have already been defined in the shell's environment.[2]

If there are only a few variables that you need to pass to Java, put them on the command line as shown above. Put that command line into a shell script and use the script to invoke the program so that the parameters are supplied every time.

But if you want to access many or all of the environment variables then you may want to do something a little more complex. Notice the syntax of the output of the `env` command. It is in the same format (`name=value`) as are properties. So if we use a shell script to invoke our program, we can have it place all these values into a file by redirecting output, then open this file as a Java properties file and thus make all the name/value pairs accessible.

The following commands in a shell script attempt to do just that:

```
env > /tmp/$$.env
java -DENVFILE=/tmp/$$.env MyClass
rm /tmp/$$.env
```

where `MyClass` is the Java program that you wish to run.

> **TIP**
>
> The shell variable `$$` is the numeric process ID of the running process. This provides a unique ID during each invocation of the program. Each run of the script will have its own process and thus its own process ID. Thus a single user could execute this script multiple times concurrently without fear of collision with himself or others.

2. The quotations around the shell variables keep any embedded spaces as part of the variable's value. The curly braces are not strictly necessary in this use.

We remove the temporary file with the **rm** command in the last line of the script to avoid cluttering our `/tmp` directory with lots of these files.

But now we have to add code to `MyClass` to open the file defined by `ENVFILE` and read the properties it contains. This leads us naturally to the Java `Properties` class, the subject of our next section, where we'll talk more about this example.

4.4 THE `Properties` CLASS

The Javadoc page for the `Properties` class describes it as "a persistent set of properties . . . saved to . . . or loaded from . . . a stream." In other words, it is a hashtable (a set of name/value pairs) that can be read from or written to a stream—which typically means a file. (Other things can be streams, but for now, think "file".)

The great thing about name/value pairs is how readable and usable they are. When they are written to a file, there's no fancy formatting, no fixed width fields, no unreadable encryptions and special characters; it's just `name=value`. You could say that the "`=`" and the newline are the special characters that provide all the formatting you need. It means that you can type up a properties file with the simplest of editors, or even generate one quickly as we saw in the previous example (here we use a simple filename):

```
$ env > propertyfile
```

Properties are also easy to use. Since they're based on hashtables, there is no searching code to write. You call a method giving it the name, it returns the value.

If we pass in the name of the file via the `-D` parameter, then we can get that filename in Java with:

```
System.getProperty("ENVFILE");
```

where `ENVFILE` is a name that we made up and used on the command line:

```
$ java -DENVFILE=propertyfile MyClass
```

We could also have used:

```
$ java MyClass propertyfile
```

so that `args[0]`[3] in the Java code to get the name of the file (see Section 4.2.1), but since we want to learn about properties, we'll use the property methods here.

Now let's open that property file (Example 4.3).

Example 4.3 Demonstrating the `Properties` class

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

public class
EnvFileIn
{
  public static void
  main(String [] args)
    throws IOException
  {
    String envfile = System.getProperty("ENVFILE", ".envfile");

    BufferedInputStream bis = new BufferedInputStream(
                                new FileInputStream(envfile));
    Properties prop = new Properties();
    prop.load(bis);
    bis.close();

    prop.list(System.out);  // dumps the whole list to System.out

  } // main

} // class EnvFileIn
```

Notice the way that we got the value for the environment file's name. This form of the `getProperty()` call provides not only the name we are looking up (`ENVFILE`) but also lets us specify a default value in case the name is not found in the properties list. Here our default value is `.envfile`.

Just as it was a simple matter of using the `load()` method to read up an entire file of properties, so you can write out the entire list of properties to the

3. In C language, the `arg[0]` is the command being invoked; not so in Java. In Java, the first element of the array is the first argument of the command line (`propertyfile` in our example).

screen with the `list()` method. The argument to `list()` is either a `PrintStream` or a `PrintWriter`. `System.out` is a `PrintStream`, so that will work.

The format of the properties file is `name=value`. But it is also possible to put comments in a properties file. Any line beginning with a "`#`" is ignored. Try it.

It's also easy to (re)write a file of properties with the `store()` method. The parameters are an `OutputStream` and a `String`; the latter will serve as a label for the parameters, written to an opening comment in the properties file.

If your program needs to examine the list of property names, you can get an `Enumerator` of the entire list via the `propertyNames()` method. Modify Example 4.3 to replace the `list()` call with a do-it-yourself version that uses the `Enumerator` returned from `propertyNames()` to list all the names and values. Hint: Use `getProperty()` on each name retrieved via the enumeration.

The Java `Properties` class extends the `java.util.Hashtable` class. This means, in part, that all the other `Hashtable` methods are available to a `Properties` class. Methods such as `containsKey()` or `containsValue()` can be helpful, as can `isEmpty()`. One caution, though. You should use `setProperty()` if you want to add values to `Properties`, rather than the `Hashtable`'s `put()` method. They do largely the same thing, but `setProperty()` enforces that its parameters are `String`s. This is important if you want to write out the properties to a file, as it's meant for `String`s only.

4.5 THE `Runtime` CLASS

Let's discuss one last way to get to the underlying (Linux) system information. Be warned, though, that this is the least portable approach of all we have mentioned.

4.5.1 `exec()`

Familiar to C/C++ programmers, the `exec()` call in the Java `Runtime` class does much the same thing. It gives you a way to start another program outside of the current Java Virtual Machine. In doing so, you can connect to its standard `in`/`out`/`err` and either drive it by writing to its standard `in`, or read its results from its standard `out`. (Yes, that's correct—we write to its input and read from its output. If that sounds wrong, think it through. Our Java code is

on the opposite side of the I/O fence. The external program's output becomes our input.)

Example 4.4 shows a Java program that can invoke an arbitrary Linux program. The output of the program is displayed.

Example 4.4 Java program to execute any Linux program

```
import java.io.*;

public class
Exec
{
  public static void
  main(String [] args)
    throws IOException
  {
    String ln;
    Process p = Runtime.getRuntime().exec(args);
    BufferedReader br = new BufferedReader(
                          new InputStreamReader(
                            p.getInputStream()));

    while((ln = br.readLine()) != null) {
      System.out.println(ln);
    }
    System.out.println("returns:" + p.exitValue());

  } // main

} // class Exec
```

The command-line arguments are taken to be the command to be executed and its arguments. For example:

```
$ java Exec ls -l
```

Be aware that in this example, only the standard output is captured and displayed from the invoked process. Error messages written to standard `err` will be lost, unless you modify the program to handle this. We leave that as an exercise for the reader.

Check your Linux knowledge—see if you understand the distinction. If you invoke the sample Exec program as:

```
$ java Exec ls -l *.java
```

the shell does the wildcard expansion before invoking the Java runtime. The `*.java` becomes many files listed on the command line (provided that you have `.java` files in this directory). If you try to pass the `*.java` through literally to `exec(ls '*.java')` it will likely return an error (which won't be displayed using our example code) and you'll see a nonzero return status (e.g., 1). That's because **ls** doesn't expand the `*`. The shell does that. So **ls** is looking for a single file named `*.java`, which we hope doesn't exist in your directory.

4.5.2 Portability

Be aware that the more environment-specific code you build, the less portable your application becomes. It's not uncommon to use a properties file as a way to parameterize your program, to customize its behavior in a given installation. But keep these to a minimum to stay portable. Avoid invoking other programs, they are likely not available in all environments where Java can run. Java's claim to "compile once, run anywhere" is amazingly true—provided you keep away from logic in your program that goes looking for trouble.

4.6 REVIEW

Java command-line parameters are not that different from C/C++ command-line parameters. Environment variables are a different story. Most of the shell's environment variables are not readily accessible, and we looked at how you might deal with this situation.

We have discussed, among other things, some uses for these classes: `java.util.Properties`, `java.lang.System`, and `java.lang.Runtime`, but we have only barely scratched the surface. There are many more methods available in these classes with which you can do lots more.

4.7 WHAT YOU STILL DON'T KNOW

The biggest topic in this area that we've avoided for now is the Java Native Interface (JNI), a mechanism whereby you can get outside of the Java environment to make calls to existing (*native*) libraries—for example, Linux system calls. In a coming chapter we'll actually give you an example of such a call.

Then you'll really be able to make your application nonportable and system-dependent. (But sometimes portability isn't your goal, right?)

4.8 RESOURCES

Perhaps the best resource for the specifics that you'll need to work with the topics mentioned in this chapter is the Javadoc documentation on the classes that we have mentioned. Learn to read Javadoc pages (see Section 3.2.2.3), bookmark them in your browser, and keep them handy as you write your Java code.

Chapter 5

The Sun Microsystems Java Software Development Kit

The Sun Microsystems Java Software Development Kit (Java SDK) is the most basic toolkit for Java development. In some ways, it remains the most flexible. Your understanding of Java development should include this very basic toolset, even if you move beyond it to more "hand-holding" Integrated Development Environments (IDEs). This chapter introduces all the major components of the Java 2 Standard Edition (J2SE)[1] development kit. The Enterprise Edition is discussed later. There is a third Java SDK (the Micro Edition) for embedded development which we will not cover in this book.

1. What does the "2" in "Java 2" mean? Hoo boy. Explaining product marketing names is not always easy. There was Java 1.0. Then there were several releases of Java 1.1.x. Then Sun released Java 1.2.x, but they started calling it "Java 2". Since then, they have released Java 1.3.x, Java 1.4.x and they still call it "Java 2". But it gets even more confusing. Sun is now releasing what had been preliminarily numbered 1.5, but is now officially called the 5.0 release, though they still call it Java 2. That's what the 2 in "J2SE" refers to. So it is Java 2 platform, version 5.0. Any questions? See `http://java.sun.com/j2se/naming_versioning_5_0.html`.

5.1 What You Will Learn

In this chapter you will learn about the purpose of the programs in the Sun Java Software Development Kit. We will provide you with some details of their options and demonstrate their use in compiling our two sample applications. The next step up in tool automation would be the build tool known as Ant (see Chapter 9 for more information).

5.2 All You Need, and Not One Thing More

These days, many programmers are what we affectionately call "tool junkies." They can only develop software with the support of complex integrated development environments, their supporting classes, and screen painting tools. By this, we do not mean to imply that we are Luddites. The right IDE can indeed be an enormous boost to productivity, but a programmer should be able to work with any tool. The Sun Java SDK is the lowest common denominator; if you can be productive with it, then you can be productive with absolutely any Java development environment. That makes your skills more portable. And that means more jobs are open to you. And that is good for you and your employers.

The Sun Microsystems Java SDK (formerly known as, and often still referred to as the Sun Microsystems Java Development Kit, or JDK) provides you with all the tools you need to compile, document, run, package, debug, and deploy Java applications and applets. It does this with a collection of purely text-based command-line tools. This is no-frills software development. But a lot of us crusty old types really like that.

You should become comfortable and familiar with these tools. Some IDEs are just fancy window dressing that calls these tools underneath (some are not—some have written their own Java compilers, for example). If you can use these tools comfortably to build any kind of Java program, then you know you have a mastery of the basics and are not "addicted" to a particular tool. You also know the "hardest" way to get the job done. This will help you to make good choices about tools that enhance productivity. Some tools, we find, actually slow you down or get in your way in some cases. If you know the lowest level, you can better recognize the merits and flaws of more advanced tools. Enough justification. On to the SDK.

> **NOTE**
>
> By the way, you will notice that we do not cover the installation of the Development Kit in this chapter. That is because we are deferring the discussion of installation for the next chapter, where we also introduce the concept of multiple concurrent Development Kits on a single box. See Chapter 6 for details.

5.3 THE JAVA COMPILER

At the heart of the SDK is **javac**, the Java compiler. The general form of **javac** follows:

```
javac [option...] [sourcefile...] [@optfile...]
```

The *option* list may be zero or more command-line options. We'll detail those later. The *sourcefile* list may be the name of zero or more Java source files. Usually you specify just the "main" class of an application. As we will describe later, **javac** generally will compile all necessary `.java` files for any classes that `main()` class references, directly or indirectly. If you prefix a filename with the at sign (`@`), the contents of the file will be treated as if they had been typed on the command line.

5.3.1 Compiler Behavior, Defaults, and Environment Variables

In the simplest case—compiling a single class, such as our `FetchURL.java` class—you get no diagnostics on success (Example 5.1).

Example 5.1 Compiling `FetchURL.java`

```
$ javac FetchURL.java
$
```

There will now be a new file, `FetchURL.class`, in the directory with the Java source file. Let's run that again with a command-line option we will detail later (Example 5.2).

Boy, our single, simple, one-class application sure uses a lot of classes! It does. Where did they come from? They come from the classes referenced by

Example 5.2 Compiling `FetchURL.java` with the `-verbose` option

```
$ javac -verbose FetchURL.java
[parsing started FetchURL.java]
[parsing completed 479ms]
[loading /usr/java/j2sdk1.4.1_02/jre/lib/rt.jar(java/lang/Object.class)]
[loading /usr/java/j2sdk1.4.1_02/jre/lib/rt.jar(java/net/URL.class)]
[loading /usr/java/j2sdk1.4.1_02/jre/lib/rt.jar(java/lang/String.class)]
[checking FetchURL]
[loading /usr/java/j2sdk1.4.1_02/jre/lib/rt.jar(java/lang/Exception.class)]
[loading /usr/java/j2sdk1.4.1_02/jre/lib/rt.jar(java/lang/Throwable.class)]
[loading /usr/java/j2sdk1.4.1_02/jre/lib/rt.jar(java/io/BufferedReader.class)]
[loading /usr/java/j2sdk1.4.1_02/jre/lib/rt.jar(java/io/InputStreamReader.class)]
[loading /usr/java/j2sdk1.4.1_02/jre/lib/rt.jar(java/net/URLConnection.class)]
[loading /usr/java/j2sdk1.4.1_02/jre/lib/rt.jar(java/io/Reader.class)]
[loading /usr/java/j2sdk1.4.1_02/jre/lib/rt.jar(java/io/InputStream.class)]
[loading /usr/java/j2sdk1.4.1_02/jre/lib/rt.jar(java/lang/System.class)]
[loading /usr/java/j2sdk1.4.1_02/jre/lib/rt.jar(java/io/PrintStream.class)]
[loading /usr/java/j2sdk1.4.1_02/jre/lib/rt.jar(java/io/FilterOutputStream.class)]
[loading /usr/java/j2sdk1.4.1_02/jre/lib/rt.jar(java/io/OutputStream.class)]
[loading /usr/java/j2sdk1.4.1_02/jre/lib/rt.jar(java/lang/Error.class)]
[loading /usr/java/j2sdk1.4.1_02/jre/lib/rt.jar(java/net/MalformedURLException.class)]
[loading /usr/java/j2sdk1.4.1_02/jre/lib/rt.jar(java/io/IOException.class)]
[loading /usr/java/j2sdk1.4.1_02/jre/lib/rt.jar(java/lang/RuntimeException.class)]
[loading /usr/java/j2sdk1.4.1_02/jre/lib/rt.jar(java/lang/StringBuffer.class)]
[wrote FetchURL.class]
[total 3469ms]
$
```

the application, either directly through composition or inheritance, or indirectly because the classes we used are themselves composed of or inherit from other classes. How did the Java compiler know where to find these classes? For this, it used what the Sun documentation calls a *bootstrap classpath*, which is set when the SDK is installed. A classpath is a list of directories and/or JAR files that are searched for classes. We seem to dimly recall that in early versions of Java, there was only one classpath, and if you changed it, you had to remember to put the Java runtime JAR file on it, or none of the standard APIs were available. This, no doubt, is why Sun created the concept of a bootstrap classpath. If you use any third party JAR files or you create your own, you must tell the compiler about it by creating your own classpath.

There are two ways to provide a classpath to the Java compiler. One is through a command-line switch, which we will cover in a moment. The other is through an environment variable. The `CLASSPATH` environment variable lists

Example 5.3 Setting the `CLASSPATH` environment variable

```
$ export CLASSPATH=/home/mschwarz/java/simpleApp:/var/java/lib/project.jar
$ echo $CLASSPATH
/home/mschwarz/java/simpleApp:/var/java/lib/project.jar
$
```

directories and/or JAR or ZIP files that contain classes. Each directory or JAR file is separated from the others by a colon (":"), as shown in Example 5.3.

The classpath for the compiler consists of the bootstrap classpath plus the user-specified classpath. What does the classpath mean in terms of Java class names? Think of the classpath as a list of "package roots." In other words, when you refer to a class like `java.sql.DriverManager` or `net.multitool.SAMoney`, the Java compiler is going to go to each entry in the combined bootstrap-and-user classpath and check there for `java/sql/DriverManager.class` or `net/multitool/SAMoney.class`. If it doesn't find the `.class` file in a candidate directory, it will look for the `.java` file. If it finds the `.java` file, it will compile it and then use the resulting `.class` file. When it has a `.class` file for the class, it stops searching the classpath. In this way, compiling the single "main" class of an application will often compile the whole application (we will get to exceptions to that rule later).

5.3.2 javac Options

The Java compiler has many command-line options that modify its behavior. We will go over the most important ones here. This is not a complete reference! See the Sun SDK Documentation for complete reference information.

`-classpath`
: Sets the classpath. This overrides the `CLASSPATH` environment variable, if one is specified.

`-d`
: This switch is followed by a directory name. Compiled classes are placed in that directory. Normally, compiled classes are placed in the same directory as the source code.

`-deprecation`
: This causes every use or reference to a deprecated class or method to be displayed on compilation.[2]

`-g`
: Put full debugging information in the compiled class files. See also `-g:` (the next entry in this list).

`-g:`*`keyword_list`*
: This switch gives you fine-grained control over the amount of debug information included in compiled class files. The argument after the colon may be either `none`, in which case no debug information is included, or a comma-separated list with any combination of `source`, to include source file debugging information, `lines`, to include line number information, or `vars`, to include information about local variable names. The default, if no `-g` flag of any kind is specified, is to include source file and line number information only.

`-nowarn`
: Disables warning messages.

`-verbose`
: Causes the compiler to output information about each class encountered during compilation. This can be helpful when trying to resolve problems with missing class definitions.

There are also a number of switches that relate to cross-compiling as well as UNIX-specific options, but these are not commonly used. Refer to the Sun Java SDK Tools documentation if you need details on these options.

---

2. In Java, it is rare for APIs to break support for existing code. Rather than remove old methods, it is more common to *deprecate* them. This is done by putting a `@deprecated` tag in a Javadoc comment on the class or method. The Java compiler will issue a warning (if the `-deprecated` switch is on) whenever a deprecated class or method is used. In general, deprecation is a warning that the class or method will be removed in a future version of the code or library. It is interesting to note that the **javac** compiler records that a method or class is deprecated in the binary. Thus the compiler produces different output based on the contents of a comment. As we have written simple compilers and interpreters, this creeps us out. We have always wondered why `deprecated` has not become a Java language keyword.

5.4 THE JAVA RUNTIME ENGINE

You can download and install just the software required to run Java applications without the development tools. This is known as the *Java 2 Runtime Edition*.

5.4.1 The Basics

You run a Java program by invoking the **java** command. Usually, the argument to the command is a class name. That class is loaded, and its `main()` method is run.

Remember, this is not a filename, but a class name! The rest of the command-line arguments that follow the class name are passed as an array of `String`s to the `main()` method of the named class. Example 5.4 demonstrates running the FetchURL program.

Example 5.4 Running a Java program with **java**

```
$ java FetchURL http://www.yahoo.com/news
http://www.yahoo.com/news:
<html>
<head><title>Yahoo! Directory News and Media</title>
<script type="text/javascript"
  src="http://us.js1.yimg.com/us.yimg.com/lib/common/yg_csstare.js">
</script>
<style>
  li { font-size: 12px; margin-bottom: 2px; }
</style>
<base href=http://dir.yahoo.com/News_and_Media/></head>
<body>
...
... etc.
```

> **CAUTION**
>
> Remember that if you override the classpath, either with the `-classpath` or `-cp` command-line options (detailed below) or with the `CLASSPATH` environment variable, you must include the "`.`" directory somewhere in the classpath if you want Java to include current working directory in the search list. Since "`.`" is the default classpath, many people are surprised when they set a classpath and suddenly can no longer run a `.class` file in their current directory.

The class invoked must be findable on either the bootstrap or user-specified classpath.

5.4.2 java Options

Just as the Java compiler, the runtime program, **java**, takes a number of command-line options. Here are the most commonly used ones:

`-classpath` **or** `-cp`
: Sets the runtime classpath. Overrides any value in the `CLASSPATH` environment variable.

`-Dproperty=value`
: Allows a system property to be set on the command line.

`-jar`
: Specifies that the first nonoption command-line argument is not a Java class name, but the name of a JAR file. The JAR file must have a `Main-Class:` specification in its `MANIFEST` (see Section 5.11). The `main()` method of the class named by the JAR's `MANIFEST Main-Class:` specification will be called as if that class had been named on the command line, rather than the JAR file. This is commonly used in shell scripts and batch files that accompany Java applications distributed in single `.jar` files.

There are several other command-line options that are less commonly used. We will cover some of them in later chapters when their use will make more sense. Of course, full documentation on all options for this command is in the Sun Java SDK Development Tools documentation.

5.5 Complete, Up-to-Date Program Documentation Made Easy

One of Java's most useful features is **javadoc**, a command that (by default) produces comprehensive HTML program documentation directly from the program source. Since it works from the source, it can be automated, and you may be certain that the documentation is up-to-date. It takes much of the documentation burden off of programmers and permits new programmers to join a project and rapidly come up to speed because there is comprehensive documentation in a standard format. The **javadoc** tool produces HTML

documentation by default, but this is because it uses a *doclet* that produces HTML documentation. You can write your own doclet that produces whatever format you wish. Most find the HTML documentation so satisfactory that custom doclets are rare.

Javadoc can be a large topic, because it not only documents all classes, methods, and class variables, but can also use detailed text from specially formatted comments in the source code. We will cover Javadoc comments only briefly here, but you will see examples in our project code throughout this book.

5.5.1 Running javadoc

The **javadoc** command has the following general form:

```
javadoc [options...] [package names...] [source filenames...]
 [@optfile...]
```

Options are covered in the next section. You can specify the classes to document in two ways. First, you can list one or more Java packages on the command line. Source code for the named packages is searched for on the source classpath (see Section 5.5.2). Wildcards are not permitted in package names.

Second, you may list as many Java source files as you like, and you may use wildcards in the names.

As with the **javac** compiler above, the `@optfile` allows you to name a text file whose lines are treated as arguments as if they had been typed on the command line.

Example 5.5 shows how to run **javadoc** on our small multiclass sample.

In this case, we were in the "base directory" of the package when we ran the command. In other words, `net` was a subdirectory of the current working directory when we ran Javadoc. Javadoc uses the same default classpaths and environment variables as **javac** does, so by default "." is on the path.

Generally, specifying packages is the most convenient way to document a number of classes, since packages are how collections of classes are generally managed in Java development.

Figure 5.1 shows the main screen of the documentation thus produced.

Example 5.5 Running **javadoc** with defaults against a package

```
$ javadoc net.multitool.Payback
Loading source files for package net.multitool.Payback...
Constructing Javadoc information...
Standard Doclet version 1.4.1

Generating constant-values.html...
Building tree for all the packages and classes...
Building index for all the packages and classes...
Generating overview-tree.html...
Generating index-all.html...
Generating deprecated-list.html...
Building index for all classes...
Generating allclasses-frame.html...
Generating allclasses-noframe.html...
Generating index.html...
Generating packages.html...
Generating net/multitool/Payback/package-frame.html...
Generating net/multitool/Payback/package-summary.html...
Generating net/multitool/Payback/package-tree.html...
Generating net/multitool/Payback/Account.html...
Generating net/multitool/Payback/Cost.html...
Generating net/multitool/Payback/DebtAccount.html...
Generating net/multitool/Payback/Payback.html...
Generating net/multitool/Payback/Purchase.html...
Generating net/multitool/Payback/SavingsAccount.html...
Generating package-list...
Generating help-doc.html...
Generating stylesheet.css...
$
```

5.5.2 Javadoc Command-Line Options

As with other options documentation in this chapter, this is not intended to be a complete reference document. We are documenting only the most important command-line switches.

`-public`

Causes only public classes, members, and methods to be documented. You might want this for end-user documentation of a library.

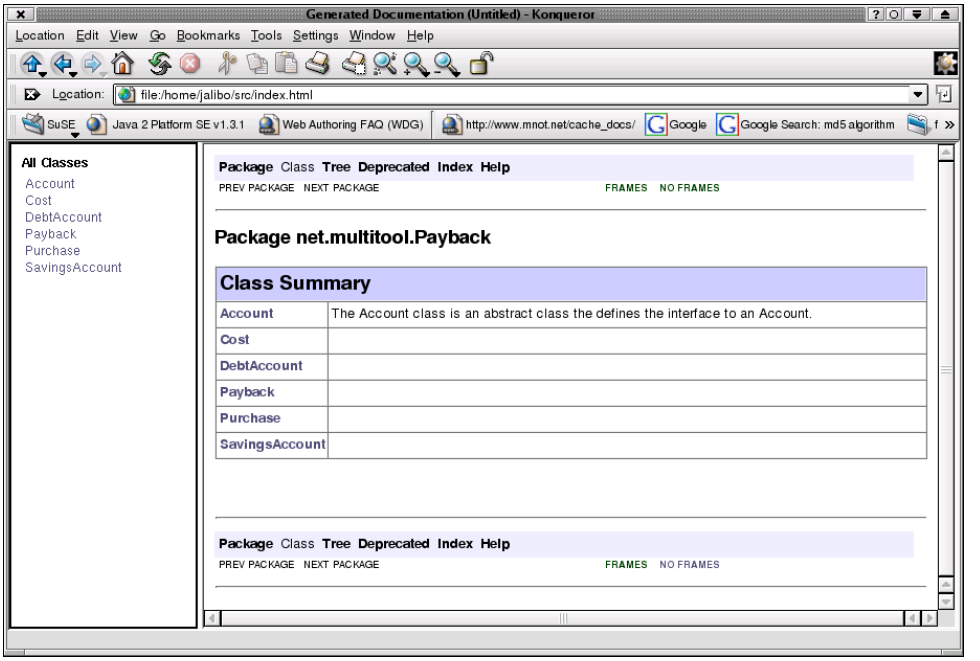

Figure 5.1 Javadoc documentation viewed in Konqueror Web browser

`-protected`
: Causes public and protected classes, members, and methods to be documented. *This is the default documentation level.* This is also the most likely level at which you would want to document code meant for distribution.

`-package`
: We suspect you can see where this is going. This switch causes package, protected, and public classes, members, and methods to be documented.

`-private`
: This switch causes all classes, members, and methods to be documented. In our experience, this is the setting you will want to use for internal projects. It documents everything.

`-sourcepath` **and** `-classpath`
: These are the paths that will be searched for source classes or referenced classes. These switches work like the corresponding switches for the **javac** compiler.

`-verbose` **and** `-quiet`

These switches control how much output is produced as **javadoc** runs. If you choose `-verbose`, detailed information is produced (more than the default; in current versions, this option mostly shows time measurements of the parsing of each source file). If you choose the `-quiet` option, progress messages are suppressed completely.

`-doclet` *`starting_class`*

We're not going to go into too much detail on this, but this switch allows you to name a doclet (a class that uses the Doclet API) to use in place of the default doclet. See the next paragraph for more information.

All of the switches documented so far are provided by the **javadoc** program itself. Javadoc, like the rest of the Sun Microsystems Java SDK, is written in Java. The authors of **javadoc** took advantage of this. The default behavior of **javadoc** is to produce HTML documentation with a standard look and feel. However, there exists an API, called the Doclet API, which allows you to write a Java class of your own to process the information parsed out of the source by **javadoc**. For details, see the Doclet Overview[3] on Sun's Web site.

Sun provides a default doclet that produces HTML documentation. That doclet takes a number of command-line options as well. We'll cover the most important of those now. Remember, these are provided by the standard doclet. If you use the `-doclet` switch, then these switches will not be available (unless, of course, the alternate doclet just happens to provide them).

`-d` *`directory`*

By default, the HTML documentation is saved in the same directory as the source. Use this switch to specify an alternate directory into which documentation is to be placed.

`-use`

Causes **javadoc** to generate a "Use" page for each class and package. Such a page is a cross-reference to all uses of the class or package.

`-version`

Causes any `@version` tag data to be included in the documentation. If you are using CVS for source control (and why wouldn't you?) we

3. `http://java.sun.com/j2se/1.4.1/docs/tooldocs/javadoc/overview.html`

recommend adding `$Id$` after the version tag, which CVS will automatically replace by its ID string containing the filename, CVS revision number, date/time and the author of last check-in. (For more about CVS, see Chapter 8.)

`-author`
: Causes any `@author` tag data to be included in the documentation.

`-splitindex`
: Causes the alphabetical index to be broken into multiple pages, one per letter. Can be useful when you have a very large number of classes and/or packages documented in a single Javadoc document set.

`-windowtitle` *`title`*
: Sets the title for the document set. The text that follows this switch will go into the HTML `<title>` element on documentation pages.

`-nodeprecated`
: This causes all deprecated methods and classes to go undocumented. Normally they are documented, but marked as deprecated.

`-nodeprecatedlist`
: Drops deprecated classes and methods from indexes, lists, and cross-references, but leaves the actual documentation in place.

There are actually many more switches. Some of the most important that we haven't covered are the `-link`, `-linkoffline`, and related tags. If you end up producing many API packages and document them separately, you can use these switches to link your separate Javadoc documentation together seamlessly, so that when you use classes from separately documented packages, the references in the documentation for your code will be live links to that separate documentation. For details on these and other switches, see the Sun documentation on **javadoc**.[4]

4. `http://java.sun.com/j2se/1.4.2/docs/tooldocs/solaris/javadoc.html`

5.5.3 Javadoc Comments

There's more to Javadoc than just documenting the types and names of classes, methods, and arguments. A developer can annotate or supplement the documentation by placing specially formatted comments in his or her code.

A Javadoc comment begins with the C-style open comment plus at least one more asterisk. It ends with a C-style close comment. In other words:

```
/* This is a C-style comment, but it
 is _not_ a Javadoc comment. */

/** This is a C-style comment, but it
 is also a Javadoc comment. */
```

This isn't a part of the Java programming language. It is merely a lexical hack to allow the **javadoc** program to recognize a comment it should pick up and process. Javadoc is fairly intelligent about where to place the text extracted from a Javadoc comment. For example, a Javadoc comment placed just before the start of a class will appear in the class summary on the package page and at the top of the class detail page. A Javadoc comment placed just before a method will appear in the method's box on the class detail page, and so on.

We encourage you to discover for yourself the relationship between Javadoc comments and the output of the standard doclet. Use it. Experiment. Or, you can go and read the official Sun Microsystems documentation on Javadoc.[5] That's your choice.

Since comment text is extracted and placed into certain positions in an HTML document, you may use HTML tags in your comments to affect how they are rendered. Be aware that when you do so, you may get unexpected results if you use any custom doclets.

There's more to it than that, however. There are a number of macros that you can place in Javadoc comments to mark data of particular significance. For example, `@author` should appear just before the name of the author of a particular piece of code.

These *at-tags* must appear after all descriptive text in a Javadoc comment.[6] A tag must be at the beginning of a line within the comment (ignoring any

5. `http://java.sun.com/j2se/1.4.2/docs/tooldocs/solaris/javadoc.html`

6. The exception is embedded tags, which we will discuss in a moment.

Example 5.6 Sample Javadoc comment with at-tags

```
/**
 * addWait - adds in the given wait time to all the counters;
 * we could say much more about the method here, but let me say
 * that we sometimes include HTML tags directly in our comments.
 * Since Javadoc will run all our text together, we may need: <br>
 *         break tags <br>
 *         or paragraph tags <br>
 *         for spacing and separation.
 * <p>We also add <i>other</i> HTML tags for <b>emphasis</b>.
 * <p>You should still try to make the comment readable, though,
 * for the programmer who is editing the source, not
 * just for those looking at the formatted Javadoc.
 * @author John Q. Programmer
 * @version $Id$
 *
 * @param delay - elapsed time, in milliseconds
 * @throws TakesTooLongException
 * @returns total time, in milliseconds
 *
 * @see net.multitool.util.TakesTooLongException, net.multitool.ctrl.Time#count
 *
 */
public long
addWait(long delay)
{
  // ...
}
```

preceding whitespace or asterisks). The tag's data is everything from the end of the tag to the end of the line (Example 5.6.)

Here are the standard at-tags:

`@author`
: Everything from the tag to the end of the line is taken as the name of the code's author.

`@deprecated`
: Marks the method or class deprecated. This tag may be optionally followed by explanatory text. If present, this text should describe when and why the class or method was deprecated and what programmers should use instead.

`@exception` **or** `@throws`
: Only valid in the comment for a method or constructor. This tag is followed by the name of an exception class (a descendant of `java.lang.Exception`) and optionally by additional explanatory text. The intent is to list the exceptions that the method throws.

`@param`
: Only valid in the comment for a method or constructor. This tag should be followed by the name of a parameter to the method followed by descriptive text. This is used to document method and constructor parameters.

`@return`
: Only valid in the comment for a method.[7] This tag is followed by descriptive text meant to document the return value of the method.

`@see`
: Populates a "See Also" section in the documentation that will provide hyperlinks to related content. There is a general format for linking to any URL, but the most common use is to refer to other elements in the same Java program. See below for the general format of such links.

In addition to these standard at-tags, there are other at-tags that may be embedded in any comment text—either the comment itself or in text that is an argument to a standard at-tag.

Such tags are placed within curly braces, for example `{@example}`, within a Javadoc comment. The one we use the most is the `@link` tag, which allows you to make a reference to another package, class, method, or class member. The general format is the same as that for the `@see` tag:

`package_name.class_name#member_or_method_name`

Any of these elements is optional.

The embedded at-tags include:

`@docRoot`
: This tag may be used when embedding HTML anchor or image tags (`A` or `IMG` tags) in a Javadoc comment to supply the root part of the

7. But not a constructor in this case, because constructors cannot return a value.

documentation path. You should always use this instead of hard-coding the full URL, or a change in directory structure or server configuration might break all of your links.

`@link`
: Allows you to embed a cross-reference to another section of the program's documentation directly in comment text. The format of a reference is the same as that for the `@see` tag.

This list is not complete. As always, see the official documentation[8] for details.

5.6 Dispensing with Applets

While this book does not cover writing applets, we should mention that, since an applet does not (generally) have a `main()` method, you need something else to launch it outside of a browser. Enter **appletviewer**. This program provides an execution environment for applets.

Why No Applets?

The decision to not cover applets was based both on limited space and on some deployment issues with applets. A surprisingly large number of people are running with either very old browsers that support only Java 1.1.x features, or that support Java runtime plug-ins but do not have them installed or enabled. Also, applets have severe limitations (for sound security reasons), and enabling various features requires a good understanding of the `SecurityManager` Java classes, which could fill a book by themselves. We chose to minimize coverage of applets for these reasons.

8. `http://java.sun.com/j2se/1.4.2/docs/tooldocs/solaris/javadoc.html`

5.7 GOING NATIVE

Now we come to the deeper darker mysteries. Let us take a look at **javah**. No, **javah** is not a Hebrew word. It is not some lost mystical text from the days before science supplanted magic. It is the Java C header and stub file generator.

If you are not already fairly experienced in writing, compiling, and building shared libraries in C on a Linux system, we would suggest that you skip this section, at least until you have developed intermediate skills in these areas. Otherwise, feel free to proceed.

We're going to walk you very quickly through building a Java native method here.[9] Don't worry if you don't quite follow it all. We will cover this topic at greater length elsewhere in the book. For now, we're giving you the highlights. Also be sure to check out Section 5.15. We'll point you to many additional resources on JNI (Java Native Interface) in that section.

Sounds pretty intimidating, huh? Well, depending upon your background and experience, it can be a bit intimidating. As much as this will hurt some die-hard Java purists, Java is not the right language for everything. Java's size and semiinterpreted nature in particular make Java ill-suited for the "close to the metal" tasks, such as device drivers and raw socket networking.

Fortunately, Java's designers were of this rocket-scientist breed (and so, for that matter, are your bending authors), so they gave Java programmers a back door: *native methods*. A native method is a class method whose name, arguments, and return type are declared in Java, but whose underlying implementation is written in "native code" (usually C, but it could be any compiled language that can match C's stack frame conventions).

As an example, let's implement a native method that will use the native Linux C library calls to get the current program's effective user ID and the name associated with that user.

First, we will write the Java class (Example 5.7).

You may never have seen code like that at the start of a class definition. The block declared

```
static { ... }
```

9. You might be tempted to call our comments in the introduction where we mentioned that we did not like purely pedagogical examples and that we would provide real, useful code. Well, we have to confess that there are some features of the Java language that we couldn't cram into our real-world examples. This JNI sample is one such. We admit our failure, and we apologize.

Example 5.7 Java application with a native method (`GetUser.java`)

```
public class GetUser {
  static {
    System.loadLibrary("getuser");
  }

  public native String getUserName();

  public static void main(String[] args)
  {
    GetUser usr = new GetUser();

    System.out.println(usr.getUserName());
  }
}
```

is called a *static initializer* and we'll discuss it in a moment.

Once you have the Java code, compile it with **javac**. You now have the compiled class. The next step is to use the **javah** tool to build the header file for your C code.

```
$ javah GetUser
```

Example 5.8 shows the header file thus produced.

Note that you run **javah** on the class file, not on the source file. The normal class name to classpath mappings apply. The file produced as a result is called, in this case, `GetUser.h`. The next step is to write the C code that implements the method (Example 5.9).

There's a lot going on here. First, the constant, `L_cuserid`, is defined in `stdio.h`; it represents the number of characters required to hold a user name. We're defining a char array to hold that number of characters plus one.[10] We are then calling the `cuserid()` function (see the manpage of `cuserid(3)`) to get the user name of the effective user ID of the process.

That much is familiar C. But what is the argument list? Our method took no arguments. And what's with the functions being called through the pointer argument?

10. What can we say? We're paranoid about the trailing null. Sue us.

Example 5.8 Header file for `GetUser` native methods (`GetUser.h`)

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class GetUser */

#ifndef _Included_GetUser
#define _Included_GetUser
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     GetUser
 * Method:    getUserName
 * Signature: ()Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_GetUser_getUserName
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

Example 5.9 Native method's C implementation file (`GetUser.c`)

```
#include "GetUser.h"
#include <stdio.h>

JNIEXPORT jstring JNICALL
Java_GetUser_getUserName(JNIEnv *jenv, jobject obj)
{
  char buffer[L_cuserid + 1];

  cuserid(buffer);

  return (*jenv)->NewStringUTF(jenv, buffer);
}
```

All of the Java class member data and Java class methods may be reached through the `JNIEnv` pointer argument. There are also methods provided by JNI itself. One of those is `NewStringUTF()`. Remember that Java `String`s are Unicode, not 8-bit ASCII, so you must convert to and from Unicode (UTF-8

is an 8-bit encoding for Unicode that coincides with ASCII in the low 7 bits, so it is often used for such conversions). You can think of the `JNIEnv` as a C++ class pointer, or you can think of it as a structure of data and function pointers (that's really what a C++ class is, after all). The bottom line is, it provides the means to access and manipulate the Java environment from your native code.

The second argument, `jobject`, is the "this" pointer. It points to the `GetUser` class, and it is upcast to the JNI equivalent of the Java `Object` type. If our method took parameters, they would follow these two constant arguments.

JNI is a huge topic. You can read more about it in the Sun Microsystems JNI Tutorial,[11] or in Java 2 SDK JNI FAQ,[12] or in the JNI 1.1 Specification,[13] or in the associated JDK 1.2 Update[14] or the JDK 1.4 Update.[15]

Even with all of this "We're too busy to explain things to you" going on here, we've got a lot more to cover before we are done. The next step in our little demo is to compile the C program and create a shared library of the code.

```
$ cc -c GetUser.c
$ cc -shared -o libgetuser.so GetUser.o
$ export LD_LIBRARY_PATH=.
```

The first line compiles the native method to a `.o` (object) file. The second command makes a shared library out of it. Now, refer back to the static initializer in Example 5.7. A static initializer is run before everything else in a class, even before `main()`. In this case, it uses the `loadLibrary()` method of the `System` class to load the shared library we just created. Note that library naming rules of the target OS are applied. The library is named `getuser` and on a Linux system it is assumed that that library will be in a file named `libgetuser.so`.

The last line sets an environment variable, `LD_LIBRARY_PATH`, to provide a path where Java will search for libraries. This is behavior inherited from Solaris. Linux uses **ldconfig** to maintain a list of shared libraries. Usually, a library is placed in a directory named in the file `ld.so.conf` and a memory cache of

---

11. `http://java.sun.com/docs/books/tutorial/native1.1/index.html`
12. `http://java.sun.com/products/jdk/faq/jni-j2sdk-faq.html`
13. `http://java.sun.com/products/jdk/1.2/docs/guide/jni/spec/jniTOC.doc.html`
14. `http://java.sun.com/j2se/1.4.1/docs/guide/jni/jni-12.html`
15. `http://java.sun.com/j2se/1.4.1/docs/guide/jni/jni-14.html`

these libraries is built and maintained with the **ldconfig** program. The library loader in the JVM, however, works as the shared library system in Solaris, where the `LD_LIBRARY_PATH` is searched for shared libraries. If you try a JNI method and get library errors, check your `LD_LIBRARY_PATH` first. Here, we used " . ", meaning "current directory." In practice, you wouldn't do this. You would deploy your shared library to a standard location and have `LD_LIBRARY_PATH` preset to that directory or directories. We just wanted to show you how it works here.

Let's see our class in action now.

```
$ java GetUser
mschwarz
$ su
Password:
# export LD_LIBRARY_PATH=.
# java GetUser
root
# exit
exit
$
```

To JNI or Not to JNI

We dislike religious debates. We have no desire to nail down what taints the purity of Java and what does not. A warning we do want to give you is, if you are an experienced UNIX C/C++ developer, you must resist the temptation to use JNI and native methods all over the place. The Java APIs are extensive, and there are probably classes that already do what you want to do. You will be tempted to use native methods because "you know how to do it in C." Resist. Find the Java way. JNI is a great way to introduce subtle and hard to find bugs into your Java programs. Leave that to the API and JVM coders. ;-)

That said, we don't want to discourage you from making use of JNI when it is the right way, or the only way, for what you need to do. The tool is there. Use it. Just remember what it does cost you in portability and what it may cost you in maintenance and debugging. Design decisions have costs and benefits. Try to find the balance.

Here you see the class being run, and, sure enough, it displays our username. We then run **su** to become root and (after setting that library path) run it again—and, sure enough, it tells us we are "root."

We'll talk more about JNI later in the book, but now you know enough to be dangerous.

5.8 INTRODUCING RMI

Remote Method Invocation (RMI) is a system for distributing application code over multiple hosts. It is a small part of multitier computing. Much of this book will be devoted to the how's and why's of multitier client/server computing. Here we are concerned only with the SDK tool **rmic**, the RMI compiler.

5.8.1 A Brief Introduction to RMI

Remote Method Invocation is a basic client-server model for invoking Java methods over a network.

5.8.1.1 History and Background

One of the most common problems in computing is how best to make an application available to the largest number of users at the least cost. To this end we have seen the development of "timeshare" systems with dumb terminals all over the place. We have seen the evolution of distributed GUI systems with X Windows and its network display system, and with tools like VNC (Virtual Network Console).[16] We have seen the emergence of the PC, providing autonomous computing at each worker's desktop. And finally we have seen the desktop turning slowly back into a terminal (albeit a prettier one) with the emergence of client-server computing.

What seems to have emerged from this progression is two major kinds of software systems. One is the PC and associated hardware and software. Developments here have dramatically increased the power and productivity of individual work. The PC revolution was indeed a real change throughout the world of business and technology. But even with this, there are a host of applications and business functions that require a collection of data and resources to be available to *multiple* workers at the same time. This is the second major kind

16. `http://www.realvnc.com/`

of software systems. This second kind used to be the only kind, but now it may, in a sense, be the minority of applications, but the most critical to an operation. This second class of system has come to be called *enterprise* systems.

In enterprise computing, we have the same problem we opened with: How do you make the information and resources available to everyone who needs them at the lowest cost? And the answer is (as it always is) "that depends."

These days, one of the most common solutions is to use a Web server to *publish* an application. This works well for a great many applications and it is much easier to do than many other methods. That explains its popularity. The Web interface is quite limited, however, so for more user interface intensive applications, client-server computing evolved. To us techies, all of this stuff is client-server. In this context however, client-server refers to a 2-tier system where the UI and logic exist in a GUI application on a PC and common resources are in an SQL database all the clients share.

This is also commonly used, but becomes expensive in a couple of cases. The first is when the database itself becomes a bottleneck because the number of users grows and grows but only one database can exist. The second is simply the cost of maintenance. Since the logic exists in the client, any change to the logic requires updating the software on all clients. Even when a scheme for automating this exists, it is still time-consuming and costly to get all the changes out to all users simultaneously. There are workarounds for both of these issues, but here we are concerned with a different solution altogether.

So, how can we have a UI richer than with a Web application but avoid the pitfalls of the traditional 2-tier client-server computing? The answer is to separate the UI from the business logic and the business logic from the underlying data store. This results in 3 tiers—*presentation*, *business logic*, and *data*.

Much of this book will concern itself with 3-tier computing solutions. Java has four major architectures for building 3-tier solutions. One of them is RMI.[17]

5.8.1.2 RMI Basics

RMI works by sharing an interface between the client and the server. The interface groups together the methods that a client may call on a server. A class is

17. The others are Enterprise JavaBeans, servlets, and JavaServer Pages. The latter two are Web-based, and therefore suffer from the UI deficiencies of Web forms, but Sun calls them part of Enterprise Java, so we will too.

written on the server side that implements the interface, and a special compiler is used to generate *stubs* for the server side and the client side. On the client side, a call to an RMI method looks like any other method call, but it is sent across the network to the server, where the actual instructions are carried out. Any return value is then passed back over the network to the client.

We will walk you through a very simple (and very pointless) example just to show you the tools.

5.8.1.3 Writing the Interface

Our interface is pathetically simple. It is a class that sums two integer arguments and returns an integer result. Example 5.10 shows the interface file.

> **NOTE**
>
> The names of the classes in the following examples may seem a bit strange, and they are. It is because we aim to build on this example later.

Example 5.10 The `Session` interface

```
package net.multitool.RMIDemo;

import java.rmi.*;

public interface Session extends Remote {
  public int add(int x, int y) throws RemoteException;
}
```

The two important things to note here are that the interface must extend `java.rmi.Remote` and that any remote method must be defined as throwing `java.rmi.RemoteException`. If anything goes wrong during an RMI call, like someone tripping over a network cable, the call will not complete successfully and an exception of that type will be thrown. It is not possible to have a RMI method that cannot throw this exception.

Beyond those features, you can see that defining remote methods is quite familiar and easy.

5.8.1.4 Writing the Server Class

An interface is an "empty vessel." Before any interface can be used, you must have an actual class that implements the interface. In an RMI application, the implementing class is the server (Example 5.11).

The class is named `SessionImpl` to emphasize its relationship with the `Session` interface. There is no requirement to match up such names. Likewise, the RMI name given, `//penfold/Session`, uses the interface name, but it could use any name. It is a good idea to develop a naming convention for RMI interfaces and their implementations. It is *critical* to develop naming conventions for RMI registry names, particularly in production environments. Without a naming convention, it is difficult to avoid confusion and even chaos. What happens when multiple business units develop RMI code destined for a single production server, and they have all made an RMI interface named `Session`, or `Payment`?[18] Bad things happen.

There is no "one size fits all" naming convention that we can offer. Possibilities include using package names in RMI registry names, using some element of the business area as a component of the name (such as `AccountingSession`, `ShippingSession`, `ExecutiveSession`). All that matters is that an unambiguous standard be created and followed.

Let's spend some time talking about what this code does.

First, notice that the class extends `UnicastRemoteObject`. This is not necessary, but using that as a base class saves a lot of server setup. There are times when you would want to do such setup manually, but for our purpose here it saves a lot of effort. The class also implements our remote interface.

The first method is a constructor that calls the superclass constructor. At first glance, this is pointless. Any Java class gets a default constructor that just calls the superclass constructor, so why is this here? It is here because the superclass constructor throws `RemoteException`. If we didn't define a constructor like the one here specifying that it throws `RemoteException`, the compiler would complain that there is an unhandled exception. So we define a constructor identical to a default constructor except that it specifies that it can throw the exception.

---

18. One solution is to use a more advanced naming system, such as LDAP. See Section 21.3.2.3.

Example 5.11 The `Session` server implementation

```
package net.multitool.RMIDemo;

import net.multitool.RMIDemo.*;
import java.rmi.*;
import java.rmi.server.*;

/** SessionImpl is the server class for the Session RMI interface.
 */
public class
SessionImpl
  extends UnicastRemoteObject
  implements Session
{
  /** Constructor needed to ensure call to UnicastRemoteObject
   *  constructor and to thus propagate the possible exception.
   */
  public SessionImpl() throws RemoteException {
    super();
  }

  /** A static main() for the server. */
  public static void main(String[] arglist)
  {
    if (System.getSecurityManager() == null) {
      System.setSecurityManager(new RMISecurityManager());
    }

    String rmiName = "//penfold/Session";
    try {
      Session adder = new SessionImpl();
      Naming.rebind(rmiName, adder);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  /** Implementation of the RMI method, add. */
  public int add(int x, int y) throws java.rmi.RemoteException
  {
    return x+y;
  }
}
```

Next, we have the server `main()` method. It first sets a security manager. The security manager controls what the VM is allowed to do. A number of default security managers are provided, and here we use one that is designed specifically to give safe and reasonable defaults for RMI applications. You can, of course, write your own security manager. Security managers use "policy specifications" to alter their capabilities. For now, we will explain enough to run a simple example. See Section 5.8.4.2 for more information on policies for our example.

Remember that `main()` is static, so there is no instance of `SessionImpl` yet, and thus also no instance of `Session`. We declare a variable of type `Session`, and set it to a new instance of `SessionImpl`. (There is no need to typecast here because `SessionImpl` implements `Session`, therefore `SessionImpl` is, among other things, a `Session`.) We now have an instance of the server class.

Next, the server must make itself available to the world. It does this by registering itself with the RMI registry (see Section 5.8.3). This is done through a static method of the `java.rmi.Naming` class, `rebind()`. Put simply, this maps a remote object to a string name in the registry. When clients contact the registry looking for a name then, if a remote object is mapped to that name, the communication can take place (yes, we are simplifying at the moment). The call to `rebind()` does not return. The server is up and running.

Finally, we have the implementation of our remote method, `add()`.

This looks like a lot of hassle to go through, and it is, but consider writing an interface that offers, for example, methods like `getDirContents()`, `chDir()`, `downloadFile()`, `uploadFile()`. You've just written something like an FTP server. No matter how many methods you add to your interface, the complexity of the setup code does not increase. Maybe now it looks a little more useful?

5.8.1.5 Writing the Client Class

At this point, Example 5.12 should be fairly obvious. Our class has just a single static method, `main()`. It, like our server side `main()`, sets up a security manager. It then contacts a registry on the machine named `penfold` looking for an instance of a remote interface named `Session` (again, `lookup()` is a static method of the `java.rmi.Naming` class). We store that reference in a variable of type `Session` called `sess`. We can then call the `add()` on `sess`. We'll show the server and client running shortly.

Example 5.12 The RMI client program

```
package net.multitool.RMIDemo;

import java.rmi.*;

public class Client {
  public static void main(String[] arglist) {
    if (System.getSecurityManager() == null) {
      System.setSecurityManager(new RMISecurityManager());
    }

    try {
      String name = "//penfold/Session";
      // Obtain reference to the remote object
      Session sess = (Session) Naming.lookup(name);

      System.out.println("Pointless RMI Client. 47 + 13 = " +
                         sess.add(47,13) + ", right?");
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

5.8.2 The rmic Tool

In order for a remote object to make itself available and in order for a client to be able to call such an object, each method needs a client and server-side *stub* to proxy the method call. Arguments to the method call are converted to streamable data (this process is called *marshaling*) by the client stub, and that data is sent over the network to the server stub, which must convert that stream into object instances on the server side (this is called *unmarshaling*). The server-side stub then calls the actual method implementation. When the method returns, any return values and changes to the state of the arguments must be marshaled by the server stub and sent back to the client stub where they are unmarshaled and stored in the correct locations on the client.

This was the traditionally painful part of writing multitier clients. What **rmic** does is automate the generation of these stubs, so writing a remote method is only slightly more difficult than writing any other method.

To generate RMI stubs for our application, run **rmic**[19] against the class that implements the remote interface:

```
penfold$ rmic net.multitool.RMIDemo.SessionImpl
```

When you are writing "traditional" Java RMI, that is just about all you need to know about **rmic**. The program actually has a large number of options and switches, but most of these are to support alternate protocols and systems, such as CORBA IDL and IIOP. If you know what these are, and make use of these, you will find details on these options in Sun's **rmic** tool documentation.[20]

5.8.3 The rmiregistry Tool

The **rmiregistry** is a naming service that binds RMI server stubs to simple names. Invoking it is incredibly simple. Just type `rmiregistry`. You may want to run it on other than the default port (1099). For that, just specify the port number on the command line:

```
$ rmiregistry 21099 &
```

That example shows us running a registry on port 21099 and running it in the background. You might want to use a nonstandard port in order to run a test version of the service while the production version remains available on the standard port.

That is just about all there is to **rmiregistry**. You can find details in the Sun Java SDK documentation.

5.8.4 Setting Up Servers and Clients

So far, we have written an RMI interface, a server implementation, and a client implementation. We have generated RMI stubs for our RMI object. We are

---

19. Be careful! If you have one or more Java SDKs installed *and* you have the GNU Compiler for Java installed, watch out for your `PATH`. The Java compiler and the Java runtime from the JDK don't collide with **gcj** because the compiler has a different name and **gcj** compiles to native binaries. But **gcj** does have an **rmic** compiler, and it is usually in `/usr/bin`, which is usually ahead of your JDK in the executable path. If you run **rmic** and it explodes with errors, make sure you aren't running the **rmic** from **gcj** against `.class` files from a JDK. (And, yes, this bit me and had me confused for a while!)

20. `http://java.sun.com/j2se/1.4.2/docs/tooldocs/solaris/rmic.html`

almost ready to fire our system up and give it a try. But first, we'll give you some information about our sample environment and talk very briefly about security.[21]

5.8.4.1 What RMI Servers and Clients Need to Be Able to Do

RMI servers and clients need to be able to listen for connections on network ports, and they need to be able to initiate connections on network ports. Back in the Java 1.1 days, there were no limits on what RMI methods could do. The `CLASSPATH` was assumed to be trusted. With the RMI 1.2 protocol specification, the ability to actually pass bytecodes between VMs over RMI was added. That means that it is possible for clients to pass code to servers. Obviously, this opens a lot of possible security risks. For this reason, RMI got a security management layer. It is the same security manager as the one that applets use. It also provides a default security manager class that has virtually all such capabilities safely turned off. We need to turn on some of these capabilities in order to make our sample work.

The RMI system expects Java classes to be made available through one of two paths.

1. The `CLASSPATH`, either the environment variable or on the command line.
2. Through a property that points at URL. This URL may be a `file:` URL, or an `http:` URL.

We are going to do the simplest case for now. We will have our compiled code installed on both our server system and our client system. The classes will all be referenced relative to the default classpath (in other words, relative to ".", the current directory).

This is not the typical case. The most common case will be for the classes to be available in a JAR file via a Web server, and for the `java.rmi.server.codebase` property to be set to point to the JAR file via an `http:` URL.

21. We're going to gloss over this subject for now.

Example 5.13 A Java security policy file suitable for the RMI example

```
grant {
  permission java.net.SocketPermission "*:1024-65535", "connect,accept";
  permission java.net.SocketPermission "*:80", "connect,accept";
};
```

5.8.4.2 Our Environment

We have two machines. One, `penfold`, is our server machine. The other, `grovel`, is our client machine. To keep things straight in our samples, the shell prompts will have the host names in them.

If you are using a JDK that supports the 1.2 RMI specification (and we hope you are—it's in all current JDKs), you have to give your server and your client permission to access the network ports needed to run. By default, the Java runtime will look for a security policy file in the home directory of the user running the VM. The default name of the file is `.java.policy`. Example 5.13 shows what we suggest you put in this file, at least to run this example.

> **NOTE**
>
> You will have to put this in your home directory both on the server and on all client machines.

5.8.4.3 Compiling and Running the Server

Our packages here follow Sun's suggested naming convention of your domain name, reversed, followed by your package names. It so happens that Mr. Schwarz's domain is called `multitool.net` (named after his first book, *Multitool Linux*), so we put all of these classes in a package called `net.multitool.RMIDemo`.

For all of the examples in this section, as well as the following section on building and running the client, assume that our current working directory is the directory that contains the `net` directory of our source code.

The output you see in Example 5.14 includes the result of running our client once. Note that the `SessionImpl` class doesn't terminate. It keeps running to service clients indefinitely.

Example 5.14 Compiling and running our server on `penfold`.

```
penfold$ javac net/multitool/RMIDemo/SessionImpl.java
penfold$ rmic net.multitool.RMIDemo.SessionImpl
penfold$ rmiregistry &
 17286
penfold$ java net.multitool.RMIDemo.SessionImpl
Asked to add 47 and 13
```

5.8.4.4 Compiling and Running the Client

Example 5.15 shows the actual steps we ran to build and run the client.

Example 5.15 Compiling and running our client on `grovel`

```
grovel$ javac net/multitool/RMIDemo/Client.java
grovel$ javac net/multitool/RMIDemo/SessionImpl.java
grovel$ /usr/java/jdk/bin/rmic net.multitool.RMIDemo.SessionImpl
grovel$ java net.multitool.RMIDemo.Client
Pointless RMI Client.  47+13=60, right?
grovel$
```

> **NOTE**
>
> We compile the server class, `SessionImpl`, on the client side and run **rmic** against it just to produce the stubs the client requires. You could copy the stub classes from the server machine, or you could put them in a JAR file, put that file on a Web server, and have the `java.rmi.server.codebase` property point to that JAR file. We're taking the simple way here, but in a real implementation, you would not do it this way. We'll cover more realistic cases later.

5.8.5 RMI Summary

RMI greatly simplifies the business of writing multitier client-server applications. It is suitable for many classes of distributed computing problems, but it does lack several features that required in large, mission-critical applications. For one thing, it lacks any sort of transaction support. If a method invocation fails, the client may not know for certain whether the server finished some work, like writing to a database, before the failure. Also, the **rmiregistry** program is

a very simplistic naming/lookup system. Clients must know where to find the registry with the resources they need.

RMI is very useful for problems of a certain scale, but it is not, in and of itself, sufficient for high-volume, highly available, mission-critical enterprise systems.[22] But that is what J2EE and EJB are for. We'll deal with those in Part V later in the book.

5.9 THE JAVA DEBUGGER

How can you stand using the SDK? It doesn't even have a debugger!

Wrong. It has a debugger. It just has an extremely basic command-line debugger. Example 5.16 shows the output of its **help**.

Again, we are not going to document everything here. That's what the online Sun Microsystems Java SDK documentation is for. Instead, we will use the debugger to step through the execution of our simple application and show you some of the debugger's basic operations.

There are two ways to invoke **jdb**. One is to attach it to an already running JVM that has been started with remote debugging enabled. See the Java SDK documentation for details on that method. Here we'll show you the simpler case of invoking the program locally by running the application directly under the debugger.

The basic invocation is:

```
$ jdb
```

You may optionally name the class whose `main()` is to be executed under the debugger, but we usually use the **run** from inside the debugger itself to do this. Remember that if you want to be able to view local variables in the debugger, you must have compiled your class or classes with the `-g` option of **javac**.

In the rest of this section, we will examine an actual debug session. We will run our single-class application, FetchURL, and use it to retrieve the `index.html` file from the Web server on the laptop on which this chapter is being written. To refresh your memory, remember that the source code for FetchURL is at Example 3.30. Example 5.17 is what that file looks like.

22. If that sentence did not cause you to get "buzzword bingo," then you aren't trying.

Example 5.16 The Java debugger **help** command output

```
$ jdb GetUser
Initializing jdb ...
> help
** command list **
run [class [args]]        -- start execution of application's main class

threads [threadgroup]     -- list threads
thread <thread id>        -- set default thread
suspend [thread id(s)]    -- suspend threads (default: all)
resume [thread id(s)]     -- resume threads (default: all)
where [thread id] | all   -- dump a thread's stack
wherei [thread id] | all  -- dump a thread's stack, with pc info
up [n frames]             -- move up a thread's stack
down [n frames]           -- move down a thread's stack
kill <thread> <expr>      -- kill a thread with the given exception object
interrupt <thread>        -- interrupt a thread

print <expr>              -- print value of expression
dump <expr>               -- print all object information
eval <expr>               -- evaluate expression (same as print)
set <lvalue> = <expr>     -- assign new value to field/variable/array element
locals                    -- print all local variables in current stack frame

classes                   -- list currently known classes
class <class id>          -- show details of named class
methods <class id>        -- list a class's methods
fields <class id>         -- list a class's fields

threadgroups              -- list threadgroups
threadgroup <name>        -- set current threadgroup

stop in <class id>.<method>[(argument_type,...)]
                          -- set a breakpoint in a method
stop at <class id>:<line> -- set a breakpoint at a line
clear <class id>.<method>[(argument_type,...)]
                          -- clear a breakpoint in a method
clear <class id>:<line>   -- clear a breakpoint at a line
clear                     -- list breakpoints
catch [uncaught|caught|all] <exception-class id>
                          -- break when specified exception occurs
ignore [uncaught|caught|all] <exception-class id>
                          -- cancel 'catch' for the specified exception
watch [access|all] <class id>.<field name>
                          -- watch access/modifications to a field
unwatch [access|all] <class id>.<field name>
                          -- discontinue watching access/modifications to a field
trace methods [thread]    -- trace method entry and exit
```

```
untrace methods [thread]  -- stop tracing method entry and exit
step                      -- execute current line
step up                   -- execute until the current method returns to its caller
stepi                     -- execute current instruction
next                      -- step one line (step OVER calls)
cont                      -- continue execution from breakpoint

list [line number|method] -- print source code
use (or sourcepath) [source file path]
                          -- display or change the source path
exclude [class id ... | "none"]
                          -- do not report step or method events for specified classes
classpath                 -- print classpath info from target VM

monitor <command>         -- execute command each time the program stops
monitor                   -- list monitors
unmonitor <monitor#>      -- delete a monitor
read <filename>           -- read and execute a command file

lock <expr>               -- print lock info for an object
threadlocks [thread id]   -- print lock info for a thread

pop                       -- pop the stack through and including the current frame
reenter                   -- same as pop, but current frame is reentered
redefine <class id> <class filename>
                          -- redefine the code for a class

disablegc <expr>          -- prevent garbage collection of an object
enablegc <expr>           -- permit garbage collection of an object

!!                        -- repeat last command
<n> <command>             -- repeat command n times
help (or ?)               -- list commands
version                   -- print version information
exit (or quit)            -- exit debugger

<class id> or <exception-class id>: full class name with package
qualifiers or a pattern with a leading or trailing wildcard ('*')
NOTE: any wildcard pattern will be replaced by at most one full class
name matching the pattern.
<thread id>: thread number as reported in the 'threads' command
<expr>: a Java(tm) Programming Language expression.
Most common syntax is supported.

Startup commands can be placed in either "jdb.ini" or ".jdbrc"
in user.home or user.dir
>
```

Example 5.17 `index.html` used in **jdb** session

```
<HTML>
  <HEAD>
    <TITLE>RedHat Linux Laptop</TITLE>
  </HEAD>
  <BODY>
    <H1>RedHat Linux Laptop</H1>
    <P>You have contacted Michael Schwarz's RedHat Linux Laptop.
    You would probably rather
    <A HREF="http://www.multitool.net/">see his permanent Web
    page</A> since this server goes up and down all the time, what
    with it being on a laptop.</P>
  </BODY>
</HTML>
```

Example 5.18 is an actual transcript of a real **jdb** session. It is annotated with explanatory comments. Our goal here is to get you going. The best way to learn **jdb**, or indeed any of these tools, is to use them.

Obviously, this little session has merely scratched the surface of the Java debugger. You can debug multithreaded applications with commands that can suspend and resume individual threads, list the running threads, switch your "executable view" between threads, and so forth. You can trace method calls. You can monitor variables. You can execute expressions (including assignment expressions, allowing you to force variables to certain values). You can browse classes. You can dump all local variables with a single command. The debugger is quite capable, if a bit limited in user interface.[23] Learn it. Play with it. Step through your favorite Java program with it.

23. As you know, we do not automatically like IDEs and GUI development tools (see Section 5.2). A debugger is an exception to that rule. When debugging, a well designed UI with a code pane, a stack pane, a data viewer, a class browser, a thread selection pane, and so on is enormously helpful. You need to be able to see all these elements nearly simultaneously; you need to see the *whole system* as it runs. The command-line debugger makes everything you need available, but with a traditional "glass-teletype" UI that is quite awkward. By all means, learn the CLI debugger, but then find a good Java debugger with a windowed UI of some kind. It is hard to say which compiler UI is the best, but I think we can safely say the command-line debugger UI is the worst! You should know it as a last resort, but use it as a last resort!

Example 5.18 An actual **jdb** session, with commentary

```
$ jdb
Initializing jdb ...
> stop in FetchURL.main (1)
Deferring breakpoint FetchURL.main.
It will be set after the class is loaded.
> run FetchURL http://localhost
run  FetchURL http://localhost
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: Set deferred breakpoint FetchURL.main

Breakpoint hit: "thread=main", FetchURL.main(), line=48 bci=0
48          for (i = 0; i < args.length; i++)

main[1] list
44       {
45          int i;
46          FetchURL f;
47
48 =>(2)     for (i = 0; i < args.length; i++)
49          {
50             System.out.println(args[i] + ":");
51             System.out.println(new FetchURL(args[i]));
52          }
53       }
main[1] step
>
Step completed: "thread=main", FetchURL.main(), line=50 bci=5
50             System.out.println(args[i] + ":");

main[1] step
> http://localhost:

Step completed: "thread=main", FetchURL.main(), line=51 bci=32
51             System.out.println(new FetchURL(args[i]));

main[1] step
>
Step completed: "thread=main", FetchURL.<init>(), line=8 bci=0
8       {

main[1] list
4    public class FetchURL {
5       private URL requestedURL;
```

```
6
7       public FetchURL(String urlName)
8 =>    {
9          try {
10             requestedURL = new URL(urlName);
11          } catch (Exception e) {
12             e.printStackTrace();
13          }
main[1] step
>
Step completed: "thread=main", FetchURL.<init>(), line=10 bci=4
10             requestedURL = new URL(urlName);

main[1] step (3)
>
Step completed: "thread=main", FetchURL.<init>(), line=11 bci=16
11          } catch (Exception e) {

main[1] step
>
Step completed: "thread=main", FetchURL.<init>(), line=14 bci=27
14       }

main[1] step
>
Step completed: "thread=main", FetchURL.main(), line=51 bci=45
51             System.out.println(new FetchURL(args[i]));

main[1] step
>
Step completed: "thread=main", FetchURL.toString(), line=19 bci=2
19          String rc = "";

main[1] list
15
16
17       public String toString()
18       {
19 =>       String rc = "";
20          String line;
21          BufferedReader rdr;
22
23          try {
24             rdr = new BufferedReader(
main[1] step
>
Step completed: "thread=main", FetchURL.toString(), line=24 bci=3
24             rdr = new BufferedReader(
```

```
main[1] step
>
Step completed: "thread=main", FetchURL.toString(), line=30 bci=28
30             while ((line = rdr.readLine()) != null)

main[1] step
>
Step completed: "thread=main", FetchURL.toString(), line=32 bci=31
32                rc = rc + line + "\n";

main[1] list
28             );
29
30             while ((line = rdr.readLine()) != null)
31             {
32 =>             rc = rc + line + "\n";
33             }
34          } catch (Exception e) {
35             e.printStackTrace();
36             rc = null;
37          }
main[1] step
>
Step completed: "thread=main", FetchURL.toString(), line=30 bci=55
30             while ((line = rdr.readLine()) != null)

main[1] step
>
Step completed: "thread=main", FetchURL.toString(), line=32 bci=31
32                rc = rc + line + "\n";

main[1] step
>
Step completed: "thread=main", FetchURL.toString(), line=30 bci=55
30             while ((line = rdr.readLine()) != null)

main[1] step
>
Step completed: "thread=main", FetchURL.toString(), line=32 bci=31
32                rc = rc + line + "\n";

main[1] dump this  (4)
 this = {
    requestedURL: instance of java.net.URL(id=378)
}
main[1] dump rc    (5)
  rc = "<HTML>
          <HEAD>
```

```
"
main[1] list 36
32                rc = rc + line + "\n";
33             }
34          } catch (Exception e) {
35             e.printStackTrace();
36 =>          rc = null;
37          }
38
39          return rc;
40       }
41
main[1] stop at FetchURL:39     (6)
Set breakpoint FetchURL:39
main[1] cont
>
Breakpoint hit: "thread=main", FetchURL.toString(), line=39 bci=79
39          return rc;

main[1] dump rc
  rc = "<HTML>
          <HEAD>
            <TITLE>RedHat Linux Laptop</TITLE>
          </HEAD>
          <BODY>
            <H1>RedHat Linux Laptop</H1>
            <P>You have contacted Michael Schwarz's RedHat Linux Laptop.
            You would probably rather
            <A HREF="http://www.multitool.net/">see his permanent Web
            page</A> since this server goes up and down all the time, what
            with it being on a laptop.</P>
          </BODY>
        </HTML>
"
main[1] step
> <HTML>
    <HEAD>
      <TITLE>RedHat Linux Laptop</TITLE>
    </HEAD>
    <BODY>
      <H1>RedHat Linux Laptop</H1>
      <P>You have contacted Michael Schwarz's RedHat Linux Laptop.
      You would probably rather
      <A HREF="http://www.multitool.net/">see his permanent Web
      page</A> since this server goes up and down all the time, what
      with it being on a laptop.</P>
    </BODY>
  </HTML>
```

```
Step completed: "thread=main", FetchURL.main(), line=48 bci=48
48          for (i = 0; i < args.length; i++)

main[1] step
>
Step completed: "thread=main", FetchURL.main(), line=53 bci=57
53       }

main[1] step
>
The application exited
$
```

---

1. Here we tell the debugger where to break execution to let us run debugger commands. We do so at the start of the `FetchURL` class' `main()` method. If we did not set a breakpoint, the `run` would have run the program to termination, making it no different from running it with the **java** command (except perhaps a bit slower).

2. The **list** command shows the source line that is about to be executed, along with some more lines to either side. It is a handy way to get a little context. The standard "next line" prompt isn't enough for most of us to get context (unless, of course, we are looking at a line-numbered printout of the source or an editor window at the same time, which we often do).

3. The **step** steps execution one "line" (what a line is can be a bit fuzzy when there's a lot of vertical whitespace in the source, or when multiple method calls occur on one line). Note the information in the status message. The name of the thread is given (our sample is single-threaded, so it is always "main"), as is the line number in the source file and the `bci`. Note that there is a very similar command, **next**, that advances to the next line *in the same stack frame*. In other words, it won't step into method calls, it steps over them.

4, 5. Here we see two uses of the **dump** command. First, we apply it to `this` (which is an implicit argument to any nonstatic method call) to dump the currently executing object. The second instance dumps the `rc` local variable, which is an accumulating string containing the requested Web page. At the moment, it contains only the first few lines.

6. Here we set a breakpoint on a specific source line number. We then use the **cont** command to resume "full speed" code execution.

5.10 Return to the Source: The Java Decompiler

Java includes a decompiler of sorts called **javap**. It is sometimes referred to as the "class file disassembler." We titled this section "Return to the Source," but it is a bit misleading; **javap** simply provides a way to examine the members and methods of a compiled Java class[24] even when you do not have its source code.

The **javap** command takes the same access-modifier command-line arguments as **javadoc** (`-public`, `-protected`, `-package`, `-private`) to determine which attributes and methods are to be reported. An additional switch, `-c`, causes the bytecodes of methods to be reported. For details, see Sun's documentation for **javap**.[25]

Example 5.19 shows what you get if you run `javap -c` on our FetchURL example.

5.11 Bundling a Java Program: Put It in a JAR

Distributing a Java application can be a pain. All but the simplest of applications will have many public classes—and since there can only be one public Java class per source file, each Java source file becomes a class file, and the elements of a package name become directory nodes in the path to the class, you end up with a fairly complex collection of directories and files. Wouldn't it be nice to be able to roll the whole mess up into a single binary file for distribution?

Well, you can. The tool to do the job is called **jar**, which stands for Java ARchive.[26] The files produced by this utility are called *JAR files*. The JAR format is the common DOS/Windows ZIP file format, with a few special files to support some special features we will explain as they come up.

---

24. In Chapter 7 we will introduce **gcj**, the GNU Compiler for Java, which compiles Java to native machine code. **javap** is useless with such a file. It deals only with JVM bytecodes as documented in Sun's JVM Specification.

25. `http://java.sun.com/j2se/1.4.2/docs/tooldocs/solaris/javap.html`

26. An abbreviation made up of syllables from words instead of just initials is called a *portmanteau*. The US Navy is particularly keen on them, using terms like COMSURPAC (Commander, Surface Fleet, Pacific), COMSUBLANT (Commander, Submarine Fleet, Atlantic), and so forth. There. Now you can't claim you didn't learn anything from this book.

Example 5.19 javap output for `FetchURL.class`

```
Compiled from FetchURL.java
public class FetchURL extends java.lang.Object {
  private java.net.URL requestedURL;
  public FetchURL(java.lang.String);
  public java.lang.String toString();
  public static void main(java.lang.String[]);
}

Method FetchURL(java.lang.String)
   0 aload_0
   1 invokespecial #1 <Method java.lang.Object()>
   4 aload_0
   5 new #2 <Class java.net.URL>
   8 dup
   9 aload_1
  10 invokespecial #3 <Method java.net.URL(java.lang.String)>
  13 putfield #4 <Field java.net.URL requestedURL>
  16 goto 27
  19 astore_2
  20 aload_2
  21 invokevirtual #6 <Method null>
  24 goto 27
  27 return
Exception table:
   from   to  target type
     4    16    19   <Class java.lang.Exception>

Method java.lang.String toString()
   0 ldc #7 <String "">
   2 astore_1
   3 new #8 <Class java.io.BufferedReader>
   6 dup
   7 new #9 <Class java.io.InputStreamReader>
  10 dup
  11 aload_0
  12 getfield #4 <Field java.net.URL requestedURL>
  15 invokevirtual #10 <Method java.net.URLConnection openConnection()>
  18 invokevirtual #11 <Method java.io.InputStream getInputStream()>
  21 invokespecial #12 <Method java.io.InputStreamReader(java.io.InputStream)>
  24 invokespecial #13 <Method java.io.BufferedReader(java.io.Reader)>
  27 astore_3
  28 goto 55
  31 new #14 <Class java.lang.StringBuffer>
  34 dup
  35 invokespecial #15 <Method java.lang.StringBuffer()>
  38 aload_1
```

```
  39 invokevirtual #16 <Method java.lang.StringBuffer append(java.lang.String)>
  42 aload_2
  43 invokevirtual #16 <Method java.lang.StringBuffer append(java.lang.String)>
  46 ldc #17 <String "
">
  48 invokevirtual #16 <Method java.lang.StringBuffer append(java.lang.String)>
  51 invokevirtual #18 <Method java.lang.String toString()>
  54 astore_1
  55 aload_3
  56 invokevirtual #19 <Method java.lang.String readLine()>
  59 dup
  60 astore_2
  61 ifnonnull 31
  64 goto 79
  67 astore 4
  69 aload 4
  71 invokevirtual #20 <Method null>
  74 aconst_null
  75 astore_1
  76 goto 79
  79 aload_1
  80 areturn
Exception table:
   from   to  target type
     3    64    67   <Class java.lang.Exception>

Method void main(java.lang.String[])
   0 iconst_0
   1 istore_1
   2 goto 51
   5 getstatic #21 <Field java.io.PrintStream out>
   8 new #14 <Class java.lang.StringBuffer>
  11 dup
  12 invokespecial #15 <Method java.lang.StringBuffer()>
  15 aload_0
  16 iload_1
  17 aaload
  18 invokevirtual #16 <Method java.lang.StringBuffer append(java.lang.String)>
  21 ldc #22 <String ":">
  23 invokevirtual #16 <Method java.lang.StringBuffer append(java.lang.String)>
  26 invokevirtual #18 <Method java.lang.String toString()>
  29 invokevirtual #23 <Method void println(java.lang.String)>
  32 getstatic #21 <Field java.io.PrintStream out>
  35 new #24 <Class FetchURL>
  38 dup
  39 aload_0
  40 iload_1
  41 aaload
  42 invokespecial #25 <Method FetchURL(java.lang.String)>
```

```
45 invokevirtual #26 <Method void println(java.lang.Object)>
48 iinc 1 1
51 iload_1
52 aload_0
53 arraylength
54 if_icmplt 5
57 return
```

A JAR file packages a subdirectory and its descendants into a single file. A Java `CLASSPATH` specification may contain a JAR filename everywhere it might contain a directory name. Let's say you use the GPL'ed Java personal finance program called **jgnash** and you've compiled it from source, so you have a directory off your home directory called `jgnash/bin`. Suppose you run the program by directly invoking **java** to run the class `jgnashMain` and you have `$HOME/jgnash/bin` on your `CLASSPATH`. You could clean up the mess on your hard drive by using the **jar** command to squash all the files in `jgnash/bin` together into a single JAR file, as shown in Example 5.20.

Example 5.20 Making a JAR file

```
$ cd ; mkdir jars
$ jar cvf jars/jgnash.jar jgnash/bin
```

You could then replace the `$HOME/jgnash/bin` entry in your `CLASSPATH` with `$HOME/jars/jgnash.jar`. After that you would still run **jgnash** with exactly the same **java** command you always did, but now you got rid of the cluttered pile of files.

This is only the most basic purpose of **jar**, however. Its uses extend well beyond merely concatenating and compressing collections of `.class` files.

5.11.1 Deploying Applications

One of the best uses of **jar** is to package applications for distribution. You can put a large Java application into a single file with **jar**, and by using a *manifest* (which we are about to discuss) you can nominate the main class to run in that JAR file. You can then provide a shell script (and a batch file, if you are also deploying to Microsoft Windows) that will set the `CLASSPATH` to point to the

JAR file and run **java** against it. With this simple setup, users need not even know they are using a Java application—it runs like any other application.

5.11.1.1 The Manifest File

The only way in which **jar** really differs from any other ZIP archive utility is in the automatic creation and use of a manifest file, by default named `META-INF/MANIFEST` in the archive. Even if you do not specify a manifest file of your own, the **jar** utility creates one for you. Let's take a moment to look at what goes into the manifest.

A manifest is basically a list of key/value pairs. The key comes at the start of a line and the value comes at the end of the line with a colon separating the two. Example 5.21 shows a sample manifest.

Example 5.21 Manifest from the Payback sample application

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.5.3
Created-By: 1.4.1_02-b06 (Sun Microsystems Inc.)
Version: 1.0
Main-Class: net.multitool.Payback.Payback
```

All of these entries were produced automatically by **ant** or the **jar** utility itself, except for `Main-Class`, which we specified (albeit with **ant**, as you will see in Chapter 9). The manifest has certain values that are always filled in by **jar**, but two that you might commonly specify are

- `Main-Class`, which allows users to run a Java application by merely typing `java someJarFile.jar`, without having to know the fully package qualified name of the class that contains the application's `main()`.
- `Class-Path`, which allows you to specify what the classpath should be when the application is run.

There are keys specific to applets, to signed applications, to beans, and so forth. We will address these as it becomes necessary. Full details can, of course, be found in the Sun's documentation for **jar**.[27]

27. `http://java.sun.com/j2se/1.4.2/docs/tooldocs/solaris/jar.html`

5.11.1.2 Putting a Compiled Application in a JAR File

Let's assume we are going to manually put a Java application in a JAR file. We will want to specify the name of the class that contains the `main()` method of the application. First off, you want the JAR's directory hierarchy to begin at the folder that contains the first node of each package's name. Our sample application here is in the package `net.multitool.Payback`, so we want our present working directory to be the one which contains the `net` subdirectory. Here's a dump of the directory tree from that point after compilation of our sample application:

```
$ find . -print
.
./net
./net/multitool
./net/multitool/Payback
./net/multitool/Payback/Account.class
./net/multitool/Payback/Purchase.class
./net/multitool/Payback/Cost.class
./net/multitool/Payback/DebtAccount.class
./net/multitool/Payback/Payback.class
./net/multitool/Payback/SavingsAccount.class
./net/multitool/util
./net/multitool/util/SAMoney.class
./net/multitool/util/SAMoneyTest$1.class
./net/multitool/util/SAMoneyTest$2.class
./net/multitool/util/SAMoneyTest.class
$
```

We now want to specify which class contains the application's `main()` method. It happens to be the `Payback` class, so we create a file called `manifest`[28] with the following contents:

```
$ cat manifest
Main-Class: net.multitool.Payback.Payback
```

Next, we use the **jar** utility to create the JAR file:

28. It can have any name. The key/value pairs from the file will be placed by the **jar** utility into the standard manifest called `META-INF/MANIFEST.MF` no matter what name you give to *this* file.

```
$ jar cmf manifest payback.jar net
$ ls -la
total 20
drwxrwxr-x    3 mschwarz mschwarz     4096 Aug  4 18:19 .
drwxrwxr-x    7 mschwarz mschwarz     4096 Aug  4 17:57 ..
-rw-rw-r--    1 mschwarz mschwarz       43 Aug  4 18:17 manifest
drwxrwxr-x    3 mschwarz mschwarz     4096 Jul 28 16:16 net
-rw-rw-r--    1 mschwarz mschwarz     7506 Aug  4 18:21 payback.jar
```

The options to **jar** tell it what to do. In our case, `-c` instructs to create a JAR file, `-m` adds the contents of the file named in the next parameter to the `META-INF/MANIFEST` file, `-f` and the next parameter is the filename of the JAR file being created. If we had not specified `-f`, the JAR file would have been written to standard `out` and an I/O redirect would be needed, but the result would have been the same:

```
$ jar cvm manifest net > payback.jar
$ ls -la
total 24
drwxrwxr-x    3 mschwarz mschwarz     4096 Aug  4 18:24 .
drwxrwxr-x    7 mschwarz mschwarz     4096 Aug  4 17:57 ..
-rw-rw-r--    1 mschwarz mschwarz       43 Aug  4 18:17 manifest
drwxrwxr-x    3 mschwarz mschwarz     4096 Jul 28 16:16 net
-rw-rw-r--    1 mschwarz mschwarz     7506 Aug  4 18:27 payback.jar
```

Everything that follows parameters required by option letters is considered to be a file or directory that is to be added to the JAR file. The option syntax for **jar** is similar to that for **pkzip** in the DOS/Windows world and the **tar** utility in the UNIX world.

As elsewhere in this chapter, we are just getting you started. See Sun's documentation for details.

5.11.2 Basic jar Operation

We have already covered the most common case, using **jar** to create a "rolled-up" Java application. **jar** has many command options besides `-c` and we'll document a few of them.

`-c`
: Create a JAR file.

`-u`
: Update a JAR file—replace updated files, add missing files.

`-x`
: Extract files from a JAR file.

`-t`
: List files in a JAR.

`-f`
: Specify the JAR filename.

`-v`
: Be verbose—display descriptions of what the **jar** utility is doing as it does it.

`-m`
: Add the contents of the named file to the manifest.

5.12 THE REST OF THE TOOLKIT

There are additional utilities in the toolkit. Below is a capsule summary of them and their purposes.

extcheck
: This utility checks an applet JAR file for any extensions it requires that are in conflict with the current Java runtime environment. (But we don't cover applets in this book.)

jarsigner
: This is a utility for digitally signing JAR file. Once a JAR is signed, anyone looking at it can be sure of two things—first, the file was definitely prepared by the owner of the private key that matches the public key used to verify the signature;[29] and second, the JAR file has not been modified in any way since it was signed. In other words, depending upon the care with which the signer treats his/her private key, this certifies the authenticity

29. A discussion of public/private keys as an authentication mechanism is beyond our scope here. As a shameless plug, Mr. Schwarz would like to point you to Chapter 10 of his previous book, *Multitool Linux*, which contains a beginner's introduction to public key authentication using GnuPG. Of course, a quick Google search will find you many online descriptions that are free.

and accuracy of the JAR file; you can trust it as much as you trust the signer.

Generally speaking, an applet must be signed by a key that the user trusts in order for the applet to increase its security access to the client machine (open/read/write files, access the printer, and so on). But we don't cover applets in this book.

keytool
: This tool is used to create and manage the keys we were talking about in the previous entry. Again, this is used mainly for JAR signing, and JAR signing is used mainly for applet authentication, but we don't cover applets in this book.

rmid
: The RMI activation daemon. In the RMI example we showed you in this chapter, you had to start a server manually and leave it running to process requests. *RMI activation* is a system where RMI servers may be started on demand.

serialver
: Reports the RMI `serialVersionUID` of a class. This can be useful when trying to track down problems with complex RMI systems, especially when multiple versions of an RMI interface with the same name must be in use at the same time.

native2ascii
: As mentioned before, Java uses Unicode for all strings. Most of the time, you and we are working with ASCII-encoded files. This program converts files from one format to the other.

policytool
: Remember how we had to set up a security policy file for our RMI demonstration (Example 5.13)? Well, this is a Java GUI application for creating and editing Java policy files.

There are a few more. One group of utilities is related to Kerberos tickets. Another is related to Java IDL and RMI-IIOP, both of which are to allow Java to interoperate with CORBA. If you are in a CORBA environment, you will want to look at those. We've got enough on our plates without trying to jam CORBA in here. You're on your own, buddy. Finally, there are a couple of

programs that support the Java plug-in, which is a way to make Java available in browsers.[30]

5.13 Review

For all of its humble command-line interface, the Sun Java 2 SDK provides a complete software development package, with everything you need to write and run a wide gamut of Java applications.

5.14 What You Still Don't Know

Oh dear, where to start? There are all those programs in the kit we glossed over right at the end there. We barely scratched the surface of JNI and RMI. We positively orbited the topics of policy files and JAR manifests.

You now have exposure to the bulk of the Java SDK. It is only the first step up the learning curve. We'll take you up a few more as the book progresses, but mastery comes only with time and experience. So use the tools. Write, compile, and debug a lot of Java code.

5.15 Resources

One of the best sources of information on Java is Sun's Web site for Java, `http://java.sun.com`, where you can find a wide variety of information—tutorials, white papers, API specifications, and more.

30. Those browsers that had Java built into them generally used Java 1.0 or 1.1. After Java 1.1 it became clear that it would be a major pain to have to reintegrate each new version of Java into each browser product. It would make browser versions and Java releases dependent on one another and would, in general, be a pain both for Sun and for Microsoft and Netscape (remember Netscape?). Thus, the plug-in architecture was born (okay, plug-ins were for things like RealPlayer and Macromedia Flash too). Since Java 1.2, browser integration has been via plug-ins.

Chapter 6

The IBM Developer Kit for Linux, Java 2 Technology Edition

This chapter introduces the IBM Java Software Development Kit. It does so mainly by pointing out how completely the kit mimics the Sun Java Software Development Kit covered in Chapter 5. Some time is spent on the minor differences and some installation issues.

6.1 WHAT YOU WILL LEARN

- The small number of important differences between the IBM and Sun Java Software Development Kits.
- How to put more than one version of Java on the same development machine and how to switch between them painlessly.

6.2 USE LINUX FEATURES TO MAKE MULTIPLE JAVA SDKS PLAY NICELY TOGETHER

We did not spend much time discussing the installation of Java on your Linux system in previous chapter. In fact, we did not discuss it at all. This is because the installation instructions that come with the SDK are more than adequate if you wish only to install a single Java SDK. But what if you want to install, say, both the Sun Java SDK and the IBM Java SDK? Then things get a bit more interesting.

We're going to review the concept of filesystem links, and how they work on Linux. While it may seem odd to discuss them here, we'll be using links to switch gracefully between different Java installations. Links are a powerful feature in Linux filesystems, and switching SDKs is a good application thereof. If you're already familiar with links, skip ahead to Section 6.2.2.

6.2.1 Links

A link is simply a name attached to a collection of data—that is, to a file. In other words, every file has one set of data and at least one link (a name). But a file may have more than one link. In other words, two different filenames may point to the same data. When you do this, you appear to have two copies of the file, but a change to one also changes the other. Deleting one, however, does not delete the other. It merely deletes the link. The file itself is only deleted when the last link is gone.

6.2.1.1 Hard Links

Links come in two flavors: hard and symbolic. A hard link looks like a file in and of itself. Let's show you a hard link, before and after.

```
$ ls -la
total 12
drwxrwxr-x    2 mschwarz mschwarz     4096 Jul  8 10:11 .
drwx------   50 mschwarz mschwarz     4096 Jul  8 10:11 ..
-rw-rw-r--    1 mschwarz mschwarz       45 Jul  8 10:11 sample
$ cat sample
This is a sample
file to demonstrate
links.

$
```

As you can see, we have a directory with a single file in it, `sample`. Now let's make a hard link to that file.

```
$ ln sample example
$ ls -la
total 16
drwxrwxr-x    2 mschwarz mschwarz     4096 Jul  8 10:13 .
drwx------   50 mschwarz mschwarz     4096 Jul  8 10:11 ..
-rw-rw-r--    2 mschwarz mschwarz       45 Jul  8 10:11 example
-rw-rw-r--    2 mschwarz mschwarz       45 Jul  8 10:11 sample
$ cat example
This is a sample
file to demonstrate
links.

$
```

Notice a few things here. First, other than the size and timestamps being the same, there is nothing obvious to show that these two files are, in fact, the same file. Note also the number just ahead of the owning user and group names. In the first directory listing, `sample` had `1` in that position; now both `sample` and `example` have `2`. This number is the *link count*. It tells you how many names are linked to the data associated with this name.

We have a couple more things to point out before we move on to soft links, which are going to be more important for our purposes.

```
$ chgrp wwwdev example
$ ls -la
total 16
drwxrwxr-x    2 mschwarz mschwarz     4096 Jul  8 10:13 .
drwx------   50 mschwarz mschwarz     4096 Jul  8 10:11 ..
-rw-rw-r--    2 mschwarz wwwdev         45 Jul  8 10:11 example
-rw-rw-r--    2 mschwarz wwwdev         45 Jul  8 10:11 sample
$ chmod o-r example
$ ls -la
total 16
drwxrwxr-x    2 mschwarz mschwarz     4096 Jul  8 10:13 .
drwx------   50 mschwarz mschwarz     4096 Jul  8 10:11 ..
-rw-rw----    2 mschwarz wwwdev         45 Jul  8 10:11 example
-rw-rw----    2 mschwarz wwwdev         45 Jul  8 10:11 sample
$ chgrp mschwarz sample
$ ls -la
total 16
```

```
drwxrwxr-x    2 mschwarz mschwarz     4096 Jul  8 10:13 .
drwx------   50 mschwarz mschwarz     4096 Jul  8 10:11 ..
-rw-rw----    2 mschwarz mschwarz       45 Jul  8 10:11 example
-rw-rw----    2 mschwarz mschwarz       45 Jul  8 10:11 sample
$
```

As you can see, a file can have only one set of owners and permissions, no matter how many links are made to it. Changing the owner or permissions of one link changes all hard links at the same time. In other words, the security of a file is like its data: A change to one link is a change to them all.

A link need not be in the same directory as the original name.

```
$ ln example /tmp/sample
$ ls -la
total 16
drwxrwxr-x    2 mschwarz mschwarz     4096 Jul  8 10:13 .
drwx------   50 mschwarz mschwarz     4096 Jul  8 10:11 ..
-rw-rw----    3 mschwarz mschwarz       45 Jul  8 10:11 example
-rw-rw----    3 mschwarz mschwarz       45 Jul  8 10:11 sample
$ ls -la /tmp
total 132
drwxrwxr-x    2 mschwarz mschwarz     4096 Jul  8 10:23 .
drwx------   50 mschwarz mschwarz     4096 Jul  8 10:11 ..
-rw-rw-r--    1 mschwarz mschwarz   118081 Jun  3 18:51 jLin.tar.gz
-rw-rw----    3 mschwarz mschwarz       45 Jul  8 10:11 sample
$
```

Here we made a third link in a different directory; `/tmp/sample` is a third name for the same data file. Note that we made it from the `example` link, not the original filename. In fact, as far as the Linux filesystem is concerned, there is no "original" name. None of these names is more significant than any other. When you remove a filename, the link is destroyed and the file's link count is decremented. If the link count goes to zero, the file is removed. That's it. Nothing else.

Hard links have a couple of drawbacks. One of them is a genuine technical limitation and the other is more of a usability problem. The technical limitation is that a hard link cannot be made across mounted filesystems. In the simplest

case (we don't want to muddy the waters with LVM[1] or RAID[2] at this point—most Linux distributions do not do LVM or RAID "out-of-the-box"), if you have more than one partition or disk drive, these are "mounted" at different points on the directory tree. For example, Mr. Schwarz's laptop's mount table looks like this:

```
$ mount
/dev/hda2 on / type ext3 (rw)
none on /proc type proc (rw)
usbdevfs on /proc/bus/usb type usbdevfs (rw)
/dev/hda1 on /boot type ext3 (rw)
none on /dev/pts type devpts (rw,gid=5,mode=620)
none on /dev/shm type tmpfs (rw)
$
```

We have one large partition mounted at `/`, or root, and a small partition mounted at `/boot`. In all of our hard link examples so far, we have been making links on the root filesystem. Example 6.1 shows what happens when an attempt is made to hardlink between two different mounted devices.

This is what we mean when we say a link cannot cross filesystems.[3]

The other problem is more "touchy-feely." With a hard link, you can see by the link count that other links exist, but you don't know where they are. Symbolic links get you around both of these issues.

6.2.1.2 Symbolic Links, or Symlinks

In a sense, symbolic links are much simpler than hard links. A symbolic link is a file that contains the name of another file or directory. Because it is marked as a symbolic link, the system will replace it with the contents of the linked file. Example 6.2 will make this more clear.

---

1. Logical Volume Manager. This is a tool that lets you arbitrarily aggregate disk drives and partitions into a "logical volume" that may be mounted and unmounted as a unit. Such tools are commonly used in serious production servers, but are rare on workstations or simple Linux servers.

2. Redundant Array of Inexpensive Disks. Another heavy server feature that allows multiple disk drives to be linked up as if they were a single disk drive and to act as backup to one another silently and transparently.

3. In a similar vein, there are some networked filesystems that do not support hard links at all because the server or host system doesn't support the concept. Attempts to make links on or to such systems will also fail.

Example 6.1 Attempt to hardlink between mounts

```
$ ln example /boot/sample
ln: creating hard link `/boot/sample' to `example': Invalid cross-device link
$
```

Example 6.2 Symlinking `/etc/passwd`

```
$ ls -la
total 8
drwxrwxr-x    2 mschwarz mschwarz     4096 Jul  8 15:30 .
drwx------   50 mschwarz mschwarz     4096 Jul  8 15:29 ..
$ ln -sf /etc/passwd passwd
$ ls -la
total 8
drwxrwxr-x    2 mschwarz mschwarz     4096 Jul  8 15:31 .
drwx------   50 mschwarz mschwarz     4096 Jul  8 15:29 ..
lrwxrwxrwx    1 mschwarz mschwarz       11 Jul  8 15:31 passwd -> /etc/passwd
$ cat passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
...
etc.
```

What we did here works just like a hard link, but note the attributes on the file: The `l` indicates a symbolic link, and the permissions are read/write/execute for user, group, and other (or for "world" for short). However, these permissions apply to the link, not to the file. Just as with hard links, there is only one set of permissions on the file, and these are on the file that is pointed to, not on the pointer.

One interesting difference between symlinks and hard links is that symlinks do not increment the link count on a file. If we remove `/etc/password` (a very bad idea, by the way), the symlink would be unaffected, but an attempt to open or read the symlink would not work, because it points at nothing that exists. This is called a *dangling symlink*.

Symlinks may refer to symlinks, and thus you need to be cautious to avoid circular symlink chains. All the shells we have used on Linux report circular symlinks as "too many levels of symbolic links," which sounds like a process exceeding its stack limits but handling it gracefully.

6.2.2 Switching Java Versions by Symlink

Here is the process we went through to install both Sun's Java 2 SDK and IBM's Java 2 SDK on a system at the same time.

1. Download the Sun JDK as a compressed TAR file.
2. Install it to `/usr/java`. The Sun installer named its directory `j2sdk1.4.1_02`, so the full path is `/usr/java/j2sdk1.4.1_02`.
3. Download the IBM JDK and untar it also in `/usr/java`. The base directory in the TAR file was `IBMJava2-141`, so the path is `/usr/java/IBMJava2-141`.
4. Create a symlink called `jdk` in `/usr/java` and make it point at the Sun JDK by default (Example 6.3).
5. Add `/usr/java/jdk/bin` to the system's default `PATH` environment variable.[4] Also add `JAVA_HOME` and `JDK_HOME` environment variables that point to `/usr/java/jdk`.

Now when we run **javac** or any other Java command, we run the version which is pointed to by the `jdk` symlink. If we wish to switch to the IBM JDK, we can just replace the link as show in Example 6.4.

From then on, the machine will be using the IBM Java SDK.

And, of course, by explicitly setting the path and environment variables, you can use whatever you prefer without changing the symlink for all other users on the system. This is an excellent example of how the features of the Linux system can make your life as a Java developer easier—with this general method, you can keep as many Java SDKs from as many vendors as you wish and switch between them at will.

Here, we showed you how to do this on a system-wide basis, but you could, by creating the symlink in your home directory and changing the path for your user account, switch between versions in your own account only, leaving the system-wide default alone. Linux provides endless flexibility for developers.

---

4. Exactly where you do this depends on your distribution. If you aren't sure, you can always do it in the `.bash_profile` file in your user account's home directory.

Example 6.3 Symlinking `jdk` to the Sun Java SDK

```
# ln -s j2sdk1.4.1_02 jdk
# ls -la
total 16
drwxr-xr-x    4 root     root        4096 Jul  8 15:51 .
drwxr-xr-x   17 root     root        4096 Jun 17 10:18 ..
drwxr-xr-x    8 root     root        4096 May 21 21:09 IBMJava2-141
drwxr-xr-x    8 root     root        4096 Mar  5 14:44 j2sdk1.4.1_02
lrwxrwxrwx    1 root     root          14 Jul  7 22:33 jdk -> j2sdk1.4.1_02
#
```

Example 6.4 Symlinking `jdk` to the IBM Java SDK

```
# rm jdk
# ln -s IBMJava2-141 jdk
# ls -la
total 16
drwxr-xr-x    4 root     root        4096 Jul  8 15:51 .
drwxr-xr-x   17 root     root        4096 Jun 17 10:18 ..
drwxr-xr-x    8 root     root        4096 May 21 21:09 IBMJava2-141
drwxr-xr-x    8 root     root        4096 Mar  5 14:44 j2sdk1.4.1_02
lrwxrwxrwx    1 root     root          14 Jul  7 22:33 jdk -> IBMJava2-141
#
```

6.3 How the IBM JDK Differs from the Sun JDK

After the last chapter, which was one of the longest in the book, this chapter should come as something of a relief. It is one of the shortest in the book. Why? Because the IBM Java Software Development Kit is practically identical in use to the Sun package. It differs in only a few respects and that is all we will talk about here.

One of the biggest differences is the version of Java available from each vendor. Sun has the newest versions, as they have been defining what those are. IBM is still releasing the 1.3 versions of Java as Sun begins to release 5.0. But you may not want or need the "bleeding edge" of the technology.

6.3.1 Performance

IBM's Java implementation appears to run most code faster than the Sun implementation. Benchmarking something as complex as a Java Virtual Machine is well beyond our scope here (and, in fact, coming up with a benchmark that will actually predict how much faster *your* application will run on one environment versus another is practically impossible). Nonetheless, we have seen some fairly dramatic performance improvements when running Java applications under the IBM JVM—improvements on the order of 50%–100%.

It is interesting to note that it does not matter which Java SDK produced the bytecode files. We see these improvements when the compiled classes are run, no matter which compiler (IBM's or Sun's) was used to produce them. This suggests that it is some combination of a faster virtual machine and/or a better Just-In-Time compiler (JIT) that gives IBM's runtime its apparent performance advantage.

For the most part, we use the Sun development kit and runtime, simply because Sun's is the definition of Java. But if execution speed is proving to be critical for your application, consider the IBM Java runtime. You may see some speed advantages.

6.3.2 Differences in the Commands

You will notice a few differences. For example, there is both a **java** and a **javaw**. Both invoke the Java runtime. The former has the Java console interface, the latter does not. For our purposes, this does not matter. The IBM Java SDK comes with an Object Request Broker Daemon (**orbd**) for CORBA/IIOP while the Sun SDK does not. Again, for our purposes this doesn't matter.

For the bulk of the utilities, the differences are so slight that you can use the Sun documentation for the IBM tools.

6.3.3 IBM Classes

IBM's Eclipse project (which we begin to cover in Chapter 10) provides a large GUI API library called SWT. We won't go into that here; it is covered in Chapter 17. Of more immediate interest is IBM's enhanced `BigDecimal` class (`com.ibm.math.BigDecimal`) which addresses a lot of deficiencies in Sun's implementation of decimal arithmetic. We will be using the standard Java class in our book (as it is the same for all development kits we cover), but you might want to take a look at IBM's FAQ document on their enhanced `BigDecimal`

class.[5] It also appears that IBM's class may become the official Sun version in Java 5.0 when it comes out. The primary feature of this class is its ability to deal correctly with rounding and precision, which is of great benefit in financial and scientific applications. Check out IBM's documentation and see if this is something you should use.

Note that Java bytecodes are Java bytecodes. You can download and use the IBM class with the Sun Java Runtime. It is there if you need it.

6.4 WHAT ARE ALL THESE "_G" VERSIONS?

One thing you will notice right away when you unpack the IBM Java SDK is that it has virtually all of the same commands as does the Sun Java SDK, but there is a whole bunch of them duplicated with a mysterious "_g" suffix. What's up with that?

These versions run a Java VM that was compiled with debug information, so that you can report information about bugs and errors that is of use to IBM SDK developers in locating and fixing problems. These versions should not be used for production work, but only to recreate and report bugs.

6.5 REVIEW

Well, we told you this one would be short. With a handful of minor exceptions, the IBM Java SDK is a complete drop-in replacement for the Sun Java SDK. You could go back to the previous chapter and repeat every example and exercise with the IBM Java SDK, and you would get the same results. There is definitely some comfort in knowing that even though you don't have an Open Source Java VM and SDK, at least you have two vendors the produce functionally identical development environments. You are not trapped into a single vendor's offering.

5. `http://www2.hursley.ibm.com/decimalj/decfaq.html`

6.6 WHAT YOU STILL DON'T KNOW

What you still don't know after reading this chapter is similar to what you still didn't know after reading Chapter 5. IBM Java SDK has many things we have not covered, including security policy files, JNI, and RMI.

6.7 RESOURCES

The best source of information about IBM's Java technology is IBM itself. Search the alphaWorks section of their Web site; we used `http://www.alphaworks.ibm.com/nav/java?openc=java+-+Developer+Kits` and found entries for the Java 1.3 Development Kit for Linux, as well as other Java-related downloads, including the Jikes Open Source compiler.

Chapter 7

The GNU Compiler for Java (gcj)

The GNU Compiler for Java provides a native binary compiler for Java code. In this chapter we'll show you how to compile a simple binary application from Java sources.

7.1 What You Will Learn

You will learn how to compile a binary executable from Java source code using the **gcj** compiler.

7.2 A Brand GNU Way

Quite some time ago Richard Stallman started an effort to create a free version of UNIX called GNU[1] (which stands for GNU's Not UNIX—a recursive

1. `http://www.gnu.org/`

acronym). More than that, he tried to convince the world that code should be Free with a capital "F". By this, he meant that it was unreasonable to provide software without both providing the source code and the right to use and modify that code as desired. To ensure this, he and his team created the GPL[2] (the GNU Public License) and founded the Free Software Foundation[3] to foster development and promote the idea.

The story of the founding of GNU/FSF and the motivations behind it[4] makes for a fascinating reading. Even if you are not interested in Free Software, the story prompts you to think in new ways about software, property, and freedom. As interesting as this story is, it is not our topic. The important thing is how the quest to create a Free operating system lead to a native Java compiler and the twists and turns on this way.

7.3 THE GNU COMPILER COLLECTION

If you are going to write a UNIX-like operating system, and one that is "Free" (certainly free of anyone else's intellectual property which might be restricted from the Free Software point of view), the first thing you need is a C compiler. Thus, a great deal of early effort by the FSF went into developing what was originally called the GNU C Compiler, or **gcc**.

Once they had a C compiler, some people began to write hundreds of utilities from **ls** to **grep**, while others began work on HURD, a microkernel for GNU. That work continues to this day. The bulk of the system commands you use on Linux were in fact developed by the FSF as part of the GNU project. This is why Stallman et al. want us all to refer to "GNU/Linux" rather than "Linux".[5] An understandable, if unenforceable, position.

It wasn't long before an effort began to integrate C++ into **gcc**. As time progressed, support for more and more languages and for more and more

2. `http://www.gnu.org/licenses/gpl.html`

3. `http://www.fsf.org/`

4. `http://www.gnu.org/gnu/thegnuproject.html`

5. A viewpoint we understand and appreciate, but we do not bow to is that we must *always* say "GNU/Linux." We say it sometimes, but it gets tedious and annoying if used all the time. So we compromise. We tell you about GNU, but we'll usually say just "Linux" in the text.

architectures[6] was being added. At some point, it was decided to rename (reacronym?) **gcc** to mean "GNU Compiler Collection."

Not too surprisingly, as Java emerged and gained popularity, it became one of the languages supported by the GCC using a front end called **gcj**.[7] That is what we'll be talking about here.

7.4 Compiling Our Simple Application with gcj

The basic form of **gcj** is

```
gcj [options...] [codefile...] [@listfile...] [libraryfile...]
```

We'll go over the options in a moment. For now, let's talk about the various kinds of input files the compiler can process.

In the above command-line synopsis, *codefile* refers to a Java source file, a compiled `.class` file (yes, **gcj** can convert already compiled Java bytecodes into native binaries), or even a ZIP or JAR file. A filename prefixed with the at-sign, `@`, indicates that the file contains a list of filenames to be compiled. That's the `@`*listfile* entry in the command synopsis. Finally, zero or more library files to link with may be specified on the command line. When you specify them directly (as opposed to using the `-l` command-line option) you must provide the full name of the library.

Like all the other Java compilers we have talked about so far, **gcj** supports the notion of a classpath. It will look in the classpath for unknown classes referenced by the classes you name to the compiler. Since **gcj** can read and compile from `.class` and `.jar` files, you might think you could just make sure that the JAR files from Sun or IBM Java SDK are on the **gcj** classpath and you would be able to compile any Java program using any Java APIs. Alas, you would be wrong. Why? Because the Java APIs are full of native methods, and which methods are implemented in Java and which are native is not documented anywhere.

---

6. A lot of people do not realize this, but **gcc** is a cross-compiler. Precompiled binaries do not always support this, but if you build your compiler from source, you can use **gcc** to compile code for any supported platform. For example, you can compile a program for a PowerPC-based Macintosh on your Intel-based PC.

7. `http://gcc.gnu.org/java/index.html`

Even if this were not so, it is not permissible under the GPL to distribute binaries without also offering to distribute source code. So, to distribute the Sun or IBM API JAR files would be incompatible with the GPL, and to not distribute them but to depend on them would mean shipping a product that doesn't work out of the box and requires users to obtain some non-Free software in order to work. That is just not acceptable. So the developers of **gcj** have opted to reimplement as much of the Java APIs as possible.

As you can probably guess if you have browsed the Java API Javadoc files, this is a monumental undertaking. The Java APIs are a moving target, and they started huge and grow larger with every new release. There is a parallel project to **gcj** called GNU Classpath[8] which is attempting to implement the entire Java API. Its target for the 1.0 release is to be fully compatible with Java 1.1 and "largely compatible" with Java 1.2. You might want to look at that project for better API support than that provided by **gcj**'s `libgcj`.[9] If you are curious about the present status of `libgcj`'s implementation of the Java APIs, there is a Web page (frequently updated) that compares the status of it against the Java 1.4 packages.[10]

7.4.1 Compiling FetchURL with gcj

We'll discuss **gcj**'s command-line switches in detail in Section 7.5, but we will have to use a couple of them here. First off, be aware that since **gcj** is actually part of **gcc**, all of the non-language-specific switches of that system also work in **gcj**; thus, `-o` specifies the name of the binary output file, and so on. There are many references on **gcc** to which you should refer for details (the manpage on **gcc** is a good place to start). Example 7.1 shows compiling and running `FetchURL` with **gcj**.

---

8. `http://www.gnu.org/software/classpath/`

9. The **gcj** and GNU Classpath projects are in the middle of an effort to merge their libraries into a common library. The GNU Classpath project aims to be a Free Software replacement for the JRE API JAR file. As such, it is meant to be a library of Java bytecodes that may be used as a drop-in replacement in any Java runtime environment. For our discussion, we will assume you are using `libgcj` as shipped with **gcj** itself.

10. `http://gcc.gnu.org/java/jdk14-libgcj.html`

> **TIP**
>
> The source code for `FetchURL` can be found in Example 3.30.

Example 7.1 Compiling and running `FetchURL` with **gcj**

```
$ gcj -o furl --main=FetchURL FetchURL.java
$ ./furl http://www.multitool.net/pubkey.html
http://www.multitool.net/pubkey.html:
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<HTML>
<HEAD>
<TITLE>Michael Schwarz's Public GPG key</TITLE>
</HEAD>
<BODY>
<CENTER>
<H1>Michael Schwarz's Public GPG Key</H1>
</CENTER>
<PRE>
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v1.0.7 (GNU/Linux)

mQGiBDuv6IQRBACn1TIWUXiEuZtfR+0Lqx6tYBAzIRpljL42O6r5nKHmndsWV71e
FUnhQpQIf+bNGGPMEt0g0vFpD6YWKP4uIEh2o+u1iyIIMs5QH3iqp8kFjbtVZa21
...
...
...
etc.
```

We already explained the `-o` switch which names the resulting binary. The other switch we use here is `--main` which specifies the class containing the `main()` that should be run when the binary is invoked. Remember that every Java class may contain a `main()`. In a multiclass program, the binary needs to know which `main()` to run when the binary is executed.

Remember that `FetchURL` is in the default package,[11] so you simply type the class name as the argument to `--main`. However, if the class is in a nondefault package, the fully qualified name must be used.

11. Any class without a `package` declaration is in the default package.

7.4.2 Compiling a Multiclass Program

For contrast, Example 7.2 shows compiling a multiclass program that is contained in a package (it is the Payback debt/savings/purchase calculator).[12]

Example 7.2 Compiling and running a multiclass program

```
$ cd payback/src
$ gcj -o payback -I. --main=net.multitool.Payback.Payback \
net/multitool/Payback/Payback.java
$ ./payback
Payback -- A savings/credit comparison tool
Copyright (C) 2003 by Carl Albing and Michael Schwarz
Released under the GNU/GPL.  Free Software.
...
...
...
etc.
```

The `-I` switch names a directory that is to be prepended to the classpath. In this case, we added "." which is the source directory for the Payback program.[13] Notice the package elements expressed with dots for the `--main` argument, and with slashes for the filename argument.

> **NOTE**
>
> The **gcj** compiler does pick up and use the `CLASSPATH` environment variable if it is specified. Also, **gcj** has a number of switches besides `-I` for classpath manipulation. We won't cover those here; `-I` is the preferred method (according to the **gcj** manpage at any rate).

12. Since this chapter was written, XML features were added to Payback that make it no longer work with **gcj**.

13. The Payback code can be found at the book's Web site: `http://www.javalinux-book.com/`.

7.5 OPTIONS AND SWITCHES

As we have said, **gcj** is part of the **gcc** suite of compilers and therefore supports all of the non-language-specific options and switches of that suite.

As with most reference material in this book, we will only cover the highlights. See the **gcj** manpage or the project's Web site for full details.[14]

`-I`*`dirname`*
: Add *`dirname`* to the classpath ahead of its existing contents.

`-D`*`name`*`[=`*`value`*`]`
: Add *`name`* and optional *`value`* to the system properties list. This is only valid with the `--main` switch.

`--main`
: Specifies which class contains the application's `main()`. This gives the starting point for an application.

`-fno-bounds-check`
: Disable array bounds checking. Like "real" Java, **gcj** checks all array operations to ensure that array bounds are not exceeded. Using this switch disables that check. It speeds up array operations but can introduce subtle and hard-to-find bugs. Use at your own risk.

`-fno-store-check`
: Like `-fno-bounds-check`, this disables a safety feature on arrays. Normally, when you store an object into an array, a check is made to make sure that the object is assignment-compatible with the array type (in other words, that the object is an `instanceof()` of the array type). Using this switch disables this test. It speeds up array operations but can introduce subtle and hard-to-find bugs. Use at your own risk.

There are other switches for native methods, bytecode (as opposed to native) compilation, and some switches related to resources. We leave it as an exercise for the reader to learn and use these where needed.

14. `http://gcc.gnu.org/java/`

7.6 REASONS TO USE GCJ

You might think that speed would be the primary reason to use **gcj**, but this is not necessarily the case. Yes, **gcj** is usually used as a native code compiler (it can compile to Java bytecode as well, and thus can be used as a replacement for **javac**), but there is a lot more to Java performance than that. First off, both Sun's and IBM's JVMs have JIT ("Just-In-Time") compilers in them, which convert some or all of a class's bytecode to native code on the fly. In some cases, these compilers may do a better job than the **gcj** compiler, so as a result, initial runs under a JVM are slower than **gcj** but later loops or iterations are comparable or faster. Also performance of both **gcj** and JVM code is highly affected by memory, stack, and garbage-collection parameters which may be modified with command-line options or properties files. So speed is not the determining factor. We have not done sufficient testing or measurement to tell you which environment produces "the fastest code" from a given source file. (We're not even sure exactly what such "sufficient testing" might consist of. All we can suggest is that your try *your* code in all three environments and then make your own choice.)

It is, perhaps, ironic that one of the main reasons why you might wish to use **gcj** is portability. You see, you can only run Sun's and IBM's JVMs on platforms for which they provide a compiled version. Linux runs on several hardware platforms (such as StrongARM) for which Sun and/or IBM do not provide JVMs. Also, if you are running Linux on some architectures, there may be VMs for the "official" OS, but none for Linux on that architecture. This is the case, for example, for SPARC and Alpha. The cross-compilation that **gcj** inherits from the GNU Compiler Collection allows you to compile Java to native code for Linux on those platforms.

Another reason to use **gcj** might be a desire for better integration with code compiled from other languages. **gcj** has JNI support, but also provides its own inter-language integration system called *CNI*, for Compiled Native Interface. We don't have space to cover CNI (and, frankly, we haven't used it enough to be good judges), but its proponents claim that it is both easier to use and more efficient than JNI. You can read up, use it, and judge that for yourself.

Still another reason might be one that we don't like very much. Again, it is ironic that the only Free Software Java compiler is the one best able to produce proprietary binary code. Code compiled with **gcj** is as difficult to reverse engineer as compiled C or C++ code. It is subject to the same sort of binary

obfuscation as other native compiled code. If you need to make your code closed and proprietary, **gcj** may be the right tool for you. Naturally, we aren't very fond of this idea, but it is still a reason one might choose the tool.

Finally, we mentioned that speed wasn't a certain factor for choosing **gcj**, but there is an exception. So far,[15] Java is particularly slow at starting and shutting down virtual machines. If you have a Java program that is invoked on demand or in a loop and the VM is started and stopped on each invocation, then **gcj** will give you a huge speed improvement, even if the code executes at the same speed or slightly slower than the JIT JVM code.

7.7 Reasons Not to Use gcj

We can think of three reasons not to use **gcj**. First, the compiled binary will run *only* on the target platform, whereas a Java bytecode binary is portable to any Java runtime without modification or recompilation. Second, **gcj** is not definitive. Sun still "owns" Java and only Sun's implementation can be presumed to be "correct." Third, the **gcj** API classes are not complete. If you visit the API status page we mentioned earlier, you can see what is provided and what is not. If **gcj** lacks an API your application requires, then you can be sure **gcj** is not the tool for you.

7.8 Review

The GNU Compiler for Java is part of the GNU Compiler Collection. It is generally used to compile Java source code into native binaries. It provides many of Sun's API classes, but not all.

7.9 What You Still Don't Know

You do not know how to interface with C/C++ code using **gcj**. You do not know how to use SWT from Eclipse to write GUI apps with **gcj**.

15. Sun claims that Java 5.0 will show considerable improvement in VM initialization speed.

7.10 RESOURCES

There are a number of resources for **gcj**, including

- The **gcj** home page.[16]
- The **gcj** FAQ.[17]
- The **gcj** documentation page.[18]
- The JDK1.4 to `libgcj` comparison page.[19] This resource is particularly useful in deciding whether **gcj** is an appropriate tool for compiling your program.
- Many features of **gcj** are, in fact, "inherited" from the parent project, the GNU Compiler Collection. You can find your way to a lot of good information from the GCC home page.[20]

16. `http://gcc.gnu.org/java/`

17. `http://gcc.gnu.org/java/faq.html`

18. `http://gcc.gnu.org/java/docs.html`

19. `http://gcc.gnu.org/java/jdk14-libgcj.html`

20. `http://gcc.gnu.org/`

Chapter 8

Know What You Have: CVS

Source control is such a necessary part of good development practice that it ranks right up there with a sound compiler as a critical part of any software project. It may seem like only an administrative overhead to newcomers, but its effect on a project of any size will be felt over time; it's not the first version of a project that needs source control so much as versions 2 and beyond. And it can be a life saver.

One of the Linux tools that is most appreciated on projects around the globe is the Concurrent Versioning System, CVS.[1] It is one of the best, most reliable pieces of software that these authors have ever used. It should be part of your repertoire of software skills, even when you're not running on Linux. But enough praise; back to practicalities.

1. As we were writing this chapter, the core developers of CVS released version 1.0 of a new version control system called Subversion. This new system supposedly contains many improvements over CVS. We do not doubt this, and we recommend that you take a look at Subversion before you select a version control tool. Meanwhile, we know CVS, and most Open Source projects are currently managed with CVS. Choosing CVS won't be a bad choice.

8.1 What You Will Learn

- Why you need CVS—the problem with source code.
- How CVS solves this problem.
- Some basic CVS mechanisms:
  - Importing source
  - Checkout
  - Commit
  - Tagging
  - Branch tagging
  - Status
  - Log
  - Export
- A quick look at a CVS GUI.

8.2 Source Control: Whys and Hows

Consider the following scenario: A customer has called with a problem in the software that your development team released over a month ago. Your developers try to reproduce the problem on their systems without success. What version of software is your team running? Well, there has been a lot of development in the last month, a lot has changed. Some new features have been added—halfway. In other words, it's close but not really the same software. And it's far from being ready to be given to the customer as a fix-release. Well, what's changed since the release was shipped six weeks ago? Can you find or create a set of sources that matches exactly what the customer is running? Can you then provide a modified version that contains only the fix necessary and no other changes?

With such low prices for hard drives these days it is now economically feasible to track your software releases simply by shelving an entire hard drive with each release of your software. It could contain the source code and all the tools in use for that version. But it does make search and comparisons a bit difficult. Still, conceptually, this is almost what you'd like—to be able to access an image of what your source looked like at any given point in time (for example, when released).

Enter **cvs**—the Concurrent Versioning System. It's a *versioning system*, allowing you to retrieve copies of the source based on either date parameters (e.g., last Tuesday) or the labels that you create. It's *concurrent* because it supports multiple simultaneous users.

You may have used a versioning system before that let multiple programmers work with a set of files. Often such systems will "lock" a file while one user is using it, keeping others from modifying it. CVS doesn't work that way—or doesn't have to. Rather it allows users to each modify the same file (truly concurrent), and then reconciles the changes when those changes are made permanent.

To explain all this, it would be best to set down some terminology, as used by CVS.

repository
: The master copy of the source.

sandbox
: A developer's local copy of the source.

checkout
: The process of acquiring a copy of the source (one or more pieces) from the repository.

commit
: The process of adding the changes from your sandbox into the repository.

update
: The process of revising your sandbox with changes that have occurred in the repository since you last updated or created your sandbox. When you "update" your sandbox, other developers' changes that have been committed to the repository are merged into your source sandbox.

tag
: As a noun, is a special label that you create to mark a milestone in your source repository; you can return to that milestone by checking out a copy of the source with that tag.

tag
: As a verb, refers to creating a tag in the source repository.

Once a repository has been set up for use by a project, each developer would check out a copy of the source. Thereafter, the typical sequence for a developer would be:

1. Edit.
2. Test.
3. Commit.
4. Go to step 1.

In some organizations, developers will commit and then test. Others will want to only commit changes that have been tested. Which order you choose is a policy decision by your project, not mandated by CVS.

> **TIP**
>
> We recommend that you test before committing because once you have committed your changes, they become available to all others developers. The more people are working together on a project, the more important it is to keep the source base workable, that is, clean compiling at least, so others can keep working.

Sometimes the developer needs to do an update step before a commit. Such a step is used to integrate other developers' changes into this developer's source. Sometimes this goes smoothly; other times it needs some additional work.

A simple scenario might help explain these steps, too.

Two developers, Ti and Kwan, are working on project Doh. They already have a repository set up with all the source for project Doh. Each developer, on his/her own system, checks out a copy of the source (`cvs checkout doh`). Now let's say that part of the source is a Java class file called `Account.java` and it has had several changes made already, so `Account.java` is now at version 1.7 in CVS.

Let's say that Ti finds a bug in `Account.java` and makes a change to fix that problem. Ti checks in (commits) the changes to `Account.java` (`cvs commit Account.java`) so that the repository now contains Ti's changes, which CVS keeps as version 1.8 of `Account.java`.

All this time Kwan has been busy modifying `Account.java` (e.g., adding a new method). Remember that Kwan is working from the 1.7 version. When

Kwan goes to commit his modified version of `Account.java` to the repository, he is notified that `Account.java` has been changed since his copy was checked out, and the commit attempt fails. So Kwan does an update which merges the 1.8 version of `Account.java` in with his modified 1.7 version. If all goes well, the resulting file will be a 1.8 version of `Account.java` which includes Kwan's new changes in the right place(s). Kwan just commits this to the repository, and `Account.java` then stands at version 1.9.

Note that cautionary phrase "if all goes well." The merge will work if Ti and Kwan have each modified different parts of the same file. If all Kwan did was add a new method, it would merge just fine. But what if they both make changes in the same region of the source file? It is up to the programmer to resolve such conflicts and commit the source once again.

In such a situation, CVS does what it can to help out. There is an example of a merge conflict later in this chapter. But such conflicts require human intervention.

Merging of conflicts is, undoubtedly, a very manual process, but you will be surprised by how infrequently you need to do this. Most changes will be merged clean with no manual intervention required. That's probably because most often, when two or more programmers are modifying the same file, they are modifying different sections of it.

With merging, you have the ability to incorporate other developer's changes into your version of the source without the fear of losing your changes. No one's changes get lost, no one's files get "stepped on."

8.2.1 Setup

Before you can use CVS to track your sources, you need to initialize a repository. You can use this repository for several different projects, so you only need to do this setup once.

There are two different ways to connect to a repository—directly on a filesystem, or indirectly over a network. We will use the simpler filesystem mechanism for this discussion. The network connections are described in the references at the end of this chapter.

In order for CVS to know where the repository is located and how to connect to it, it looks for an environment variable called `CVSROOT`. You can assign a value to `CVSROOT` from the command line each time you create a CVS project, or for more convenience, you can set it in the shell startup script (e.g., `.bashrc`) so that its ready all the time. The `CVSROOT` value is really only

used, though, to set up the project. Once a project is established, the information in CVSROOT is kept, along with other data, in a directory of files (called CVS). From that point on, CVSROOT (the environment variable) no longer needs to be set. The CVS commands will always use what is in the local sandbox to determine where the repository is; the value of the environment variable will be ignored.

It is possible to have different repositories for different projects. One repository might be for your personal work—revisions of memos and documents that you create on your local machine and store in a repository also on your local machine. Another repository might be a shared network-based repository, used for a project at work. Still another might be a network-based project for some Open Source work that you do in your spare time. Since the CVS repository keeps track of whence it comes, you needn't set a value for CVSROOT every time you switch projects. Instead, CVS knows from within the sandbox where to go for its updates, commits, and so on.

So let's get started and create a CVS repository on our local Linux system, in our own home directory. We will call the repository srcbank, as it will be the "bank" where we will deposit our source files.

```
$ mkdir ${HOME}/srcbank
$ export CVSROOOT="${HOME}/srcbank"
$ cvs init
```

The **mkdir** creates the directory named srcbank as a subdirectory of our home directory. The **export** command sets the shell variable CVSROOT to refer to the location of the new directory. The cvs init command initializes the repository with some needed directories and data files.

Before the cvs init command, the srcbank directory is empty. Afterward it contains a directory called CVSROOT (literal name, not the shell variable's value) which contains a variety of administrative files—most of which you need never worry about.

If your are using a remote repository, that is, one that you connect to over a network (typical when you are sharing a repository amongst team members), then you need one additional step—you need to log in to the CVS repository's server:

```
$ cvs login
```

which will prompt you for a password. Having logged in once, you will not need to log in again, even after reboots of your system, as CVS keeps the password (by default; it can be changed) in a file called `.cvspass` in your home directory. This makes using CVS with a remote repository (once you've logged in as simple as if the repository were local). From here on, the commands will all look the same. If your repository is remote, CVS will use the password from your `.cvspass` file, without asking you for it.

8.2.2 Import

Are you wanting to use CVS on an existing project? Have you already got your project at least partly underway? Let's look at how to enter all those files into CVS with a single command.

Not every file that is in a working directory needs to be kept under source control. Some, like `.class` files, are created from the `.java` source files. Others may be just scratch files that you don't want to keep versioned.

To automatically exclude certain files from ever being included in your repository, CVS uses a file called `.cvsignore` that lists filename patterns. Any filename matching a pattern will be ignored by all CVS commands.

Here is a `.cvsignore` file that we recommend for Java developers:

```
*.zip
*.class
```

This will exclude any file whose name ends in `.class` or `.zip`. Note that the comparison is strictly based on a name, not the actual contents. CVS doesn't know what a "class" file is; it is only excluding a file based on its name.

Certain files are not really source files and can't be managed as such, but we would still like to keep versions and a history of changes for them. A good example would be an image file. For example, you may have a corporate logo in a file called `logo.jpg` and at some point you may get a new or revised version of that file. You can use CVS to track such files, but you need to tell CVS that this is a binary file, so that CVS doesn't try to do some special substitutions that it does on check-in and check-out. (More about those substitutions later.)

For now, let's just consider how to tell CVS which files are binary. We can do that on the command line when we create a new file, but for importing a lot of files at once, and to avoid the need to remember doing that each time we add a file, we can put patterns for binary filenames in a CVS file called `.cvswrappers`.

Here is a `.cvswrappers` file that we recommend for Java developers:

```
#
# A recommended .cvswrappers file
#
# jar files - treat as binary:
*.jar -k 'b'
#
# Image file formats - treat as binary:
*.gif -k 'b'
*.jpg -k 'b'
*.png -k 'b'
*.tif -k 'b'
#
# Document file formats - treat as binary
# both MSOffice and OpenOffice.org file formats:
*.doc -k 'b'
*.ppt -k 'b'
*.xls -k 'b'
*.sx? -k 'b'
```

The format of the file is very UNIX-like. A leading `#` means that the rest of the line is a comment. The asterisk matches any number of any characters. The question mark matches a single character.

Now we're ready to import. The `.cvsignore` file should be placed in the topmost directory of the set of files that you want to import. Then, from that directory, issue the command:

```
$ cvs import Project YourCo import
```

where *`Project`* is whatever name you want to use for this project (or *module*) in CVS, and *`YourCo`* is the name of your company or some other designator to differentiate this source from other third-party packages that you may keep in your repository.

Most importantly, execute the `cvs import` command from within the directory, even though the name of the project is likely (but doesn't have to be) the same as the name of the directory in which you sit.

For example, consider a fragment of the filesystem shown in Figure 8.1. You would want to **cd** *into* the directory `coolj` and then issue the **import** command:

```
$ cd coolj
$ cvs import coolj GJSinc import
```

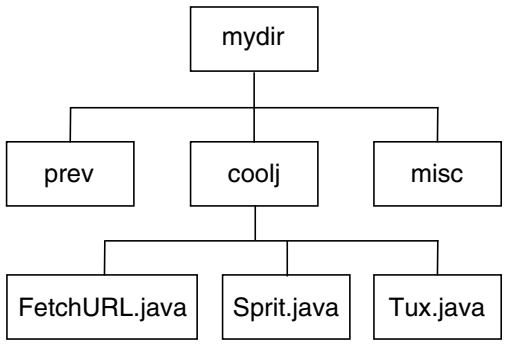

Figure 8.1 A sample directory structure prior to import

This will create a module named `coolj` in the repository, whose contents are all the directories and subdirectories that you see there. But you had to be *in* the `coolj` directory, which may seem counter-intuitive.

Now go to some other directory, one that is *not* part of the `coolj` part of the tree, and check out a copy of the source. For example:

```
$ cd
$ mkdir devsrc
$ cd devsrc
$ cvs checkout coolj
```

> **NOTE**
>
> It is important to check out the source after you've done the import, and *before* you make any changes, because the part of the filesystem that you imported remains untouched. It has no CVS knowledge, so you can't commit changes from that directory, unless you somehow make it CVS-aware. Since these files are your originals, until you've verified that the `cvs import` has gone as planned, it's best not to disturb those files. Create a new directory and check out the module there.

What do you see after the checkout? There should be a single directory, `coolj`, in the directory where you did the checkout (since it was empty when you started). That directory contains a copy of all the files that you checked in, along with a directory named `CVS` inside that directory and every subdirectory. The `CVS` directories contain administrative files that help CVS keep track of things for you, which means no CVS tracking information needs to be kept in

your source. You should never need to mess with the files in the `CVS` directory; see the Cederqvist reference in Section 8.6 for more information about these files.

8.2.3 Normal Use

The typical use of CVS occurs after you've made some changes to your source code. At some point, typically after the code compiles cleanly or after the changes have been tested to some extent, you will want to commit your changes to the CVS repository. When you commit one or more files, they become the latest version, the version that others get when they checkout or update the module. To say it another way, when you commit, you make those changes a permanent part of the source repository, available to others.

You can commit a single file at a time, like this:

```
$ cvs commit Account.java
```

Or you can commit several files at a time, like this:

```
$ cvs commit Account.java User.java Xyz.java
```

Or you can commit all the changes from a certain point in the filesystem hierarchy (e.g., the current directory) on down, like this:

```
$ cvs commit
```

(Specifying no files implies the current directory. You can also name a directory explicitly.)

When you commit changes, CVS wants you to provide a bit of commentary to explain what you've changed, to say something about this new version. The comment can be supplied on the command line, with the `-m` option:

```
$ cvs commit -m "bug fix"
```

If you don't provide the `-m` parameter and its argument, CVS will invoke your favorite editor (as specified in the environment variable `CVSEDITOR` or `VISUAL` or else `EDITOR`, in that order of precedence). The default, on Linux systems, is to invoke **vi** (see Figure 8.2). In the editor, you can type one or more lines of text; when you exit, the commit will continue to completion.

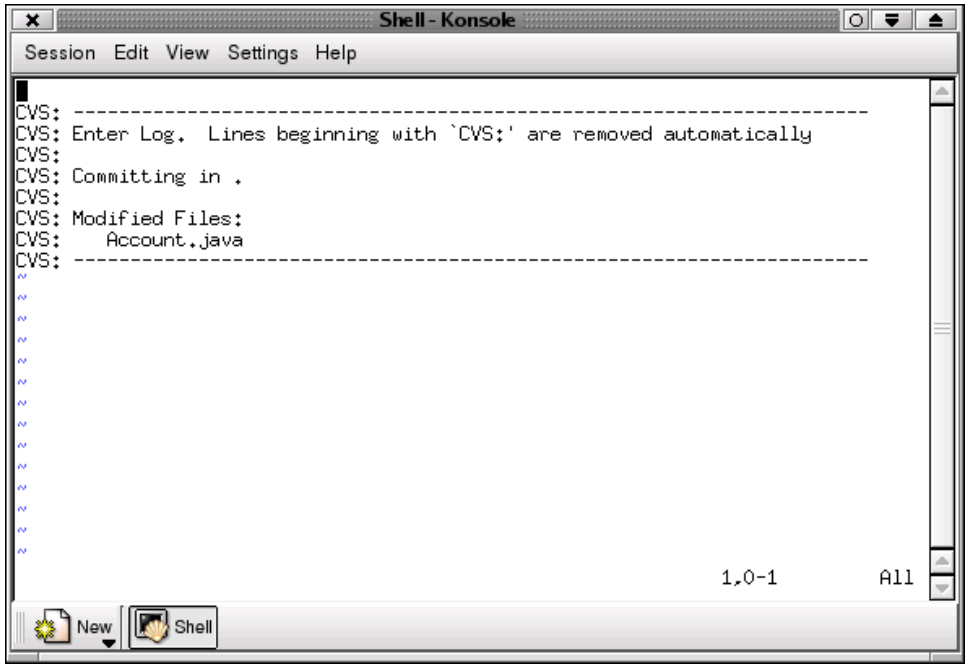

Figure 8.2 CVS asking for commentary as part of a commit

> **NOTE**
>
> If you quit the editor without writing your changes (in **vi**, that would be `:q!`) then CVS will ask if you want to abort the entire commit. If you choose to abort, no changes will be made to the repository. You'll be right back to where you were just before typing the `cvs commit` command.

You will be able to see the comments associated with each version of the file using the `cvs log` command (see Section 8.2.6).

As you will want to provide brief but meaningful descriptions in these comments, it may be helpful to remind yourself what in fact has changed. You can see the differences between the version that you checked out and the file as it stands today by using the `cvs diff` command:

```
$ cvs diff Account.java
```

Here, as in `commit`, you can name one or more files, or even a directory. CVS will display what lines you've added, modified, or removed in each file.

Example 8.1 Sample output from `cvs diff`

```
$ cvs diff Account.java
albing@cvs.multitool.net's password:
Index: Account.java
===================================================================
RCS file: /usr/lib/cvs/cvsroot/JavaAppDevLinux/majorApp/net/multitool/core/
Account.java,v
retrieving revision 1.10
diff -r1.10 Account.java
31d30
<       this.parent = null;
66a66
>     * returns an iterator
93c92
<       children.put(acct, name);
---
>       children.put(name, acct);
$
```

In Example 8.1, CVS has found three differences—one line being removed, one line being added, and one line being changed. The < precedes lines from the repository version, and the > precedes lines from the new, that is, changed, version. The `31d30` shows the line numbers from both versions, separated by a single character to indicate what difference action is being described: `a` for adding lines, `d` for deleting lines, and `c` for lines that change.

A typical work sequence might go something like this:

1. Edit some files.
2. `cvs diff` those files.
3. `cvs commit` those files.
4. Go to 1.

The `cvs diff` command is also quite useful for finding out what changed between some previous version of a file and the current version:

```
$ cvs diff -r 1.15 Account.java
```

or between two different previous versions:

```
$ cvs diff -r 1.12 -r 1.15 Account.java
```

or since a certain date:

```
$ cvs diff -D 06-Sep-03 Account.java
```

8.2.4 Update

If there are other people working on this project with you, they will also be making changes. To bring there changes into your sandbox, run the `cvs update` command:

```
$ cvs update
cvs server: Updating .
P Account.java
M User.java
cvs server: Updating subdir
```

Here, `P` indicates CVS has patched in changes to that source file; and `M` indicates you have modified the file. Note that `Xyz.java` is not mentioned. That means there were no updates involved.

The subdirectory `subdir` was also updated, but no changes were made. Had a change been made, you would see the modified files mentioned by name.

You can update a single file at a time by naming that file on the command line, but typically you want to get the changes for all the files in a directory, or even all the changes throughout the project, since a change in one file may be dependent on changes in other files.

Sometimes when you try to commit your changes you will be told that the commit did not succeed because one or more of your files was not up to date. Not to worry; it's easy to bring your files up to date. This leads directly into our next topic. Read on!

8.2.5 Merges

When you commit changes, a new version of each changed file is now part of the repository. If someone else commits changes, that person's changes are now part of the repository as well. But those changes (unlike your own local changes) are yet to appear in your own local copy of the files, that is your sandbox.

The following CVS command will bring your files up to date with all the changes made since you checked out your copy (or last did an update):

```
$ cvs update
```

With that command all the files from the current working directory on down will be updated with the most recent versions of the files from the repository—and not just updated: changes that you have made in your local files will be preserved and *merged* with the new version of the files.

Here's what a successful merge looks like:

```
$ cvs update Account.java
cvs server: Updating Account.java
M Account.java
RCS file: /usr/local/srcbank/JavaAppDevLinux/Account.java,v
retrieving revision 1.17
retrieving revision 1.18
Merging differences between 1.17 and 1.18 into Account.java
M Account.java
$
```

Remember our scenario earlier in the chapter? Our two programmers, Ti and Kwan, have each modified the same file. If all Kwan changed was adding a new method, it would merge just fine. But what if they both made changes in the same region of the source file? Well, the first one to check in his changes will be fine. His commit will succeed. But the second person to try to commit changes to the file will find that CVS will report an error:

```
$ cvs commit Account.java
cvs server: Up-to-date check failed for `Account.java'
cvs [server aborted]: correct above errors first!
cvs commit: saving log message in /tmp/cvsQ9rk01
```

Now, attempting to update will put *both* versions in your local file, marked up by certain character strings to highlight and separate the sections. It is up to the programmer to resolve those conflicts and commit the source once again.

Here's an example of how a conflict might look in a source file:

```
<<<<< ver. 1.7
for (i=0; i<20; i++) {
    myData.callSomething(dollars, time);
}
======
while (i<20) {
    myData.callOtherwise(dollars*(i++), time/60);
}
>>>>>
```

In such a case, the programmer must decide which changes to keep, or how to combine them. After editing the file and removing the dividing lines (i.e., `<<<<<`, `=====`, and `>>>>>`), recompiling and probably a bit of testing, too, the programmer can now do a `cvs commit` to incorporate his changes in the repository.

8.2.6 Log

With each `cvs commit` you are prompted for a comment, to describe the changes that you are committing. What happens to these comments? How can you see them again? Use the `cvs log` command to show the history of a file's revisions and associated comments.

See Example 8.2 for an example of the cvs output command.

Looking down the output of `cvs log`, you can see

- The complete filename—in the repository—of the file whose log we're checking out.
- The local filename in your sandbox.
- Which revision is the "head," that is, the front-most or default revision.
- Which branch, if any.
- What kind of locking mechanism CVS uses. There are some choices, but most users of CVS leave this as is.
- The access limitations. CVS can limit who can modify files (see our reference list if you need to use this).
- A list of all the tags (symbolic names) for this module and to which revision each refers.
- What kind of keyword substitution happens. For binary files this would be `kb`.
- The count of revisions for this file.

Then comes a description of each of the revisions, showing

- The revision number.
- Some stats on the change including the user ID of the user who committed the change.
- How many lines were added and deleted compared to the previous revision.

Example 8.2 An example of running the `cvs log` command

```
$ cvs log Account.java
RCS file: /usr/local/srcbank/JavaAppDevLinux/Account.java,v
Working file: Account.java
head: 1.4
branch:
locks: strict
access list:
symbolic names:
keyword substitution: kv
total revisions: 4; selected revisions: 4
description:
----------------------------
revision 1.4
date: 2003/05/20 11:59:59;  author: albing;  state: Exp;  lines: +80 -5
more comments added
----------------------------
revision 1.3
date: 2003/05/18 15:03:23;  author: albing;  state: Exp;  lines: +3 -2
end case fixed
----------------------------
revision 1.2
date: 2003/05/17 11:05:40;  author: albing;  state: Exp;  lines: +69 -2
actually runs - unit tested
----------------------------
revision 1.1
date: 2003/05/17 10:15:18;  author: albing;  state: Exp;
a rough structure
=============================================================================
```

- The comment that was entered when the user committed the change.

(For a description of state, and why you will almost always see `Exp;`, see the Cederqvist reference in Section 8.6.)

Do you want less output from `cvs log`? You can restrict the information to cover only a certain user's changes (`-w`), to a certain range of revisions (`-r`), and/or between certain dates (`-d`).

For example,

```
cvs -walbing -r1.2:1.4 -d05-Sep03 -d28-Sep-03 Account.java
```

will list only changes committed by user `albing`, only in the revision range of 1.2 through 1.4, and only between the dates of 05 and 28 September of 2003.

Note: do *not* put a space between the `-w`, `-r`, or `-d` and its parameter or CVS will think that the parameter is the name of a source module, and you will see a message like this:

```
$ cvs log -r 1.2:1.4 Account.java
cvs log: nothing known about 1.2:1.4
...
```

which will be followed by output about the `Account.java` module that CVS does know about.

For more variations on the logging output, type:

```
$ cvs log --help
```

8.2.7 `cvs status`

While the `cvs log` command will tell you about the history of all revisions of a file, you sometimes need to know the status of the current file in your sandbox: Which revision is it? From where did it come? And, most importantly, is it part of the head or part of a branch?

Those questions can be answered with the `cvs status` command. Its output will show the revision number of the file in your sandbox and any "sticky" tags. But to understand what that means, we need to talk about tags first.

8.2.8 `cvs tag`

We began this chapter asking: "Can you find or create a set of sources that matches exactly what your customer is running? Can you then provide a modified version that contains only the fix necessary and no other changes?" Part of the answers to these questions will depend on your use of the `cvs tag` command. With it, you can set down a label across all your source to mark a particular milestone, so that later you can recall that version of the source. For example,

```
$ cvs tag Rel_2_4
```

will put a tag (that is, a label) called `Rel_2_4` on the head revision of all source files from the directory where this command was executed on down through

all its subdirectories. If you run this command from the uppermost directory in your project sandbox, it will label your entire project.

A tag can be applied to a single file or group of files by listing them explicitly on the command line.

> **NOTE**
>
> Certain special characters are not allowed in CVS tags. Specifically, the characters `$,.:;@` are not allowed. So you can't use `release_2.4` as a tag. Too bad.

Tags cut across the various revisions of the source. While you *can* specify that a tag goes on the same revision of all sources (e.g., `cvs tag -r 1.3 one_dot_three_tag`), the more typical use is to tag different revisions of each module, the revisions that you've just been working with and testing.

Figure 8.3 shows a tag (`QA`) that cuts across the various revisions of the different sources. With such a tag, someone can check out a copy of the sources to get the `QA` release:

```
$ cvs co -r QA project
```

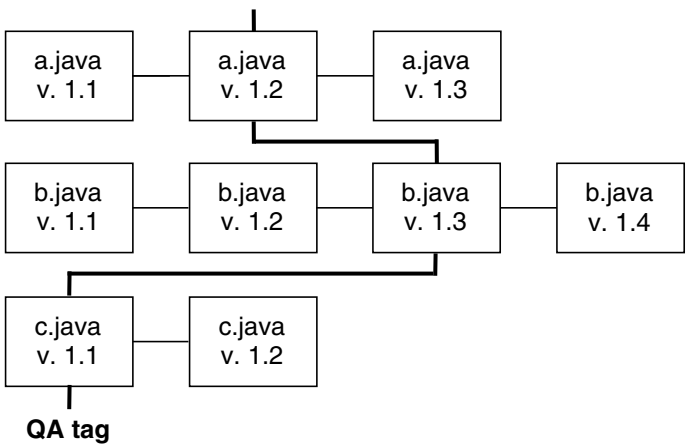

Figure 8.3 A tag across three files

Since your project would likely have more than one time in its life that it would be handed off to `QA`, some people will put date information in the tag, for example, `QA_2003_07_15`. Others will use a simple tag, such as `QA`, but such a tag may need to be reused.

If you've put down a tag and decide that you no longer want that tag (for example, your product is preparing for a new `QA` cycle and you want to reuse last cycle's tag, or maybe you simply misspelled your tag), you can delete it using the `-d` option.

> **WARNING**
>
> Once you delete a tag, it's gone forever. It is not available even when you recall earlier versions. If you reuse a deleted tag, it doesn't remember any history from its first use.

Imagine your project has just reached a milestone, like the hand-off to `QA`, so you have tagged your source accordingly. Now the `QA` group finds a bug and you fix it. What do you do with the tag? The tag will be on the unfixed version of source. One thing that you can do, *after* you commit your changes, is simply to move the label:

```
$ cvs commit Account.java
...
$ cvs tag -F QA Account.java
```

This will "force" the tag on `Account.java` to move to the current version.

Such a mechanism works fine for the short term, for quick changes that are put in shortly after the tag has been set down. But what if it takes `QA` several days to find the bug, and what if, during that time, you've been refactoring `Account.java`, or adding features for a future release? In those cases, what you really need is a branching tag.

8.2.9 Branching Tags

When you use the `-b` option with a `cvs tag` command, then the tag you create is a "branching" tag. That means that you now have two paths in your source repository. You can check out source from, and commit changes to, either of those paths. This allows you to keep moving ahead with new development on the head or tip of the source while providing fixes against a previous version.

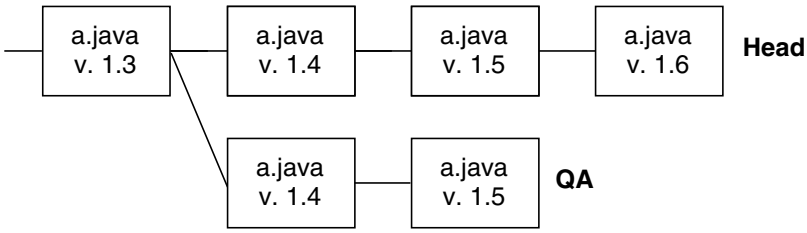

Figure 8.4 A simple branch and merge

Figure 8.4 shows a single source file with a single branch. The tag (`QA`) may have been applied to multiple files, typically to your entire project. The branched version of each file (for example, 1.3.1.1) is not created until the next change is checked in for that file, so many of the files with the tag may still be on their main source branch.

> **TIP**
>
> When do you want to create a branching tag? You can do it at any time that you lay down a tag. We have found it best to do it right away when you "release" your software, that is, whenever you hand it off to another group (e.g., `QA` or customers). This provides a label (the tag) to identify what exactly was handed off, but also puts the possibility for branching in place for fixes that may be needed on that branch.

Let's look briefly at the steps you would take to lay down a branching tag named `QA`, and then apply a fix to that branch.

In the directory where you have your current source, which is what you just released, set down the branching tag:

```
$ cvs tag -b QA
```

> **NOTE**
>
> You have just set down the branching label on the source but you have *not* changed your current set of sources. If you make changes in the current directory (and subdirectories) and check those changes in, you will be making those changes to the head, not the branch, of the source.

Example 8.3 Checking out a tagged revision

```
$ cd
$ mkdir fixes
$ cd fixes
$ cvs co -r QA myproject
cvs checkout: Updating myproject
...
U myproject/Account.java
U myproject/Caltron.java
U myproject/Demo.java
U myproject/Employee.java
U myproject/Person.java
...
$ cd myproject
$ cvs status Account.java
===================================================================
File: Account.java      Status: Up-to-date

   Working revision:    1.2     Sat Oct 26 03:32:17 2002
   Repository revision: 1.2     /usr/local/srctree/myproject/Account.java,v
   Sticky Tag:          QA (branch: 1.2.2)
   Sticky Date:         (none)
   Sticky Options:      (none)

$
```

Now that you have the label set down, you need to check out a copy of *that* version of the source. Since we are checking out a new copy, be sure that your `CVSROOT` environment variable is set appropriately (see above). Then find some new directory to put your source and check out a copy with the tag, as shown in Example 8.3.

We did a `cvs status` after the checkout to show you the important difference between this version and the other versions. These files will all show a `Sticky Tag` in their status. This is the label used to check out or update this version of the source. When you check in changes to these source files, the changes will be against that branch, and not the head.

From there on, everything is the same. Make your changes and just check files in as usual. CVS will remember (via the files in the `CVS` directory) that you're on the branch, so when you check things in, they'll go to the branch.

The important thing is to create the tag as a *branch* tag so that you can commit changes against that branch. The downside, however, is that you now

have two different source versions; bug fixes have to be made in both sources, new features have to be added twice, and so on.

The easiest way to deal with that is to keep your number of active branch tags small; you likely don't want to have to apply a fix to 14 different branches. Also, keep the lifespan of the branches brief—which is, of course, a relative term.

CVS does provide commands to merge a branch back into the source head. But for this, we will refer you to other CVS material. Our job is to give you an overview and a feel for the possibilities. For this sort of task you will want a complete reference manual.

For more variations on `cvs tag`, type:

```
$ cvs tag --help
```

8.2.10 `cvs export`

If you want to produce a copy of your source tree without the `CVS` subdirectories—just the pure source—you can use the `cvs export` command. Like the inverse of import, it will check out a copy of the source, but will not create any of the `CVS` subdirectories that allow CVS to manage the commits, checkouts, logging, status, tags, and so on. In other words, the exported directories are not a CVS sandbox—they're just a copy of the files.

> **NOTE**
>
> Changes made to an exported collection of files cannot be committed back to CVS. Of course you can get the changes back into CVS by creating a sandbox with a `cvs checkout` command, copying all or some of the exported files into that sandbox, and then committing the changes from there. But it's better to think of export as a one-way street.

8.2.11 A Quick Look behind the Scenes

If you are one of those people who worry excessively about efficiency, let us reassure you that CVS is OK. You could think of a CVS repository as saving each revision of a file (for example, versions 1.1, 1.2, and 1.3), but in fact CVS only keeps a single full version of a file—the latest version—and then stores the *deltas*, that is, changes required to revert back to the previous versions. So it keeps a full version of 1.3, but then only the differences between 1.3 and 1.2

and the differences between 1.2 and 1.1. This means that it is always very efficient to get the latest version of any file. (Other systems have tried keeping the original and the deltas for each revision going forward—but that gets very expensive to retrieve versions with hundreds of modifications. With CVS, the latest version is always at hand.)

An exception to this are "binary" files, those on which CVS can't do keyword substitutions. The revisions of those files, such as JPEG image files, won't be stored by deltas, but by complete copies of each revision.

8.3 A GUI: JCVS

If you are a die-hard GUI kind of developer, and aren't yet convinced of the power and convenience of the command line, then reread Section 1.3.10. If you are still not convinced, that's OK—you can still use CVS with the help of a GUI written entirely in Java. This is an implementation of the CVS client, that is, the portion of the CVS system that communicates with a remote server. The server does the real work of managing the versions; the client collects the data, manages the local files, and communicates with the server.

If you're going to use jCVS, you will need to get a CVS server up and running—or maybe your project administrator has already done that. If so, read on.

8.3.1 Installing jCVS

jCVS can be downloaded from `www.jcvs.org` where you can get it as a zipped archive file. Unzip it into a directory and create a shell script to make it easy to invoke. Since jCVS is an actual Java program, all the shell script needs to do is to ensure that its JAR files are on the `CLASSPATH` and then invoke the jCVS main class.

Here's a straightforward shell script which will accomplish that:

```
JCVS="/usr/local/jCVS-5.2.2"
CLASSPATH="${CLASSPATH}:${JCVS}/jars/activation.jar"
CLASSPATH="${CLASSPATH}:${JCVS}/jars/jcvsii.jar"
CLASSPATH="${CLASSPATH}:${JCVS}/jars/jh.jar"
CLASSPATH="${CLASSPATH}:${JCVS}/jars/js.jar"
java -jar ${JCVS}/jars/jcvsii.jar
```

You would need to change the definition of `JCVS` to match the directory where you unpacked the ZIP file, but the rest will work with your location. Of course the classpath could all be set on one longer line, but this way is more readable.

Run jCVS (Figure 8.5) and navigate to the checkout page (Figure 8.6) to fill in the parameters for your CVS server. Then you should be able to contact it for checking out your sources (Figure 8.7).

Figure 8.5 jCVS splash screen

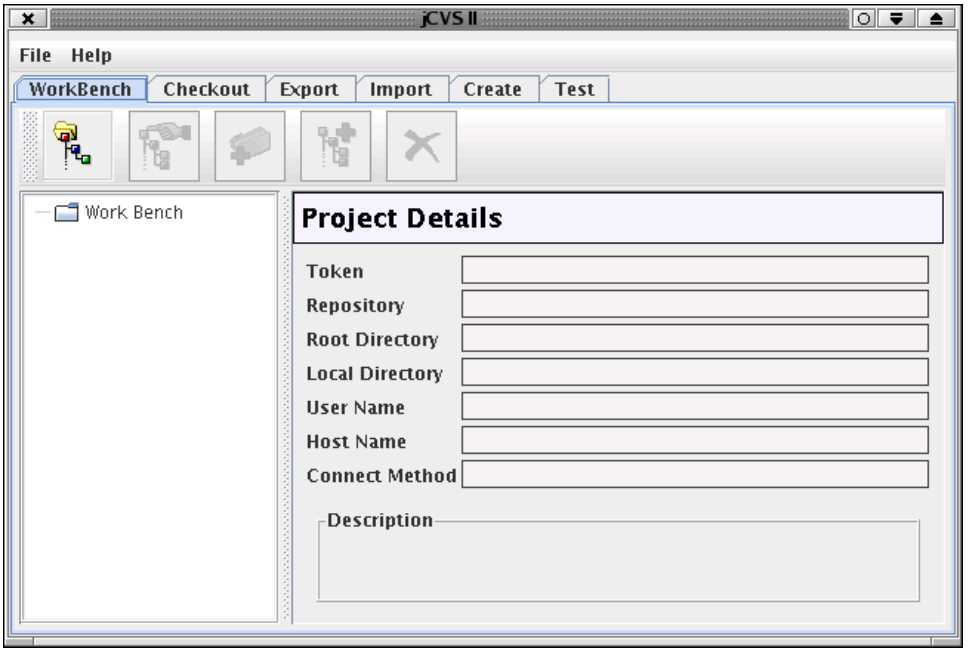

Figure 8.6 jCVS initial screen

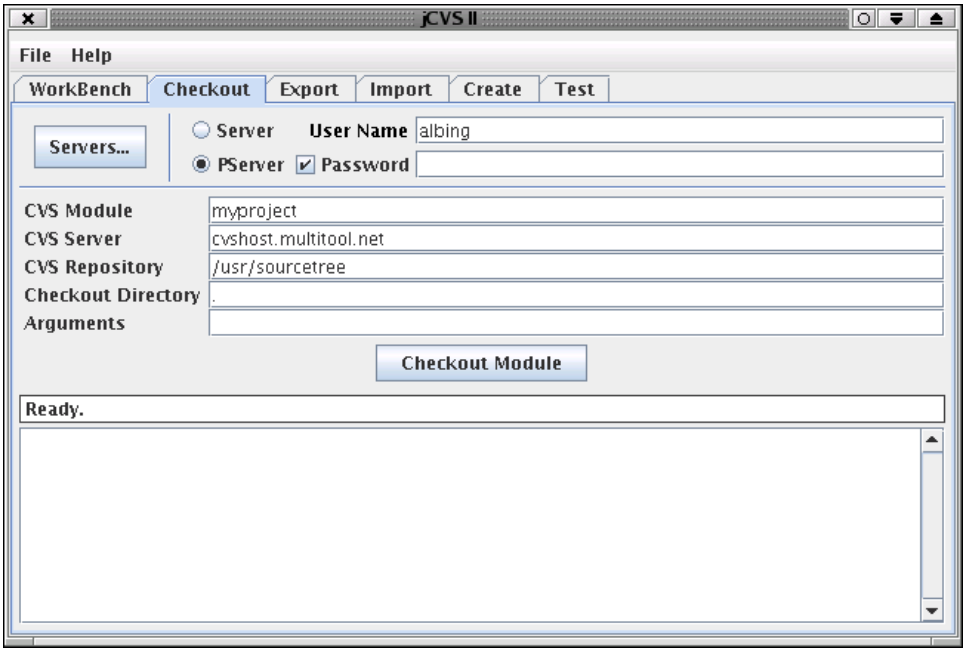

Figure 8.7 jCVS checkout screen

Conspicuous in its absence on the jCVS Web site is a good user manual. Since we're proponents of the command line, don't look for it here, either.

Many, if not most, Java developers these days will be using, at least part of the time, an Integrated Development Environment (IDE). One of the tools that these IDEs integrate is a source code manager, and typically for Open Source tools that means CVS. So while we're not giving you much info on jCVS, you will find a useful GUI for CVS inside most IDEs.

8.4 Review

This chapter has been all about CVS, one of the great jems of the Open Source world. Projects all across the globe depend on CVS to track their source changes, as programmers half a world away collaborate and share source.

We discussed how to import source files into CVS and how to get them back out. We discussed the mechanism for checking in changes and how to sort out collisions for the rare occasions when automatic merges don't succeed. We described how to tag a set of source files for later retrieval, and how to make

those tags into branches in your source tree. We also discussed how to show the history of those changes and the status of a source file. Finally, we took a quick look at a GUI for use with CVS, for those so inclined.

8.5 WHAT YOU STILL DON'T KNOW

- The myriad of administrative commands which help you manage and maintain a source library. Most of those commands you won't need, but it's nice to know that they are available, just in case. Refer to the Cederqvist document (Section 8.6) for all the gory details.
- How to set up remote users for sharing a CVS repository on a network, especially the use of the `CVS_RSH` environment variable.
- How CVS integrates into development tools. We'll see this in the coming chapters; for example, CVS interaction is built into NetBeans, SunONE Studio, and Eclipse.
- How good it feels to have CVS come to the rescue so you can recover a version of something that you thought was lost. May you never have to learn this the hard way.

8.6 RESOURCES

- *Version Management with CVS* by Per Cederqvist et al. is the "official" manual for CVS. It can be viewed as HTML or downloaded as HTML, PDF, or PostScript from `http://www.cvshome.org/docs/manual/`.
- Chapter 6 of *The LINUX Development Platform: Configuring, Using and Maintaining a Complete Programming Environment* by Rafeeq Rehman and Christopher Paul (ISBN 0130826758, Prentice Hall PTR) gives a good introduction to CVS. They give more information, too, on jCVS as well as on how to integrate CVS into Emacs.

Chapter 9

Ant: An Introduction

Ant[1] is a tool for building and deploying collections of files. It is particularly suited to building and deploying Java applications (in no small part because it is written in Java). Ant is well worth knowing and using for all but the simplest Java applications.

9.1 What You Will Learn

- How to install and set up Ant.
- Enough about XML to read an Ant buildfile.
- The basic tags used in most buildfiles.
- The extent of our remaining ignorance after completing the chapter.

1. I must mention something about the title of this chapter: There was a Monty Python's Flying Circus episode that had this title, and my inner geek couldn't resist.

9.2 THE NEED FOR A DIFFERENT BUILD TOOL

James Duncan Davidson had a problem. Perhaps you've had this problem, too. It has to do with the building of software—compiling, copying, and otherwise modifying files to get all the pieces in all the right places for running a collection of programs.

There are a number of ways to automate the building of software. You can script the compilation using shell scripts or batch jobs. That works fine, but there are two problems with that solution. First, scripts are generally not portable across very different operating systems. That's a serious problem for a language like Java, which is intended to be portable across operating systems. Second, it is difficult if not impossible, using scripting languages, to prevent wasted compiles; the checking and comparing of date stamps on source and object files makes scripts large and difficult to maintain.

Very well, we hear you say. There's **make**. The **make** program has been around for a long time. It is available on many operating systems. It handles the conditional compilation of files very well. It has been around for centuries (it seems). It is well known and widely used. All of this is true, but even this venerable tool falls a bit short in the Java world. First of all, although makefiles are generally far more portable than other scripts, there are still considerable variations in details, and **make** does nothing to mask the differences in file, path, and disk designations that exist across operating systems. Moreover, both **make** and scripts suffer from a more basic problem. Although Java programs can execute reasonably quickly, the overhead of starting a JVM and tearing it down again is considerable. Since **javac** is written in Java, each time it is invoked to compile a source file (one file at a time is the **make** way) this setup and teardown time cost is paid.

But, we once more hear you protest, you can just use **javac** on the entire project! Doesn't it build everything that needs to be built? In the simplest case, yes, it does. But as soon as you share code between projects, or use RMI which requires execution of the **rmic** compiler, or use EJBs, or link in native methods, then **javac**'s dependency resolution just will not work.

Luckily, James Duncan Davidson had this problem. And luckily it really bothered him. And even more luckily for us all, he decided to share his solution.

His solution was Ant, which we will from now on refer to as **ant**. Why **ant**? Well, he suggests that it might be because ants are little things[2] that build big things. He has also suggested (in his preface to the O'Reilly book *Ant: The Definitive Guide*, Jesse Tilly and Eric Burke) that it might stand for "Another Neato Tool." We're inclined to put forth the former, but believe the latter.

James Duncan Davidson wrote **ant** and contributed it to the Apache project, so it is Free Software. And it makes the problems cited above rather piffling. Through the rest of this chapter we will describe how.

9.3 Obtaining and Installing Ant

You can obtain **ant** from the Apache Web site.[3] Which version you download will depend on your system and your needs. There are stable releases and daily builds. Unless you have a compelling need for a feature not yet in a stable release, we would suggest you stick with the most recent stable release. As of this writing, that is version 1.5.1.

If you are using RedHat, or another Linux distribution that uses the RedHat Package Manager, **rpm**, then the simplest way to install would be to download the RPMs linked from the Web site and install those:

```
$ rpm -i ant-1.5.1-3jpp.noarch.rpm
```

You have two other options besides the trusty old RPM method. First, you may download a binary *tarball*, a word often used for a compressed file created with the **tar** utility, or you may download and compile the **ant** source code.

Let's take these each in turn.

9.3.1 Installing a Binary Tarball

Binary distributions of **ant** are available in `.zip`, `.tar.gz`, and `.tar.bz2` formats. Utilities are available for all of these formats for Linux, although you

2. Not so little anymore. As of this writing, the head of the CVS tree for **ant** weighs in at just shy of 48MB, and there are 5,239 files in there! These totals include a lot of project documentation, but even considering only the `src` subdirectory, we are still looking at 18MB and 1,687 files. It is probably incorrect to call **ant** a "little thing" these days.

3. `http://ant.apache.org/`

will find that generally `.zip` files are intended for Windows machines and `.tar.gz` and `.tar.bz2` for UNIX systems. The `.gz` format is decompressed with the **gzip** utility and `.bz2` files with the **bzip2** utility. The **bzip2** compression algorithm produces better compression, while **gzip** is "more common." If you have a modern Linux distribution, you almost certainly have both installed already.

Once you have the archive file downloaded, you should download one of the files linked next to it. These are cryptographic hashes of the legitimate archive file so you may be (more) assured that the software is the software you think it is. The first, PGP, is an actual digital signature. If you are already familiar with PGP or GnuPG and are comfortable with them, then by all means use this. It is, as you know, a superior form of validation compared to MD5. But explaining how to use digital signatures and GPG keys is beyond our scope here. As for MD5, however, this is fairly easy.

An MD5 hash is a 128-bit value generated in such a way that it is impossible for two different files of the same length to have the same hash value (actually, the term used in the literature is "computationally unfeasible," but for our purposes that is the same thing). If you run the program **md5sum** with the tarball file as an argument and you get the same number as the one you downloaded, you may be certain that the file you have is an exact match with the one that was used to produce the number you downloaded from the Web page. Remember that this is all that is proved by this. If both the file server *and* the Web page have been compromised, then the fact of a match doesn't mean much. A *mismatch* however proves that one of the two has been compromised and you probably shouldn't use the tarball.

You should get in the habit of verifying checksums and digital signatures where they are supported.

If you are still worried about the dual compromise, well, that's where a PGP digital signature can help. It not only proves the integrity of the data; it also proves the identity of the generator. Learn more about PGP (actually, the Free Software version of it, called GnuPG, at the GnuPG Web site.[4]

Once you have downloaded both the `md5` file and the tarball, validate and extract the tarball (Example 9.1).

Note that we did this in a regular user's home directory. If you just wish to use **ant** yourself, then this is the way to go. If you wish to make **ant** available

4. `http://www.gnupg.org/`

Example 9.1 Validating and extracting the **ant** tarball

```
$ cat jakarta-ant-1.5.1-bin.tar.gz.md5
2be27d9e09011bf1cc3d1967ee34f7d1

$ md5sum jakarta-ant-1.5.1-bin.tar.gz
2be27d9e09011bf1cc3d1967ee34f7d1  jakarta-ant-1.5.1-bin.tar.gz
$ zcat jakarta-ant-1.5.1-bin.tar.gz | tar xf -
$ cd jakarta-ant-1.5.1
$ ls
bin           docs         etc          KEYS            lib
LICENSE       LICENSE.dom  LICENSE.sax  LICENSE.xerces  README
welcome.html  WHATSNEW
$
```

to multiple (or all) users on the system, you will want to untar as `root` and move the resulting directories to locations convenient to other users, such as `/usr/local`.

Whether for one user or for many, there is a handful of remaining tasks to make **ant** usable.

9.3.1.1 Environment Variables

The `JAVA_HOME` environment variable should already be set as a result of setting up your JDK. `JAVA_HOME` should point at the base of your JDK installation.

The `ANT_HOME` environment variable should be set to point at the untared installation of **ant**. In our sample here, it would be `~/jakarta-ant-1.5.1`.

Make sure that the `bin` directory of the **ant** installation is added to your `PATH`.

9.3.2 Installing a Source Tarball

We do not encourage you to install **ant** from source, although we do encourage you to download and study the **ant** source. It is an excellent sample Java application.

If you must build from source, the start of the process is the same as above. You download the tarball, verify it with GPG or **md5sum**, then unzip and untar it.

It begins to differ at this point. The **ant** source package comes with a shell script, `build.sh`, that actually builds a minimal version of **ant** and then runs **ant** to complete the install.

Make sure that the `JAVA_HOME` and `ANT_HOME` are set as you want them, then execute `build.sh install`. Unless you have installed the optional tasks,[5] you will see several warnings about missing classes. You may safely ignore these.

As with the installation of other packages built from source, you will need to have appropriate permissions for the target directories. This might mean running the install as root, with all appropriate attention and dread.

9.4 A Sample Ant Buildfile

Let's go over the basics of creating an Ant buildfile. We'll start with an introduction to XML, and then move on to the specific tags Ant supports and how you might use them to automate a build.

9.4.1 XML for the Uninitiated

The buildfiles of **ant**, usually named `build.xml`, are written in Extensible Markup Language, or XML. Some of the reasons for this are:

- XML is hierarchical.
- XML is standardized.
- XML is widely used and familiar to many programmers.
- Java has many classes for reading, parsing, and using XML.
- XML-based representations of hierarchical data structures are easy to read and parse for both humans and programs.

XML is a successor to SGML, Standard Generalized Markup Language, which is a language for defining markup languages. A markup document may be *validated*. A validated document is one that conforms to a structural specification of the markup tags in the document. Such a specification may be made using a Document Type Definition (DTD), which is a holdover from the way SGML markup languages were specified, or using one of the newer specification

5. There is a horde of optional tasks. As the name suggests, they are optional. Include these if you need them. This is the only mention they will receive.

standards, such as W3C's XML Schema. In either case, the DTD or schema specify what tags may be used in the markup, where they may exist with respect to one another, what attributes tags may have, and how many times a given tag may appear in a given place. A document can thus be validated—that is, checked against the corresponding DTD or schema. It's not necessary, however; in many situations, documents can also be used without validation so long as they are *well-formed*—that is, conform to the basic syntax of XML.

HTML, with which even nonprogrammers are familiar, is an instance of a markup language defined in terms of SGML (and XHTML is its reformulation in terms of XML). This book itself was written in Docbook, which is another SGML markup language.

So, if SGML is such a wonder, why is XML all the rage? Well, SGML is one of those standards that attempt to "subsume the world." SGML has very complex and flexible syntax, with many different ways to represent a simple markup construct. Thus, to completely implement an SGML parser is difficult. Recognizing that 90% of the complexity of SGML is needed in only about 1% of cases, the designers of XML realized that they could make a markup specification language only 10% as complicated that would cover 99% of cases (of course, like 85% of statistics, we're making these numbers up, but you get the point).

Implementing an XML parser, while not exactly trivial, is much easier than implementing an SGML parser.

SGML/DSSSL and XML/XSLT are efforts to make the transformation and presentation of hierarchical data easier and more standardized. If what you have read here is all that you know about XML (or SGML), you should certainly consider getting yourself a book on these important standards.

For now, we can say that XML consists of *tags* which are set off from data content by the familiar less-than and greater-than brackets we are used to seeing in HTML:

```
<samplexmltag>
```

Just as in HTML, the tags may have start tag and end tag forms:

```
<samplexmltag>Sample XML tagged data</samplexmltag>
```

The entire construct, including the pair of matching tags and everything inside them, is called an *element*. The start tags may also, like in HTML, carry data inside them in the form of *attributes*:

```
<samplexmltag color="blue">Sample XML tagged data</samplexmltag>
```

If you have an empty element, one that that either does not or cannot have data between its start tag and end tag, you may "combine" the start and end tag by putting the slash at the *end* of the tag:

```
<samplexmltag color="blue"/>
```

Obviously, there is more to it than this, but it is enough to begin with.

XML's uses range from publishing to networked interprocess communications. Our interest here is in using it to represent a model of a piece of software and the various ways that software might be built and deployed. So from here on, we will be discussing not XML in general, but the **ant** document type. Actually, **ant**'s markup language uses unvalidated XML. In other words, there isn't officially a *schema* for **ant**. Thus, the only formal definition for an **ant** XML file is what **ant** accepts and understands. This is more common than it should be. Any XML markup vocabulary really should have a schema, but often XML use starts with "Oh, this is just a quick thing. No one will ever read or write this markup. Just these two programs of mine." These famous last words will one day be right up there with "I only changed one line of code!" As strongly as we feel about this, **ant** really can never have a DTD, at least not a complete one. The custom task feature makes this impossible.

9.4.2 The Buildfile Tags

The buildfile (usually named `build.xml`) begins with a header announcing that this is an XML document and specifying what version of XML is being used in it:

```
<?xml version="1.0"?>
```

The `<?` and `?>` delimiters mark up an XML *statement* (as opposed to an XML tag).[6] In this case, we are declaring that this is an XML document and that it is using XML version 1.0.

6. Note that these are the terms we are using to describe XML to a new user. They are not the formal terms for these document elements. For the proper names, consult an XML reference.

9.4.2.1 The `project` Tag

Every buildfile must begin with a `project` tag. A buildfile must contain exactly one `project` tag.

The `project` tag contains three attributes:

`name`
: The name of the project.

`default`
: The default target (see next section).

`basedir`
: The base directory of the project. Usually this is "." meaning the directory the buildfile is in.

The project name is just a name. It is not of particular importance to **ant**, although many IDEs that integrate **ant** will make use of the name attribute of the project.

The `default` attribute names a `target` tag that will be built by default if a build target is not passed to **ant** (see Section 9.4.2.2).

9.4.2.2 The `target` Tag

Every time you run **ant**, you are building a *target*. If you do not specify a target, **ant** will run the target named in the `default` attribute of the `project` tag.

A project may contain any number of targets. The `target` tag has five attributes:

`depends`
: The name or names of other targets that must be built before this target may be built.

`description`
: A descriptive name for the target. Displayed when `ant -projecthelp` is run.

`if`
: Specifies a property name (see Section 4.4). The target is only built if the named property is set.

`name`
: The name of the target. This is the name entered as an argument to **ant**. This is also the name that may be used in the `default` attribute of the `project` tag.

`unless`
: This is the reverse of the `if` attribute. The target is built unless the property is set.

9.4.2.3 Properties

There is more than one way to set what we might call variables in **ant**. The only one we will concern ourselves with here is *properties*. Properties are like a simple hash, or associative array. They associate value, which is stored as a `String`, with a name, which is also a `String`. They behave very much like the `Properties` class introduced earlier in this book.[7] You can use buildfile properties to associate a single name with a single value that you use in multiple places throughout the buildfile to make configuration changes easier and less error-prone. Some tasks also expect certain properties to be set, as we shall soon see.

You set a property with the `property` tag (Example 9.2).

Example 9.2 A useless `build.xml` example to demonstrate properties

```
<?xml version="1.0"?>

<project name="pointless" default="useless" basedir=".">

  <target name="useless">
    <property name="example.utility" value="nil"/>
      <echo>This example's usefulness:
      ${example.utility}.  OK?</echo>
  </target>
</project>
```

Running **ant** with Example 9.2 gives this output:

7. In fact, an examination of the **ant** source code reveals that **ant** properties are stored in a `HashTable`.

```
$ ant
Buildfile: build.xml

useless:
     [echo] This example's usefulness: nil.  OK?

BUILD SUCCESSFUL
Total time: 1 second
```

9.4.2.4 Tasks

A *task* is something that must be done to build the target. There is no single "task" tag; instead, each kind of task has its own tag[8] so there are many tags referred to collectively as *task tags*.

There are dozens of standard task tags, but only a few of them are "everyday." We'll introduce a few of them here, and then talk about the tags that don't fall into the project/target/task hierarchy.

Standard task attributes. All **ant** task tags have at least the three attributes:

`id`
: A unique ID for the task (not required).

`taskname`
: A name for the task, used in logging (not required).

`description`
: A description of the task, used for comments.

The `javac` task. The `javac` task, not surprisingly, runs the Java compiler. Note that since the Java compiler is written in Java and so is **ant**, there is no VM launch overhead to running the compiler. This can make **ant** many times faster than **make** simply for normal compiles.

The `javac` tag is one of the most complex in **ant**.

The `javac` task tag has a very large number of attributes, and may contain quite a number of other tags within it. First off, it is sensitive to a property, `build.compiler`, which may be used to specify that a particular Java compiler version must be used. The use of this will come up later when we build part of

8. In fact, task tag names correspond to the names of the Java classes that implement them. This will matter to you only if you wish to write your own **ant** tasks. We will not take you that far in this book.

our application using the GNU Compiler for Java, but for now, and in general, you will not set this property,[9] compiling with the default compiler for the JDK version you are using. This is **ant**'s default behavior.

`srcdir`
: Location of the Java source files.

`destdir`
: Location to store the class files.

`includes`
: Comma- or space-separated list of files (optionally using wildcards) that must be included; all `.java` files are included when this attribute is not specified.

`excludes`
: Comma- or space-separated list of files (optionally using wildcards) that must be excluded; no files (except default excludes) are excluded when omitted.

`classpath`
: The classpath to use.

`sourcepath`
: The sourcepath to use; defaults to the value of the `srcdir` attribute or to nested `src` elements. To suppress the `sourcepath` switch, use `sourcepath=""`.

`classpathref`
: The classpath to use, given as a reference to a path defined elsewhere.

`extdirs`
: Location of installed extensions.

`nowarn`
: Indicates whether the `-nowarn` switch should be passed to the compiler; defaults to `off` (i.e., warnings are shown).

---

9. The one place you are likely to need to set this is when you are using a recent JDK to compile applets that you wish to work in Netscape 4.0 and IE 4.0 and older browsers. But this book doesn't teach you Java applets. We swear.

`debug`
: Indicates whether the source should be compiled with debug information; defaults to `off`. If set to `off`, `-g:none` will be passed on the command line to compilers that support it (for other compilers, no command-line argument will be used). If set to true, the value of the `debuglevel` attribute determines the command-line argument.

`debuglevel`
: Keyword list to be appended to the `-g` command-line switch. This will be ignored by all implementations except modern and classic (version 1.2 and more recent). Legal values are `none` or a comma-separated list of the following keywords: `lines`, `vars`, and `source`. If `debuglevel` is not specified (by default) nothing will be appended to `-g`. If `debug` is not turned on, this attribute will be ignored.

`optimize`
: Indicates whether compilation should include optimization; defaults to `off`.

`deprecation`
: Indicates whether source should be compiled with deprecation information; defaults to `off`.

`verbose`
: Asks the compiler for verbose output; defaults to `no`.

`depend`
: Enables dependency tracking for compilers that support this (Jikes and classic).

The `jar` task. The `jar` task makes a JAR file.

The `javadoc` task. One of the greatest benefits of developing in Java is the nearly automatic generation of thorough and correct documentation. Javadoc is the tool that does the job, and the `javadoc` task is the way to automate document production in **ant**. This tag has a number of attributes that specify where the source to be documented is located, how the documentation is to be produced, and where the documentation is to be placed.

The `copy` and `mkdir` tasks. These tasks are used to copy files and make directories.

The `rmic` task. Remote Method Invocation is a distributed computing technology (Section 5.8). RMI requires the generation of server and stub classes that provide the networking support for an RMI-enabled class. This is normally done by the **rmic** compiler, and this is one of the common reasons the Java programmers turn to build automation tools. The `rmic` tag allows **ant** to build RMI classes.

9.4.2.5 Other Tags

So far we have ignored a major component of **ant**. In order to introduce it, we need to give you a (painfully) high-level view of how **ant** works "under the hood."

Task tags actually map directly to Java classes that implement the tasks. Each task class is an instance of the `Task` class (in other words, it is a Java class that either directly or indirectly extends the `Task` class). This is how you can write your own tasks—download the **ant** source code and write your classes that extend the `Task` class.

Tasks are not, however, the only tags that map to Java classes. There is another category of tags that do so. They are called *datatypes*. These are classes that directly or indirectly extend the **ant** `DataType` class.

Generally speaking, a task may require zero to many datatypes to specify the data with which the task works. Some such tags include the `manifest` tag used in our sample `build.xml` file discussed later in this chapter.

We'll mention a couple of the most frequently used datatype tags here and leave looking up the details as an exercise for you.

The `PatternSet`, `Include`, and `Exclude` datatypes. As you may have noticed, the most common tags we have covered allow you to specify the files to be processed using the tag's attributes. Usually, you nominate a base directory and let the task tag process everything in that directory.

There are times, however, when you need finer grained control than that. For example, you might wish to exclude all Java classes whose names end in "`Test`" (Example 9.3).

Other datatypes. There are many other datatypes used for various purposes. One of them, `FilterSet`, is able to modify files before they are copied or moved. This can be useful, for example, to put build information into a source file for an **About** dialog.

Example 9.3 Using the `PatternSet` datatype

```
<patternset id="nontest.source">
  <include name="**/*.java">
  <exclude name="**/*Test.java">
<patternset>

<target name="build">
  <javac destdir="build">
    <src path="src"/>
    <patternset refid="nontest.source"/>
  </javac>
</target>
```

In general, datatypes give you more sophisticated control than do the attributes of a task. Take a look at the *Ant User's Manual*[10] for full details on **ant**.

9.4.3 A Real, Live Buildfile

Let's take it to the next level and examine a real working buildfile.

9.4.3.1 Project Organization

All but the simplest of projects will require multiple classes. Some will require libraries, multiple programs, binary deployment, Web deployment, enterprise deployment, and so on. A project will be most successful if you plan out what goes where in advance. We're going to present a series of suggestions for how to organize the files involved in developing, building, and deploying a Java project with **ant**. By no means is this the only way it might be done, but it has worked well for us.

9.4.3.2 The `build.xml` File for Payback

Example 9.4 is the actual Ant buildfile for the Payback program in our source code examples. These examples are available on the book's Web site.[11]

10. `http://ant.apache.org/manual/`

11. `http://www.javalinuxbook.com/`

Example 9.4 The `build.xml` file for the Payback application

```
<?xml version="1.0"?>

<!--
  $Id: 070_antIntro.sgml,v 1.51 2004/04/13 05:10:45 mschwarz Exp $
  Buildfile for the Payback program.  Payback will calculate
  the length of time and real amount of money it takes to make a
  purchase using various savings or credit accounts. -->

<project name="Payback" default="all" basedir=".">

<!-- The "init" target sets up properties used throughout
  the buildfile. -->
<target name="init" description="Sets build properties">
  <echo>Running INIT</echo>
  <property name="src" value="${basedir}/src"/>
  <property name="build" value="${basedir}/build"/>
  <property name="doc" value="${basedir}/doc"/>
</target>

<!-- The "all" target does nothing but tie together the "jar" and
  "doc" targets. -->
<target name="all" depends="jar,doc"
        description="Pseudo-target that builds JAR and Javadoc">
  <echo>Building ALL</echo>
</target>

<!-- The "build" target compiles the code in the project. -->
<target name="build" depends="init"
        description="Compiles the classes">
  <echo>Running BUILD</echo>
  <mkdir dir="${build}"/>
  <javac destdir="${build}" srcdir="${src}" debug="true"
         deprecation="true"/>
</target>

<!-- The "doc" target generates Javadoc documentation of the
  project.  The "author", "version", and "use" attributes set to
  true cause those Javadoc tags to be used in the final document.
  The "private" attribute set to true causes private methods and
  attributes to be included in the documentation.  We tend to use
  this for projects to provide complete reference documentation.
  You would probably not want to do this for an app or lib
  distributed as a JAR file only. -->
```

```
<target name="doc" depends="init"
        description="Generates Javadoc documentation">
  <echo>Running DOC</echo>
  <mkdir dir="${doc}/api"/>
  <javadoc packagenames="net.multitool.Payback.*"
           sourcepath="${src}" destdir="${doc}/api"
           author="true"       version="true"
           use="true"          private="true"/>
</target>

<!-- The "jar" target depends on the "build" target.  It places
  all of the class files in the project into a JAR file, and
  builds a manifest using the "manifest" tag. -->
<target name="jar" depends="build"
        description="Builds an application JAR file">
  <echo>Running JAR</echo>
  <jar basedir="${build}" jarfile="${basedir}/Payback.jar">
    <manifest>
      <attribute name="Version" value="1.0"/>
      <attribute name="Main-Class"
                 value="net.multitool.Payback.Payback"/>
    </manifest>
  </jar>
</target>

<!-- The "run" target depends on the "jar" target.  It executes
  the class named as the "Main-Class" in the JAR's manifest. -->
<target name="run" depends="jar" description="Runs the program">
  <echo>Running RUN</echo>
  <java jar="${basedir}/Payback.jar" fork="true">
    <arg value="${basedir}/accounts.properties"/>
  </java>
</target>

<!-- The "clean" target erases all files and directories that other
  Ant targets might have generated.  It returns a copy of the
  project tree to a "pristine" (some might say "clean") state. -->
<target name="clean" depends="init"
        description="Erase all generated files and dirs">
  <echo>Running CLEAN</echo>
  <delete dir="${build}" verbose="true"/>
  <delete dir="${doc}/api" verbose="true"/>
  <delete file="Payback.jar" verbose="true"/>
</target>

</project>
```

9.5 REVIEW

We've taken a very quick tour through the most popular tool for building and deploying Java applications. We've shown you how to install and set up **ant**. We've given you enough description of XML so that you can read an **ant** buildfile. We've touched briefly on the basic tags used in most buildfiles.

9.6 WHAT YOU STILL DON'T KNOW

What we've covered here will probably explain most of the buildfiles you encounter in the wild. It will probably also give you what you need to know to build most of the buildfiles you will have to build. But we have left out a fair amount.

You can write your own tasks. That's a biggie right there. There are many built-in tasks that we didn't cover. Look those up. They might be just what you need if you find yourself saying, "Boy, **ant** is nice, but I wish it could do X." X might already be a built-in task. And if not, you can write it.

Ant has datatypes that can often be used in place of the simple strings allowed in task attributes. Tasks and datatypes are instances of Java base classes defined in the source code for **ant**. If you download the source, you can write your own classes that extend these base classes and you can thus add your own tags to **ant**.

9.7 RESOURCES

- *The Ant User's Manual*[12] at the Ant project homepage[13] (which itself is part of the Apache Project[14]) is the definitive resource for **ant**.

12. `http://ant.apache.org/manual/`

13. `http://ant.apache.org/`

14. `http://apache.org/`

- O'Reilly has also published *Ant: The Definitive Guide* by Jesse Tilly and Eric M. Burke (ISBN 0596001843), which was the first book on **ant** that we read.
- If you are into Extreme Programming you'll want to check out *Extreme Programming with Ant: Building and Deploying Java Applications with JSP, EJB, XSLT, XDoclet, and JUnit* by Glenn Niemeyer and Jeremy Poteet, published by SAMS (ISBN 0672325624).

Chapter 10

Integrated Development Environments

Some people prefer glitz. The shiny sparkle has always attracted the human eye. Sometimes that sparkle is a real gem, a treasure; sometimes it's only a gum wrapper on the sidewalk. Integrated Development Environments (IDEs) add glitz to Java development. At their most basic, they combine (integrate) an editing environment with a compiler. This gives you

- Language-specific formatting of your Java text (different colors, comments in italics, and so on)
- Quick feedback on errors (the ability to click on a compile error message to begin editing at the offending Java statement)
- Automatic word completion, to help you finish typing the names of methods in your Java code
- A point-and-click GUI for that all important "modern" look-and-feel

If those were all they gave you, IDEs would be, in our opinion, leaning toward "gum wrapper." But a good IDE can be more than that. It can be extended to integrate many different tools, including:

- Version control (e.g., CVS, see also Chapter 8)
- One or more Web servers (e.g., Tomcat)
- A build control mechanism (e.g., **ant**, see also Chapter 9)
- Other editors besides the built-in editor
- A specialized editor for building GUIs
- Other languages besides Java

10.1 WHAT YOU WILL LEARN

In this chapter we will examine two major Open Source IDEs, NetBeans and Eclipse. We will show a straightforward installation of each. We will describe the "operating paradigm" of each and show a few major features. It should be enough to get you started using them.

There are several major commercial IDEs, including Idea by IntelliJ, JBuilder from Borland, WebSphere Studio from IBM, SunONE Studio (Enterprise Edition) from Sun, and others. Because they are commercial, and not Open Source, we will not be covering them; their vendors and other experts can provide the documentation and training you need. Be advised, however, that the licenses for such commercial products typically cost anywhere from several hundred to a few thousand dollars per seat. That can make Open Source IDEs look very attractive.

10.2 NETBEANS: THE OPEN SOURCE IDE

NetBeans is an Open Source IDE, freely available, with full source code. It is also the basis for the SunONE Studio (more on that product later).

10.2.1 A Brief History of NetBeans[1]

NetBeans (originally called Xelfi) began in 1996 as a student project in the Czech Republic, with the goal being to write a Delphi-like Java IDE in Java. A company called NetBeans was formed around this project. By May of 1999, after two commercial releases of Developer, the company released a beta of what

1. From the `netbeans.org` Web site.

was to be Developer 3.0. In October 1999 NetBeans was acquired by Sun Microsystems. After some additional development, Sun released the Forte for Java Community Edition IDE—the same IDE that had been in beta as NetBeans Developer 3.0. There had always been interest in going Open Source at NetBeans. In June 2000, Sun open-sourced the NetBeans IDE; now it can be found at the `netbeans.org` Web site.

10.2.2 Installing NetBeans

NetBeans can be downloaded from the `netbeans.org` Web site. You will want the NetBeans "IDE" and *not* the NetBeans "platform." The IDE is the fully featured Java development environment. The platform is the underlying core of NetBeans on top of which one can develop other tools—for example, IDEs for other languages. Installation of the IDE consists of only three steps:

1. Download.
2. Install.
3. Execute.

10.2.2.1 Downloading

The first step is to get the software downloaded onto your system. From the `netbeans.org` Web site, navigate your way to a download of the latest IDE. The prepackaged "installers" might work—but if they fail, you have no information as to why, and still less as to what you can do about it. We'll act like "real programmers" and download an archive file. (Here "archive" means a collection of software compressed for easier transmission, not "archive" in the sense of "old documents.") Click on a link to begin the download (you'll need to read, review, and accept the license agreement to proceed). The result should be a file on your system named something like `NetBeansIDE-release35.tar.bz2`.

10.2.2.2 Installing

The installation consists of three steps: untarring the file, adjusting a parameter in a configuration file, then creating a symbolic link for easy startup.[2]

2. Thanks to John Zoetebier from New Zealand for his contribution on the NetBeans users mailing list, on which this manual installation procedure is based.

Uncompress the archive:

```
bunzip2 NetBeansIDE-release35.tar.bz2
```

This will leave the file `NetBeansIDE-release35.tar` in place of the `.bz2` file.

You can examine the contents of the TAR file with:

```
tar -tvf NetBeansIDE-release35.tar | more
```

Here the options (`-tvf`) specify to show a table of contents (`-t`) in verbose, that is, long, form (`-v`) from the specified file (`-f`) followed by the TAR filename. The output from **tar** here is piped into **more** so that you can page through it. Type `q` when you've seen enough, or leave off the `| more` to let it run through without pausing.

Notice that the names of all the files in the TAR archive begin with `netbeans/` which tells us that if we untar the file, it will put all the files into a directory called `netbeans`. Therefore, we don't need to make such a folder beforehand.

Change directory to the directory where you would like to install NetBeans. If you are on a system that may be used by different users, you'll probably want to put it in a more public location like `/usr/local` or `/opt`. If it is for your personal use, you can put it anywhere—just be sure that you have write permissions on the directory where you want to install NetBeans. (Reminder: use `ls -ld .` to see the permissions of the current directory.)

The **tar** command to untar everything in place is simple:

```
tar -xf NetBeansIDE-release35.tar
```

This will extract (`-x`) all the files that are in the TAR file (`-f`) named `NetBeansIDE-release35.tar`. If you'd like to see each file get named as it is extracted, then change the `-xf` to `-xvf` (`v` for verbose) and you will see a whole long list of filenames displayed on your screen as the file is unpacked.

Next, we need to adjust the startup parameter in the configuration file. The file is in the `netbeans` directory that you just untarred. In there is a directory named `bin`, and in there is a file called `ide.cfg`. Open this file with an editor and change the line that begins `-jdkhome` so that the pathname refers to the location of your Java Development Kit (JDK, see Chapter 5).

Here's an example of the contents of `ide.cfg`:

```
-J-Xms24m -J-Xmx96m
-J-Xverify:none
-jdkhome /usr/local/java/j2sdk1.4.1_01
```

This specifies that the Java SDK is located in `/usr/local/java/j2sdk1.4.1_01`, and in that directory there is a `bin` directory which contains **java**, **javac**, **javadoc**, and so on.

Finally, to make the executable easy to run, we will construct a symbolic link for easy access:

```
$ ln -s /usr/local/netbeans/bin/runide.sh /usr/local/bin/nb
```

This creates a symbolic link from the `runide.sh` file in the current directory to the `/usr/local/bin` directory, as a file named `nb`.

10.2.2.3 Running

Now that you have the symbolic link to the script which runs NetBeans, simply invoke it with `nb` at a shell prompt. NetBeans will start loading (Figure 10.1).

If you get a message like "Command not found," check to see if the shell knows where to look. If you used the same name and location as in the previous example, then make sure that `/usr/local/bin` is part of the search path for the shell. (Reminder: `echo $PATH` will tell you what directories are searched. If you need to, add `/usr/local/bin` to `PATH`, as in:

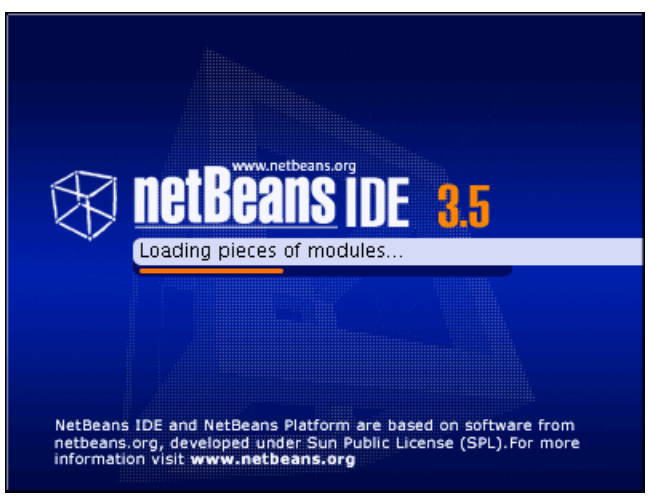

Figure 10.1 NetBeans begins

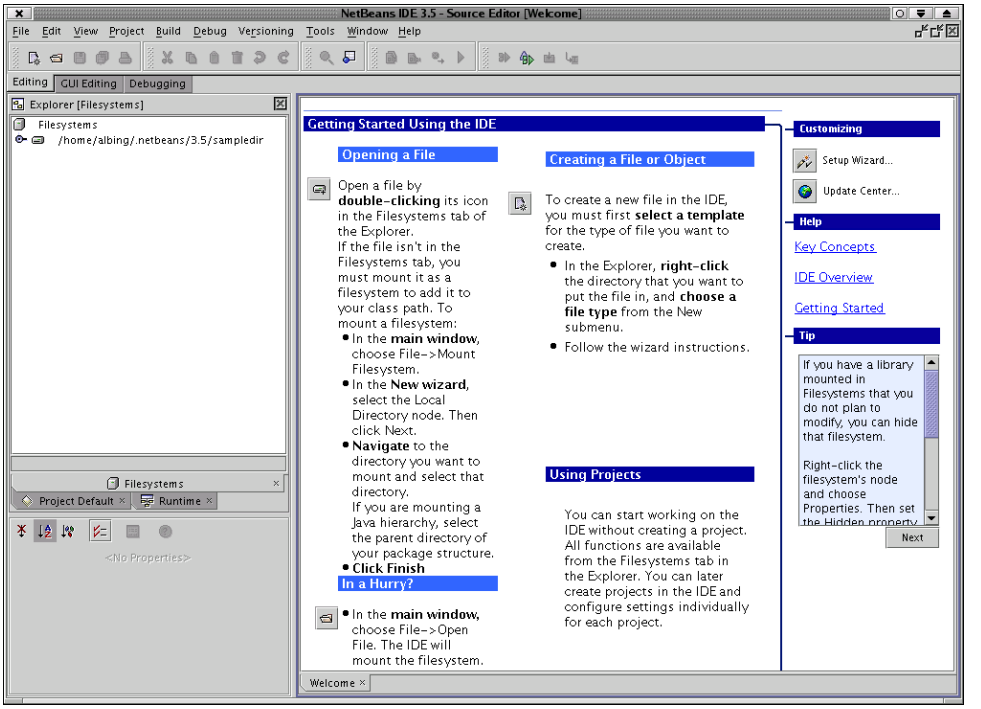

Figure 10.2 NetBeans' main window

```
export PATH="${PATH}:/usr/local/bin"
```

which you can type from the command line for immediate use and then put in the `.bashrc` file in your home directory to set things this way next time you invoke a shell.)

Also be sure that the script `runide.sh` has execute permissions on it. (Reminder: check this with `ls -l runide.sh` and change with `chmod a+x runide.sh`.)

If all goes well then after the splash screen, you should see the window shown in Figure 10.2.

For more information on installing NetBeans, check out `http://www.netbeans.org/kb/articles/install.html#unix`.

Now let's take a look at how you might use NetBeans.

10.2.3 Getting Around in NetBeans

Let's take a look, from top down, at NetBeans' main window. First, of course, is the menu bar. There are lots of choices to explore there. Much of what you'll do with NetBeans won't require much use of the menus—there are so many shortcuts elsewhere.

Next comes a row of icons, which are just shortcuts for menu times. This row of icons can be customized, and you can even add your own (see Section 10.2.5).

The three tabs below the icons, labeled **Editing**, **GUI Editing**, and **Debugging**, modify the window to provide three distinct workspaces. Each one customizes the window environment for a specific task, but it is still working on the same files.

Next, on the left, comes the **Explorer**, which is in many ways similar to the tools that you may use for traversing filesystems on a Windows or Linux system.

One oddity of NetBeans is that it doesn't just use the files as it finds them in the directories on your hard drive(s). Rather, is requires you to designate a piece of the filesystem as the part that you want to use. You can designate several such pieces. Each piece is "mounted" as if it were a mountable filesystem. (This is an operating system concept. If you're not familiar with it, don't worry. For the purposes of NetBeans, just think of the IDE as too dumb to know about any files until you tell it about them.)

There are three different types of files that you can mount—local, CVS, or JAR. By specifying the type, NetBeans can treat each one in its special way.

- Local files need no special treatment; they are just the local files on your hard drive.
- If a filesystem is mounted under a version control system (CVS or generic VCS), then its files can have version control operations performed on them (checkin, checkout, and so on), via commands in the IDE. (More on that below.) Also, special directories used by the version control system (e.g., CVS) are hidden from the display, as you almost never want to manipulate these files directly.
- When you mount a JAR file or ZIP archive as a filesystem, NetBeans displays the *contents* of the archive as if they were just files in a directory—which can make them easier to manipulate. More importantly, the JAR is automatically added to the classpath for Java compiling.

Therefore, any third-party JARs that you may need for your project should be mounted.

To mount a local directory as a filesystem, right-click on the little icon labeled **Filesystems** in the **Explorer [Filesystems]** window. Choose **Mount**, then **Local Directory**, and you'll get a filechooser to let you navigate your file structure and choose a directory to mount.

IMPORTANT

To ensure that NetBeans knows how to compile your source, you need to mount the directory that contains the base level of your source as a mountpoint, not just have that directory somewhere in a tree of directories.

For example, let's say that your source is kept in two packages, `com.coolco.projecta` and `com.coolco.util` which implies that you have a directory structure with those names. Let's further assume that you keep them in a directory called `src` which is itself contained in a directory called `brolly`, as shown in Figure 10.3.

The likely thing to do is to mount the `brolly` directory, since it will contain the source and all sorts of other project-related directories. That's fine, as far as it goes. But since the mountpoints in NetBeans are also the `CLASSPATH` directories, you need to also mount `brolly/src`, so that directories like `com/coolco/util` are found when your Java sources have statements such as `import com.coolco.util.*;`.

It's OK to have the same directory show up in different mountpoints. NetBeans won't get confused, although you may. You'll probably want to edit and compile from the mountpoint of, in this example, `brolly/src`. The `src` folder inside the `brolly` mountpoint would refer to the same files. Just keep the one always open and the other closed, and you should be able to keep them straight.

10.2.3.1 Filesystem versus Project

The **Explorer** window has tabs at the bottom which let you look at different aspects of your work. In some instances you'll want the **Project** view. For this, you have to mount file's from the already mounted filesystems in the **Filesystem** view. Seems redundant, no? The **Project** view lets you set properties for the project as a whole or for individual files in the project. These settings apply to that file only for the project. Another project, looking at the same files, might have different settings.

For now, don't worry about the difference. Many people like to work in the **Filesystem** view and never bother with projects. Others, especially those working on multiple products or projects, like **Projects** as a way to switch

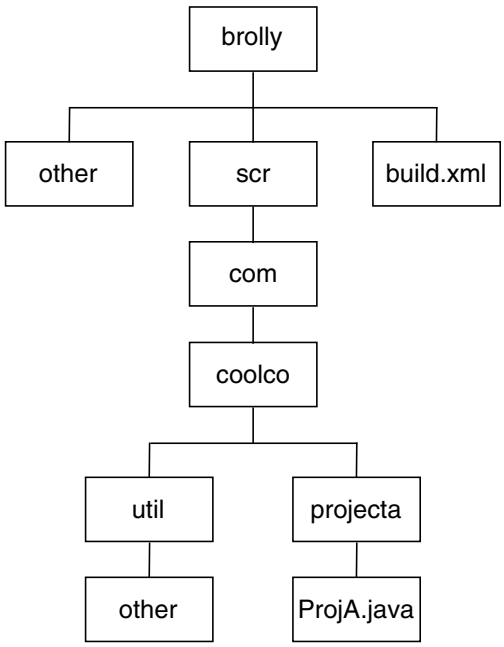

Figure 10.3 A simple source structure

between tasks—you can only have one project active at a time, but when you switch projects, it switches all the mounted filesystems and other settings that you have configured.

10.2.3.2 Editing

Like other IDEs, NetBeans provides its own editing window. It's a GUI point-and-click environment, with syntax highlighting and other helpful features for a programmer.

At the top of the editing window is a toolbar (Figure 10.4). Each icon on the toolbar has a tooltip, a help text that pops up when you rest your mouse pointer over the icon, to explain the somewhat cryptic little icons. Most of the tools are quite handy. With the pulldown menu, you can navigate to any method or class variable within the class. The next four buttons deal with searching: Select any text in your source file, click on the magnifying glass icon, and the search will be performed for the next occurrence of that text. In addition, all occurrences are highlighted. This highlighting can be toggled on or off.

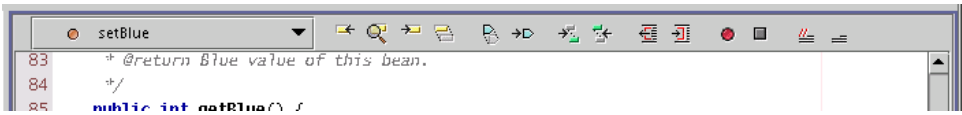

Figure 10.4 NetBeans' **Edit** screen toolbar

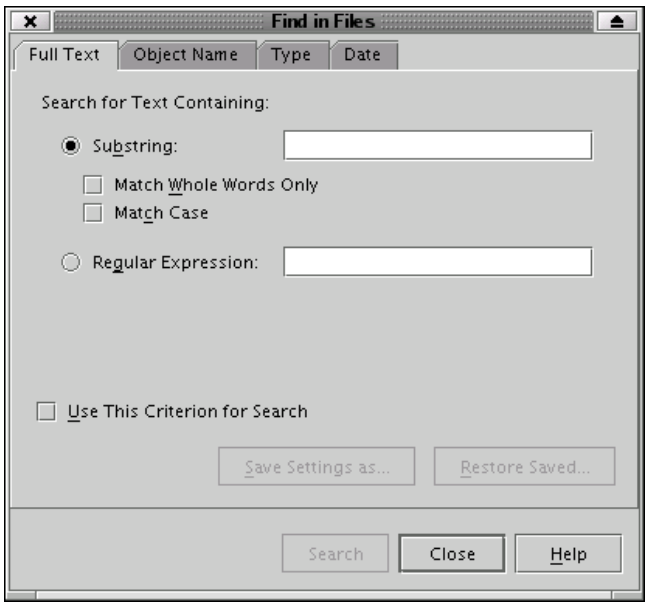

Figure 10.5 NetBeans' **Find** dialog

The toolbar search only works within a single source file. If you want to search across multiple files, go back to the **Explorer** window and right-click on the folder containing the files you wish to search. There is a **Find . . .** command in the pop-up menu. That brings up a dialog box (Figure 10.5) that has multiple tabs for quite extensive filtering of your search. In its simplest use, just type in the text you want to find, and press Enter.

A list of the files which contain the text will appear in a different window, citing filename and linenumber for each file. There you can double-click on any citation to bring up that file in the edit window, at that location.

If you heeded our admonition to learn **vi**, you'll be glad to know that NetBeans can handle the fact that the source files can be modified externally from the IDE. Go ahead and edit any of your source files, even while the IDE

is running. When you next touch the file from within the IDE, NetBeans will recognize the fact that the file has been modified and load the new version.

If you haven't yet learned **vi**, you may find yourself quite comfortable using the NetBeans editor. If you dig deeper into NetBeans you can find how to map certain keystrokes to make it even more editor-like. However, mousing and cut-and-paste may suffice for beginners for quite some time.

10.2.4 Integration with CVS

NetBeans comes with a built-in CVS client, which means that you don't need to install any additional features in NetBeans to get it to talk with a CVS server. It has all it needs to check out and commit files from and to a CVS repository. NetBeans can be configured to use external (i.e., outside of NetBeans) CVS commands, but you likely won't need to do that.

What you will need to do, however, is tell NetBeans that the files you are using are under CVS control. You do this by mounting the filesystem not as just a regular filesystem, but as a CVS filesystem. In the **Explorer** window, go to the **Filesystem** tab if you are not already there. On the **Filesystem** icon, right-click your mouse, and from the pulldown menu choose **Mount**, then **Version Control**, then **CVS** (Figure 10.6). What follows will be a wizard-like series of dialogs which you will fill in to describe the type and location of the CVS repository with which you want to work. Those choices and values are specific to your installation, so we'll leave that for you to figure out with your network administrator or whoever has set up your repository.

Once mounted, the CVS filesystem's files will look much like any other filesystem you have mounted—except that the files will show, via their icon, when they have been modified and need to be committed, and will show the version number in parentheses after the filename. The other difference is that there is now a **CVS** command on the pulldown menu (Figure 10.7) that appears when you right-click on one of the filenames (or on its tab in the Edit view).

Move your mouse over the **CVS** command; an additional pulldown menu appears (Figure 10.8). If you've used CVS at all, then you'll recognize the list of commands in the cascaded menu. There are the **Commit**, **Update**, **Diff**, **Log**, **Status**, and **Checkout** commands that you are familiar with. The first item, **Refresh**, is likely unfamiliar, though. Not being a CVS command (it's not part of the command-line syntax), it is a way for you to tell the IDE to

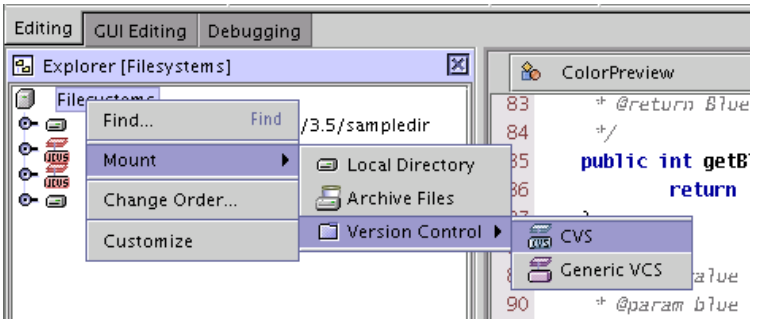

Figure 10.6 Mounting a CVS filesystem

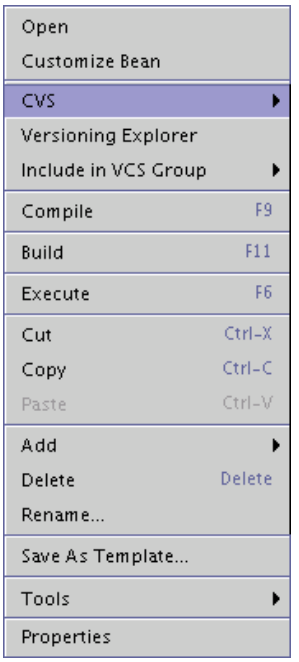

Figure 10.7 Right click on a CVS file

reconsider what it thinks about the CVS-related information displayed in its icons and the parenthetical text.

If you click on a folder instead of a single file, then the **Refresh** command will be followed by **Refresh Recursively** which will do a refresh on each file from there on down the tree.

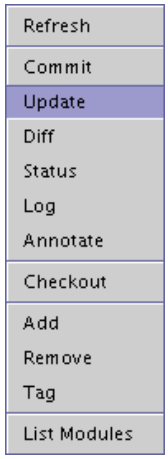

Figure 10.8 Cascaded menu with CVS commands

Using the integrated CVS is much like using the command line. If you want to see the changes that you've made (before committing), use the **Diff** command. A window will appear showing the two different versions and coloring the lines that have been added, deleted, or changed.

When you need to commit your changes, click the filename, then right-mouse your way to **Commit**. A dialog window will appear for you to type in the comment that you want to be associated with this change. (This comment will appear in the **Log** command display).

To incorporate the changes others on your project may have made on the file, use the **Update** command. A dialog box will display the CVS output showing what was updated and if any merge conflicts occurred. (See Chapter 8 for more on this.)

The CVS commands in the menu, as we've described them so far, don't allow you to add any options to the commands. They just run with the defaults. What if you want to use some of the options available on the CVS command line? Then hold down the Ctrl key just before your make your CVS choices. You should see an ellipsis ("**...**") appear after each CVS command for which you can now select options (Figure 10.9).

Of course one of the great things about knowing the command-line version (see Chapter 8) is that you're not limited to what the GUI tools will do for you. If you can't find the option you want, just go back to a window with a shell prompt, `cd` into the appropriate directory in your source tree, and type the CVS command by hand. As noted earlier, NetBeans is smart enough to

Figure 10.9 Cascaded menu after choosing CVS with Ctrl pressed

catch on to the changes made outside of NetBeans to its files, though you may need to do a **Refresh**, as described above.

10.2.5 Integration with Ant

If you've already discovered **ant**, either by using it on a Java project or by reading this book in chapter order, then you'll know that it's a plus to have **ant** integrated into NetBeans. As of NetBeans version 3.5, **ant** comes bundled with NetBeans and you don't need to install it separately.

NetBeans recognizes a `build.xml` buildfile and gives it a special icon. If you click on the icon for the `build.xml` file in the **Explorer**, it will show each of the properties and then each of the targets (Figure 10.10). Right-click on a target to choose **Execute** to run **ant** with that target. As a shortcut you can either select that target and press the F6 key, or you can just double-click on the target name.

If you are making frequent use of an **ant** build script in your project, you may want to add a shortcut—an icon that you can put on the icon panel—that

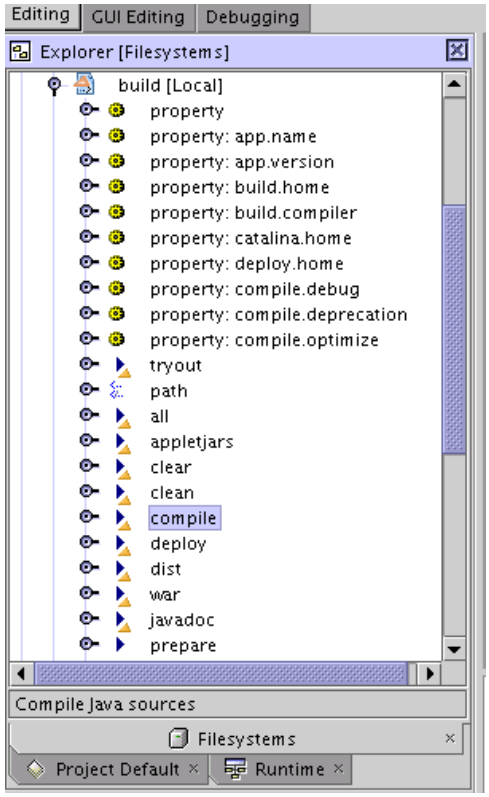

Figure 10.10 Cascaded menu after choosing CVS

will run a specified **ant** target. You can also add a keyboard shortcut, to invoke the target with a keystroke or two. The specific steps for doing that are found in the NetBeans online help. Just look for the section titled Creating a Shortcut to a Target.

There is one other topic worth mentioning about integration with **ant**. Normally in NetBeans, when you compile a Java class, the IDE does the compilation. You can configure the IDE to use different types and versions of Java compiler, but it is the IDE which controls the compile. You can invoke a compile with the F9 key as a shortcut. But if you are building with **ant**, you may want **ant** to do the compiles for you. Fortunately, you can configure NetBeans to do this. Again, we'll refer you to the NetBeans online help, to a section called Indirect Ant Compilation.

10.2.6 Other Add-on Tools

NetBeans is built to allow other tools to be plugged into it. There is a module that you can add to NetBeans to support C/C++ development. But there are various other plug-ins available. The best reference for all of those is the NetBeans Web site.[3] Many of the add-ons are commercial, but one notable tool is available, at least in its simple form, as a free download.

Poseidon for UML Community Edition is a UML modeling tool that integrates with NetBeans. A product of Gentleware AG, it is distributed at their Web site.[4] The Community Edition is offered free of charge, but they have more advanced (professional, enterprise) versions for sale. Their Web site says that Poseidon for UML Community Edition has the following features:

- Fully implemented in Java, platform-independent
- All 9 diagrams of the UML supported
- Compliant to the UML 1.4 standard
- XMI supported as standard saving format
- Runs under Java 1.4
- Diagram export into GIF, PS, EPS, SVG, JPEG, and PNG
- Undo/redo (can be turned on in options)
- Copy/cut/paste within the tool
- Drag and drop within the tool
- Zooming and bird's eye view of the diagrams
- Internationalization and localization for English, German, French, and Spanish
- Code generation for Java
- Sophisticated support of OCL
- Reverse engineering from Java sources
- Auto-layout of reverse engineered diagrams
- Cognitive support, critique mechanism (by default turned off)
- Simple install and update with JavaWebStart

3. `http://www.netbeans.org/about/third-party.html`

4. `http://www.gentleware.com/`

We won't be discussing its installation or use, but we encourage you to explore this option if you need a UML tool.

10.3 SunONE Studio Community Edition

SunONE Studio (SOS) Community Edition was built on NetBeans and was distributed for free. You would even see the NetBeans logo on startup. As of SOS 5.0 (summer 2003), the Community Edition is no longer available—NetBeans is what Sun recommends for the no-cost download. (The SOS Enterprise Edition is still actively developed and sold by Sun; it is licensed at a cost similar to other commercial packages.)

10.4 Eclipse: The Source of SWT

Eclipse is yet another GUI IDE. The Standard Widget Toolkit (SWT) was invented as part of this project. Since Eclipse uses SWT for its graphics, it has the most familiar GUI look and feel, and it is the fastest performer on lower end hardware.[5] Eclipse is built as an IDE toolkit. Although it was developed in Java/SWT, you can use it to develop AWT/Swing applications. It is build around a plug-in architecture, so it can be an IDE for virtually any language. To use it for Java, you must actually install the Java plug-ins.

10.4.1 Selecting and Installing Eclipse

The main site for obtaining Eclipse is `www.eclipse.org`. Installing Eclipse, particularly for a single user, is incredibly easy. But first, you have a few choices to make as to what to download. As of this writing, the current production release of Eclipse is 2.1.2. If you follow the Downloads link from the main page to the 2.1.2 build, you will see a fairly lengthy list of download choices.

The first major choice you must make is whether or not you plan to write your own plug-ins and extensions to Eclipse. If you do, you will probably wish to download the Eclipse Software Development Kit or the source code and build Eclipse for yourself. We strongly recommend that you choose the Eclipse SDK binary for GTK. The Motif UI is quite dated in comparison. The Eclipse

5. It's the fastest performer on high-end hardware too, but the difference is much less perceptible. Really fast and incredibly fast are hard for humans to discern.

SDK contains the Eclipse Platform (which you need), the Java Development Toolkit (which you need), and the Eclipse Platform Development Kit (which you need only if you plan to develop plug-ins for Eclipse). But by downloading the Eclipse SDK package, you get everything you need in one go. You could also download the Eclipse source package and build the whole thing yourself; save that for a spare week. For now, start with a prebuilt binary.

If you scroll down, you will see a collection of files that have "platform" in their names. You will note that there are two choices of Linux binary: one is Motif, the other GTK. If you are not familiar with these, Motif and GTK are two common extensions to the X Window API that provide widgets and other common functions and UI features. One of these, Motif, is rather old and (to brashly add opinion) dated in appearance, but it is very stable and mature. The other, GTK, stands for GIMP Toolkit and was developed to support the remarkable GNU Image Manipulation Program (GIMP). There are other widget/UI libraries that run on top of X Window, notably the Qt library used by KDE.

So, which to use? If you read the documentation on the Eclipse Web site, you will see that the Motif version has been more heavily tested and is available for other platforms than Linux. This is probably because Motif is standard on most commercial UNIX versions, and thus is where emphasis was placed to get the "most bang for the buck" in development and testing.

However, we much prefer the look of the GTK version and, to date, have found no major problems with it, so that is what we use for our examples. There should be no functional difference between the two—merely differences in the look and feel of menus, toolbars, and dialogs. One reason to select Motif might be if you are working in a mixed environment of Linux and other UNIX platforms, where you may be forced to use the Motif version on some platforms, and do not want the "cognitive dissonance" of switching between the two.[6]

So, **step one:** Download `eclipse-SDK-2.1.2-linux-gtk.zip`.

The Eclipse platform is a "generic IDE." You will see the term *perspective* all over Eclipse. A perspective is kind of a collection of tools in the IDE. The package you just downloaded contains a generic perspective called Resource. A

6. Please note that the differences are fewer and smaller than the differences involved in switching between any common X Window desktop and Microsoft Windows. If you can handle that (and many of us do every day), switching between Motif and GTK versions of Eclipse will be no problem for you.

perspective is a set of *views*, which are panes within the IDE, each having a specific purpose, such as editing the project contents, editing files, keeping a task list, and so on, as well as menus and toolbars relevant to those views. The Resource perspective has its uses, but it is not the main one you will be using as a Java programmer. As a Java programmer, you will most likely want the Java perspective.[7]

First, you must decide if you are going to install Eclipse in a common location (such as `/usr/local` or `/opt`), or if you are just going to install it in your home directory for your own use. The answer to this question, naturally, depends on whether or not you have root access on the machine and whether or not multiple people actually use the machine.

> **CAUTION**
>
> We are assuming you already have at least one Java SDK installed. If you do not, refer to Chapter 6 for some tips on installing Java SDKs.

We're going to install in the user's home directory. Doing this could hardly be simpler. So, **step two:** From your home directory, type:

```
$ unzip eclipse-SDK-2.1.2-linux-gtk.zip
```

That's it. You're done. Now just `cd` to the newly created `eclipse` directory and type `./eclipse`. The first time you do this, you will see a "Completing the install" banner (Figure 10.11).

Figure 10.11 Running Eclipse for the first time

7. Although you will also often be using the Debug perspective.

Figure 10.12 Eclipse splash screen

During this initial run some workspace directories and data files are set up. These store meta-information about projects and perspectives. After a moment, you will get the standard splash screen (Figure 10.12).

Following this, you will see the initial Eclipse IDE, with a welcome screen in the default Resource perspective (Figure 10.13).

Eclipse works with *projects*. A project is a collection of files that you manage as a group. Usually a project is a single program, although it need not be. Eclipse remembers the state of all projects. If you close Eclipse in the middle of a debug session on a project, the next time you open Eclipse, it will have that same project open in the Debug perspective. If you then switch to another project and switch back, you will come back to the Debug perspective. Eclipse remembers. But we get ahead of ourselves here. You need to create a project.

10.4.2 Using Eclipse for a New Project

Now that you have Eclipse up and running (Figure 10.13), you will want to create a Java project. Eclipse has "wizards" (to use the Microsoft Windows terminology) for creating projects. From the main menu, select **File** > **New** > **Project**. You will get the screen shown in Figure 10.14.

Now, you are not an idiot. This is not "Java Applications Programming on Linux for People with Well Below Average Intelligence." We're not going to walk you slowly and painfully through a screenshot of every window Eclipse

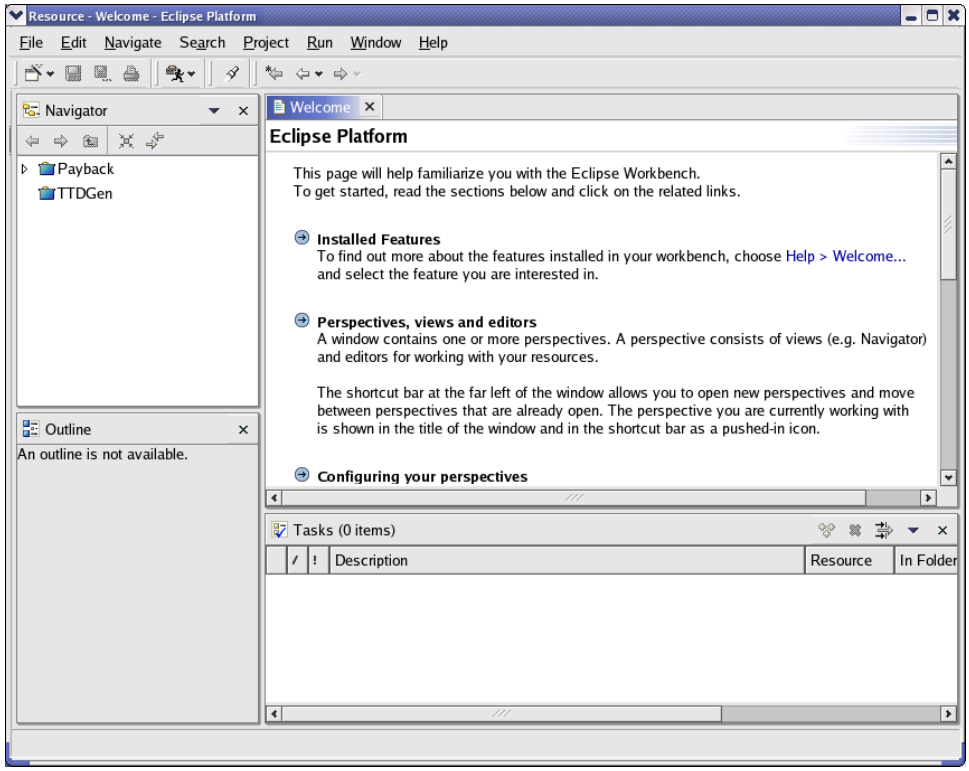

Figure 10.13 The Eclipse Resource perspective

can display and pretend that this is "educational." We like trees too much to do that. So let's pick up the pace a bit. The first screen in Figure 10.14 asks you to select the type of project. The next screen asks for a project name and it automatically chooses a directory to house the project. In Section 10.4.3 we will talk about changing away from this default. For now, trust Eclipse to do it right. Enter a project name and hit the **Next** button.

The next screen shows a tabbed display. The first tab allows you to add folders for source code. If you have already worked out a directory structure for your project, you may build it here. The next tab is the **Projects** tab. Here you specify other projects on which this project depends. The next tab is the **Libraries** tab. Basically, it lets you specify the project's classpath. Eclipse puts its preferred set of JARs on the classpath by default, including those that are standard with whichever Java runtime Eclipse found on startup. The last tab is called **Order and Export**; it allows you to specify the order in which source

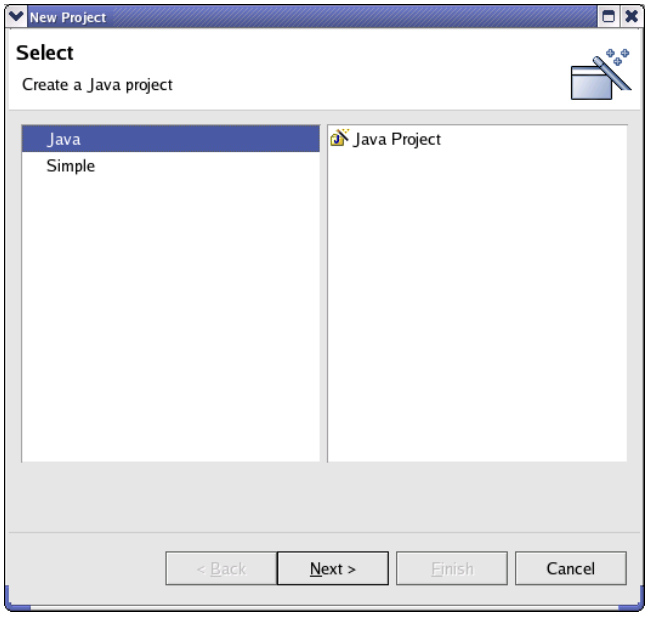

Figure 10.14 Step one: New project

folders should be built, as well as to specify which folders are accessible to other projects that name this project on their **Projects** tabs (in other words, this is the "other side" of the **Projects** tab).

That's the final wizard box. Hit the **Finish** button. A dialog box pops up to tell you that the project you just created is associated with the Java perspective. Since you are currently in the Resource perspective, it asks you if it may switch you to the Java perspective. Hit the **Yes** button, and you will be rewarded with the Java perspective display (Figure 10.15).

At this point, your exploration of the Eclipse tools would be more fruitful than anything we could tell you. There is a marvelous book on Eclipse, *The Java Developer's Guide to Eclipse* by Sherry Shavor et al. from Addison-Wesley. We encourage you to seek out that book.

10.4.3 Using Eclipse for an Existing Project

In the previous section, we mentioned that the second wizard screen in setting up a Java project allowed you to override the default path for a project. If you have an existing Java program, point the project path to the directory that is the base for that program's packages. When you create the project (in other

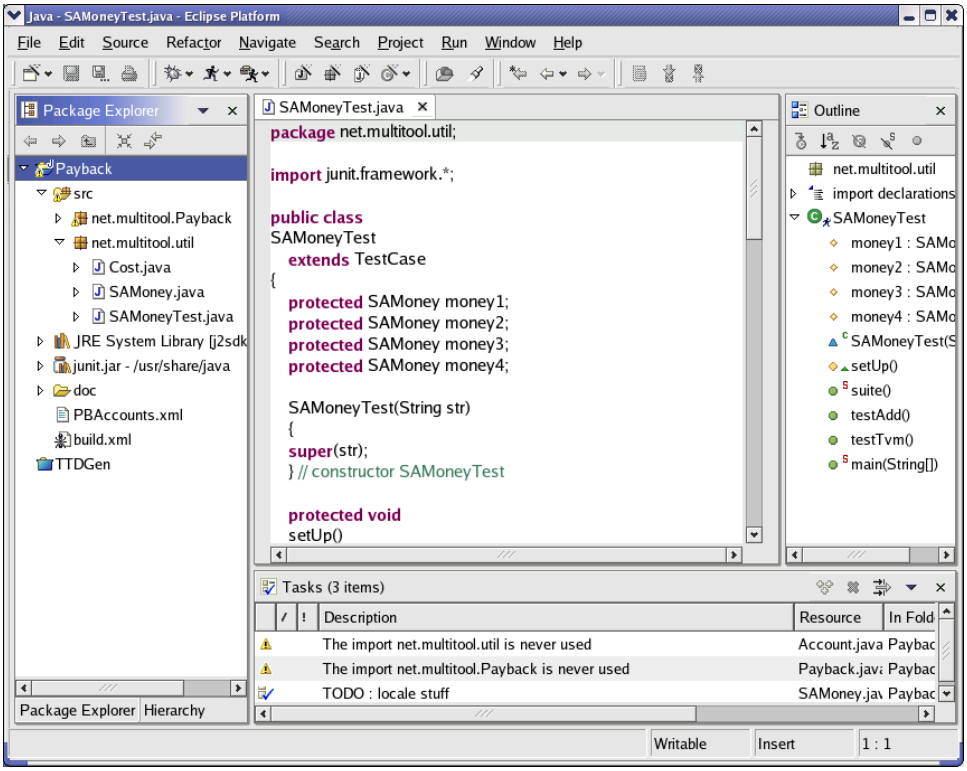

Figure 10.15 The Java perspective

words, when you hit the **Finish** button in the wizard), Eclipse will scan that directory and analyze what it finds there. It does an excellent job of importing a project and "eclipsifying" it, even to the point of recognizing and analyzing an **ant**'s `build.xml` file.

10.4.4 Working with Eclipse

Eclipse is a huge topic. We can't devote enough space to it for this chapter to qualify as a user guide. The best we can do is to offer you a handful of tips.

- The Java perspective consists mainly of the **Package Explorer** on the left, the edit window in the middle, the outline view in the right, and a bottom window that changes based on context. Initially, it is a task list. When you run a Java application, the console output is displayed there, overlaying the task list.

- You can do a lot of what you need to get started on a project by right-clicking in the **Package Explorer**. For example, you can create a package by right-clicking and selecting **New>Package**. When you type a new package name into the resulting dialog box, all required folders are created under the project.
- You can create new classes and interfaces in the same way. If you right-click on a package in the **Package Explorer** and select **New>Class** or **New>Interface**, an appropriate dialog box comes up, and a skeletal file is created in the appropriate place.

You can compile, run, and debug programs by hitting buttons on the toolbar, by selecting from the menu, by right-clicking almost anywhere, and by keyboard shortcuts. To put it plainly: Eclipse is a modern, rich IDE that works like other IDEs you have seen.

Eclipse's real advantages lie in some of the dynamism it offers. As you know, the authors of this book like text mode and command line, but we must admit that Eclipse's refactoring features are a great timesaver. For example, when you use the refactoring tools to change a method, you can be certain that every call to that method, everywhere in the project, is updated. Sure, we staunch CLI guys will tell you that you can use pipes and **sed** to similar effect, but even we must admit that you can miss some. We stand by what we have said: Know all the tools, and you can then choose the right one. But if all you have is a hammer, you'll end up using it to repair a china cup.

10.5 REVIEW

We've given you the choice of two great development environments. With them you can do so much more than just edit and compile. Both are expandable to include other tools, like CVS and JUnit. Each has a slightly different paradigm for how they manage files and projects. It seems the longer we work with one (either one), the more we like it. They kind of grow on you and you get used to some of the shortcuts that they offer. And yet, there are still those times when it's handy to be back at the simple command line.

10.6 WHAT YOU STILL DON'T KNOW

NetBeans comes with a built-in version of Tomcat for serving up Web pages and JSP and Java Servlets. It's very handy for developing and testing on your desktop. We'll look at that more in Part IV of this book.

In the NetBeans help file, you'll find this intriguing note:

> Using Scripting Languages in NetBeans: NetBeans provides you with a scripting feature that lets you use scripts to operate the IDE remotely or from the Scripting Console or by using a scripting file. You can use the scripting languages provided in the Scripting Console, or you can create a scripting class through the New From Template wizard. The following scripting languages are provided with NetBeans: DynamicJava, BeanShell, and JPython. For information on the scripting languages provided, see DynamicJava at `http://www-sop.inria.fr/koala/djava/`, BeanShell at `http://www.beanshell.org/`, JPython at `http://www.jpython.org/`.

We barely got you into Eclipse. Eclipse supports CVS (check out the **Team** submenu). Eclipse provides code refactoring features that allow you to rename classes and methods with automatic update of all affected source. Eclipse provides a feature to "externalize" strings (which takes all string constants out of a module and makes them into properties references, allowing for easy internationalization). It is a powerful Java development platform.

10.7 RESOURCES

NetBeans. NetBeans has some very extensive online help. There are also some very good Web-based documents, including the user guide which can be found at `http://usersguide.netbeans.org/`. Of particular value is the Getting Work Done guide at `http://usersguide.netbeans.org/gwd/` which describes itself as "a more detailed introduction to the IDE than available in the Getting Started tutorial."

Support for NetBeans, as with many other Open Source projects, happens online. There is no toll-free number to call. Instead you subscribe to an e-mail list; all messages sent to the list are then forwarded to everyone on the list. Anyone can respond, and you are encouraged to respond too, to share what you know with others. The NetBeans developers are often the ones who answer

the most difficult questions, but lots of times answers come from others who have just made it a little further up the learning curve than you.

To subscribe to the nbusers list, send e-mail to `nbusers-subscribe@netbeans.org`. You might want to create a special mail folder for the constant stream of messages that you'll get from nbusers. We've seen about 15–20 messages per day, on average, over the past year. You don't need to read them all, but as you scan the subject lines, see if there are ones that you might be able to answer. If you want others to reply to your requests for help, it would only be fair for you to do likewise. For a directory of the many e-mail lists related to NetBeans, go to `http://www.netbeans.org/community/lists/`.

Eclipse. *The Java Developer's Guide to Eclipse* by Sherry Shavor et al. (ISBN 0321159640, from Addison-Wesley Professional) is an excellent book on the Eclipse platform, particularly from (as the title suggests) the Java developer's point of view. Eclipse is, however, more than just Java. It is designed to be an "IDE Factory," providing a framework for almost any task that involves an edit/compile/deploy kind of lifecycle.

Part II

Developing Business Logic

Chapter 11

Balancing Acts: An Imaginary Scenario

In this chapter, the authors wax poetic on practical software development methodologies, and attempt to introduce a simple, or maybe simplistic, example application for use in future chapters.

11.1 What You Will Learn

We want to share with you some thoughts on practical software development. We are not covering anything specific to Linux or Java in this chapter; there will be no commands to type or syntax to learn here. You may skip ahead to the next chapter—but at your own peril. Those who know and use good software process won't need this chapter, but many programmers don't fall in to that category. In fact, some who think they are using good process may be using too much or too little for their actual situation.

If you are relatively new to the corporate world, or have only worked for one or two companies in your career to date, you may get a taste of how software is done in other corporate cultures. If you are a seasoned programmer with many such experiences, see if this doesn't sound all too familiar.

You will also see the requirements for a simple budget application that will be used in succeeding chapters. It has little to do with real budgets, but lots to do with a simple application that we can use to demonstrate various Java technologies.

11.2 Statement of the Need

Financial planning is something that everyone does. The basic tool of financial planning is the budget. But unlike the home budget, the budget of a large corporation is managed at multiple levels within the organization. Sure, at some level, the board, the CEO, and the CFO decide that "we will spend X million dollars for operations for this quarter," but that is the start of the process, not the end.

From there, the feeding frenzy of middle management begins. And keeping track of things becomes an important aspect of financial control and corporate governance.

Realistically, any large business will already have processes and tools in place that meet this need. We are not expecting that what we develop here will be anything that a business will actually adopt to manage this process. Rather, our goal is to illustrate some methods of software development that actually work to build real Java application. The outline of methods and code could be used to address many classes of enterprise software.

11.3 How to Develop Software

There is a science of Software Engineering. The development of software can be made an engineering discipline, with mathematical rules and metrics of success. Every aspect of a complete system can be worked out in detail beforehand so that you know, well before a line of code is written, what the outcome will be.

That's not what we're talking about here.

We are talking about software development not as it happens at NASA, medical device companies, and in factories where nuclear missiles are made. In those contexts, the potential costs of error are extremely high, ranging from the multibillion dollar loss (and public embarrassment) of crashing a spacecraft into Mars, on through having heart pacemakers fire off incorrectly, right to ending life as we know it on this planet. In such cases, no matter how much the correct

software costs, you pay it because the consequences of not doing it perfectly are far too high.

> **TIP**
>
> Our discussion is not meant to be scholarship on the topic of software development methodology; instead, it is meant to show simple, basic processes that can bring a reasonable amount of control to a software development project. These steps are, to name a few, requirements gathering, specification, object analysis, database design, development iteration (code, unit test, repeat), and so on.

But most of us who write software do not deal with such consequences. Most of us are keeping track of purchases and payments. We're recording production data. We're tracking manifests and updating inventories. We are the great unwashed mass of MIS software developers. Here we, too, want to do it perfectly right. But every time we go to management and tell them how much it will cost and how long it will take, the little "mass layoff" vein throbs in their foreheads. We are always being told to do it faster and cheaper. And so we find ourselves, again and again, tilting at the windmill of quality.

So where does that leave us? When we go to management with the textbooks of software engineering, they either laugh or scowl. Clearly, the money people are not prepared to support the cost of doing it right. So what do you do? The best that you can. The one thing we can tell you for certain is that the formula for success is not "start writing code and trust to luck."

It is fair to say that even the minimal software development method should include the following steps:

- Requirements gathering
- Use case specification
- Class discovery and problem domain decomposition
- Technical requirements specification (architecturing)
- Testing
- Code and release management
- Production and operations support
- Bug and enhancement tracking

This list, when done in that order, has been referred to as the classic "waterfall" model—each step is done in its entirety (or largely so) before proceeding on to the next step.

Or at least that's the ideal which programmers have often pursued.

The problem is that the process involves people, and people, especially those responsible for the requirements, a) are sometimes unimaginative and 2) keep changing their minds. They start out with some requirements, based on what they think they're going to need. But they just aren't imaginative enough to think of how terrible their system will be for the average user. They also keep changing their minds as to what they want.[1]

The "iterative" approach has been tried as a way to address this problem. Rather than wait for all requirements to be spelled out perfectly, with the iterative approach you jump right in with what you do know, build that, but expect changes to come. The sooner you get a working product or prototype into the hands of the users, the sooner you'll get feedback on what works, what doesn't, and what is really wanted ("what works" is used here not in the testing sense, but in the usability sense).

Note, however, that in the iterative approach, one still gathers requirements, develops designs for the code and the tests, develops them, tests (and fixes) the code, and releases it. It's just that one does that on a much smaller and more rapid basis. You get something runnable sooner, and continue to modify it.

Some people will complain that this makes for more expensive rework, but we (and others) would disagree. You are refining the process. Your reworks are less expensive than if you went to the work of building the entire system only to have some key requirement(s) change—there can be a lot more "wasteage" there.

Be aware, however, that the iterative approach is *not* just "whipping the horses to run faster." It is not just the waterfall model run at high speed. Rather, it is using the early iterations of the product as a sort of a "living" requirements specification, one that you can show to people and that they can try out, in real-world scenarios, and on which they can give feedback. Don't expect to be able to compile complete requirements, but don't give up on talking to your end

1. Did you notice that we tried to hint at that ever-enjoyable mid-project shifting of requirements as we went from a) to 2), changing our numbering scheme midway? Minimal humor, admittedly, but if you've lived it, you understand.

users and other stakeholders either. Requirements are still key to delivering a solution.

So with either approach, you'll start with requirements. Let's look at the art of requirements.

11.4 What Makes a Good Requirement

A good requirement is one that states a need but not a solution. Sounds simple, but it's easier said than done—especially with solution-oriented technical types.

A typical first cut at a requirement might be something like "Our budget application should store its data in the database." While it sounds reasonable, it is really a solution posing as a requirement.

The first step in refining such a requirement is to ask the simple question: "Why?" The answer we're looking for is not "Because we've paid so much for our database software," nor is it "Because we all know SQL." Rather, it should be something dealing with reliability, fault tolerance, the need for transactional integrity, and so on.

Sometimes you may have to ask the "why" question more than once, to refine the requirement(s). "Transactional integrity" is, in a way, a solution. You could ask, "Why do we need that?" For some projects it may be appropriate to ask this, because there may not be a real need for it after all.

But don't overdo it. Push any requirement in a business setting far enough, and you could get something like "To make money." That's not a helpful requirement. You've gone too far. Part of the art of requirements is recognizing when to stop asking why.

A more detailed description of a requirement is that it should be SMART—Specific, Measurable, Attainable,Repeatable, and Testable. Consider the following.

A common concern among users of almost any application is that it be "fast" or "responsive." While we can sympathize with the concern, it will need some refinement before it can be considered a (good) requirement. Applying the "Specific" and the "Measurable" aspects of SMART, we need to specify what constitutes "fast enough."

We can try "No button press in the GUI will delay more than .1 second before providing some evidence of activity to the user, or more than .5 second before completing its operation."

Sounds more formal, and more specific, but is it realistic (i.e., attainable)? If the "button press" is one that updates a database across a network, what effect will network traffic have? What about the size of the operation? If the button press starts an operation that is dependent on the size of some data set, what's the largest it could be and how long will that take?

Depending on how obsessive you or some colleague will be in enforcing these requirements, you would do well to add a few "weasel words" to give you some flexibility in the requirements. Phrases like "on average" or "most" will help. Notice, though, that such words are also the cause of much ambiguity, working against the "Specific" and "Measurable" aspects of good requirements. Use them sparingly, if at all.

We should also consider the "testable" aspect of our requirement for speed. Will we be able to measure this? Can we do so repeatedly? Consider the effect of network traffic on response times. Under what network load will the tests be done and the real usage occur? If you want to test under "normal" network loads, how can you control this (for the sake of repeatability)?

It really is an art to craft good requirements. Moreover, a good requirement for one organization may not work well for another. Some teams, groups, or companies want to be very precise in their use of requirements, viewing them almost like legal contracts for what will be delivered. Such requirements, however, would be greeted with derision in other, more informal, organizations. It's not that the one will produce good software and the other garbage (well, they might). It's more a matter of style. Excessively formal organizations will drown in the details and spend way too much time (and money) arguing over the minutiae of the requirements. Overly informal groups will get sloppy with their requirements and not reap the benefits of building the right thing the first time. As is so often the case in life, the answer lies in striking a balance between two forces, one pushing for exactitude and the other pulling you to get going and do something.

So let's keep going.

11.5 Whom to Ask for Requirements

There are many people to ask about the requirements for a software project or product. Ask yourself the following questions:

- Who is going to use the software that you develop?

- Who is going to use the data that comes from the use of the software (i.e., who will read the reports generated from the data collected either directly or indirectly from the running of the software)?
- Who is going to support the software and who will support the machines on which it will run?

All these people can be considered "stakeholders" in the project.

So where do you start? That's a political more than a technical question. Start with your boss and with whoever is the major backer of the project. Then ask your customers. For in-house IT projects, the "customers" are usually very accessible; for software products, the customer's point of view may need to be represented by marketing and/or customer support people who have had direct contact with the customer base.

11.6 Requirements for the Budget Application

Let's take a look at how such requirements might evolve. We'll look at the situation through the eyes of a fictional IT guy named Bob.[2]

11.6.1 Monday Morning, 10 A.M.

Bob gets called in to the office of his manager, Ellen. The conversation goes something like this:

Bob: Yes, Ellen, you wanted to see me?

Ellen: Come in, Bob. Yes. We're just about to enter another budget planning cycle. We've got to propose our next year's budget to the VP by the end of the quarter, and I got to thinking . . .

Bob: Uh-oh.

Ellen: . . . on my way to work today, I got to thinking that we ought to be able to develop a software tool that would help us do a better job of this process.

2. We're avoiding giving Bob a title because titles vary so much within our industry. Call someone an analyst and it may mean that they never code. Call someone a programmer and it may mean that they only code and never deal with requirements or even designs. Some use those terms interchangeably. We'll just call him an IT guy.

Bob: We've used a spreadsheet these past few years to do our budgets. You want us to develop another spreadsheet application?

Ellen: No, I want a whole new application.

Bob: You want us to reinvent the spreadsheet?

Ellen: No, I want something simpler and more specific to the budgeting process.

Bob: Tell me more. What are the key features that you see in this application?

Ellen: Well, first of all it needs to be able to work concurrently with all the users. With our spreadsheet, we'd have to take turns with the data entry or we'd risk loosing each other's changes.

Bob: It may just be that we're not using our spreadsheet's advanced features. Shouldn't we investigate that first?

Ellen: No, I'd rather have us invest our time in building the tool we know that we need. At the end of the day your investigation may only show that we still need the tool, and by then it might be too late to build it.

Bob: I hear you saying that the deadline is rapidly approaching.

Ellen: Yes—I want to be able to use it for the budget planning at the end of this quarter. How long do you think it will take you to build it?

Bob: Build what?

Ellen: Haven't you been listening? The budget tool!

Bob: I know that you mean the budget tool—but you haven't really given me enough requirements upon which to base an estimate. Tell me more about how you envision this tool being used.

Ellen: Well, in the past we've taken last year's numbers and just bumped them up by a few percent. Then we look at each category and tweak them. I want a different approach this year. I'm going to take our department's budget, give it a bump, then assign a chunk to each of my reports. I want you to take those discretionary dollars and spell out how you would spend them.

Bob: Shouldn't we be providing you with estimates of what we need for the coming year, rather than you telling us what we have to spend?

Ellen: In theory, perhaps so. But in practice we can only grow the budget by so much. I'd rather skip the charade and jump right to allocating the dollars we will likely be able to spend. Then as the year progresses, I'd like to use this tool to track our spending against this plan.

Bob: But isn't that why we have that big SAP application?

Ellen: Have you ever tried to use it?! Please! The CFO thought it looked great—and on paper it did. But that user interface makes it almost impossible to be productive. And it's as slow as molasses.[3]

Bob: But back to this new application . . . I'm assuming you'll want a GUI on this?

Ellen: Of course. Give it a standard, simple GUI. Something like this. *(She begins to draw on her whiteboard.)*

For any given department there will be a "pool" of money. Those dollars are displayed and can be subdivided into smaller pools of money by creating subaccounts.

But as the money is subdivided those new accounts and associated dollars should become visible by others. And as dollars are spent during the year, we'll want to track those dollars, so those amounts should be visible, too, and subtracted from the overall pool of available dollars.

Bob: Wait . . . back up. What needs to be entered to subdivide an account?

Ellen: The user just picks an account, then chooses to subdivide it, entering the amount to put in each account . . . or even just a percent of the larger pot of money.

Bob: So if he picks one account to subdivide, does it split into two, or three or how many?

Ellen: Let the user choose, but maybe two as a default.

Bob: OK, but we may need to take a harder look at that interaction.

Ellen: So how long will that take? Can you have it ready by the end of this month?

Bob: I'd like to try the "spiral" approach on this project. I can have something for you by the end of this week— from which you can tell me if I'm heading in the right direction. It will just be a beginning, but you'll be able to see something run. By the way, is this tool only for our group?

Ellen: For now it is, but I could see other departments wanting to use it some day. Who knows how far it could go?

---

3. Remember, this is a fictional account. We are providing justification for why they can't use the corporate application. Anyone's use of such a tool can be less than optimal, reflecting more on themselves than on the value and usability of the tool.

11.6.2 Back at His Desk

Bob is now back at his desk pondering the conversation he had with Ellen. "These are not like the requirements we learned about in my software engineering courses," he muses. "I've got that sketch of the UI and a brief description of its functionality. But there seem to be so many unanswered questions."

So what is Bob supposed to do? He could go back and try to get more "face time" with Ellen, and ask lots more questions. Sometimes that's a smart thing to do. Other times such repetition is seen as annoying and a sign of a slow-witted analyst, never mind how obscure the initial discussions were or how many times someone changed their mind about what they want. You will have to judge each situation as you encounter it. At some point, though, you have to deal with whatever information you've been given, and try to make the best of it.

So where do you turn? The next best things to do are to begin to document the requirements as you understand them, to prototype a solution, and to start getting buy-in from other stakeholders. Each of these activities may help bring out more requirements, but that's not a bad side effect.

11.7 Documenting, Prototyping, and Stakeholder Buy-In

Once a project is started, the design must be documented. A prototype may be built to validate and refine the design. Finally, everyone with a stake in the success of the design has to be brought up to speed and needs to agree on what is to be built.

11.7.1 Documenting

After such a conversation, it's smart to try to get your thoughts down on paper as soon as possible. Some of what gets said will fade with time, so work quickly to capture what you can of the requirements that were spoken. Even if you have to leave lots of blanks, keep moving and get as much of the major requirements written down as you can, even if they don't sound very formal or polished. Then go back, revise and edit your statements, filling in the blanks where you can. Sometimes you will need to ask others to get the answers to fill in the blanks. Other times you can use your own judgment and initiative to provide an answer. Out of this process with its subsequent rewrites will come the requirements document.

Some organizations are very formal in their understanding of requirements. They will have company-standard formats which you must follow. But there is no magic format that will make for good requirements. It really all comes down to content.

Here's an informal list of the requirements for the budget application, based on the conversation between Bob and Ellen.

Features:

- Starts with a single lump sum of dollars.
  - How does this first sum get entered?
- Each dollar amount is associated with an "account."
- Any account may be divided into two or more subaccounts.
- The dollar amount associated with a subaccount is specified either in absolute dollars or as a percentage.
  - What if they don't add up?
  - Can the user mix $ and %?
  - Can the user leave the last subaccount's amount blank for "remaining dollars"?
- Tracking of the dollars—not enough info, so not in first prototype.
- Multiple users will have access to the data.
- Concurrent use is allowed and supported.
- Short development time, limited resources.
- Has a graphical user interface; earliest versions may be command-line and terminal interaction.

Not all requirements will be easily forthcoming; not all can be traced back to an exact quote from the previous discussion. Other requirements will need to be inferred from the discussion or from department "culture," or come from your own judgment:

- Platform: "any" PC in Ellen's department—but her developers are all using Linux platforms.
- Future platforms: "any" PC in the company means any Windows, Linux, or Mac OS X.
- Reliability: once entered, data is never lost.
- Maintainability: the application must be easy to maintain.

- Interoperability: there's no requirement to interoperate with any other software but here's an idea for a future version: export/import into CSV format for spreadsheets, and/or XML format for future expansion).
- Response time: "reasonable" interactive speed; subsecond response when entering new accounts and values, so that the user can type quickly and continuously; waiting, if it occurs, should only be at button presses, not between data entries.

11.7.2 Stakeholder Buy-In

Stakeholder buy-in can be another important part of a software project. As we discussed in Section 11.5, stakeholders are any of those people who are touched in some way, direct or indirect, by this software project.

For this simple budgeting program, there will be few stakeholders—it will largely be Ellen and her direct reports. The system will not likely be a large drain on computing resources, so system admins don't need to be brought in at this point. If and when the project expands to include other users across the network and across the enterprise, then the system administrators should definitely be included. There will be few reports from this first cut of the project, and what few there are will only be read by Ellen and her direct reports, so again, there are few others that need to be consulted as stakeholders.

The idea at this stage is to listen to other points of view—those of your stakeholders—to get a different perspective before charging headlong down one avenue of development.

It's not that you will be able to satisfy all points of view—it can be a worthy goal, but it is often unattainable. Rather, you need to hear from all those involved since your software will affect all those people, and understanding something about how it will fit into their roles and daily tasks will help you make better tradeoffs and design better software. It will likely uncover previously unseen requirements. It also has the political benefit of those people knowing that you cared enough to listen to them before sending them a finished solution. It increases the likelihood that your software will be seen as a help, not hinderance.[4]

---

4. As engineering types it is difficult for us to understand and appreciate the importance of this, but in many ways these personal, political, and psychological factors are much more important to the success of a project than are technical choices. It has taken us years to appreciate

11.7.3 Prototyping

Prototyping can be an effective way to carry on the discussion of both requirements and user interface design. Given only a hypothetical or abstract description of some software, it can be very difficult for people to imagine what the implications of its use will be. A simple prototype can immediately bring the discussion down to the concrete; people can point at things and say "I like this" and "I don't like that" and "How would I do so-and-so" and then see whether or not it would work. Sometimes, ideas that sound great on paper turn out to be pretty poor ideas when realized. Prototypes can help you discover that quickly and easily.

One very useful but inexpensive prototyping mechanism can be HTML—that is, creating Web pages. Simple static HTML can be fast and cheap to build, but can begin to approximate what the user interaction will look like—especially for, but not only for, Web-based solutions. It may not be an exact replica of the final product, but for a first step it can really get the discussion moving.

If the UI is too complex for a Web page mock-up, you can still use HTML for prototyping by getting images (screenshots) of what you want your final solution to look like and then making these images clickable on Web pages, to simulate some simple user interaction with hyperlinked image sequences.

The idea is to get something "tangible" in front of people as soon as possible, to further the discussion in a way that written descriptions never can. ("A picture is worth a thousand words.")

Once you've built a prototype, shop it around. Hold informal meetings where you demonstrate the basic functions to stakeholders. We recommend, as much as possible, meeting with one group of stakeholders at a time. That way you can keep your conversations focused. If you have two different stakeholder groups represented and their expertise and interests are wildly different, you'll be boring ½ the participants all the time. Even if their expertise is similar, you may have groups with competing or conflicting requirements. While you need to understand such conflicting requirements and eventually come to some detente, this meeting is not the best venue for settling those issues; it would more likely simply scuttle your meeting and void any value from it.

---

that Dale Carnegie is as important to the software designer as Yourden or Booch. Your users need to be your friends if you want to succeed.

After each meeting, review your requirements and see what more you need to add. Likely at such meetings, you'll begin to get requests for new features.

You have, in fact, begun the iterative process. Even the most bare-bones prototype that may only consist of a sequence of pictures is a first cut of your product. The sooner you can get to a running version, the sooner you will be able to respond to stakeholder suggestions by adding real features.

11.8 Review

A good requirement is one that states a need but not a solution. Your first step is to uncover the needs, while listening to everyone's solutions. These requirements will develop into feature descriptions. These should be documented and then prototyped. The prototype, which is in effect the first release of your product, can then be shown to various groups—stakeholders—as a way to elicit their feedback. This feedback should begin to factor in to what you will build, so now you need to move quickly on to building the real product; do not get stuck enhancing the prototype.

11.9 What You Still Don't Know

Writing good requirements is as much art as it is science, and it involves political science as well. This is not something easily taught in a book, but learned through hard experience.

11.10 Resources

One of the original purposes of the World Wide Web was to allow researchers to share their results. So, you should be able to search the Web for requirements documents from various projects for examples of requirements specification. As with any Web search, remember to consider your source. Just because someone has posted a requirements specification or a template doesn't make it a good example.

Here are three examples that we found on a single simple Google search. They may still be there by now.

- http://www.progsoc.uts.edu.au/~timj/thesis/web/srs.html
- http://www2.ics.hawaii.edu/~johnson/413/lectures/5.2.html
- http://www.cc.gatech.edu/people/home/tomoyo/rocky-axel.1.doc

For those who are serious about their software development process, the Capability Maturity Model for Software from the Software Engineering Institute at Carnegie Mellon University is the standard. Visit their Web site at http://www.sei.cmu.edu/cmm/.

If you would like to know more about the spiral approach to software design, you might want to start with the seminal paper on the topic, "A Spiral Model of Software Development and Enhancement," in *Computer* 21, no. 5 (May 1988), pages 61–72.

To see how the director of the Software Engineering Institute views the spiral approach, check out the short and readable introduction at http://www.dacs.dtic.mil/awareness/newsletteres/technews2-1/disciplined.html.

Another good look at the spiral, or iterative, approach can be found at http://www.stickyminds.com/se/S3420.asp which has a hyperlink for a PDF file of a paper by Philippe Kruchten of Rational Software. The paper covers some pitfalls common to the first uses of the iterative approach; worth the read.

A great survey of key papers on three major approaches—spiral and related topics (including newer work by Boehm), aspect-oriented programming (AOP), and the rational unified process—is at http://www.rspa.com/reflib/PrescriptiveModels.html.

11.11 EXERCISES

1. Write requirements for a simple word processor or spreadsheet. Start with some obvious functionality. Add only enough "bells and whistles" for it to be usable for beginners. Show this list to others, especially people familiar with similar applications. What features do they find missing that are important to them? How quickly does your list expand? What might you do to limit the size and the rate of growth of the features list?

2. Discuss the requirements for your application with someone who has no experience with a similar product. How difficult is it to get useful feedback? Now show them (the simple features of) a working spreadsheet or word processor, as if it were your prototype. Does the conversation change? In what ways? Is the feedback now more or less useful than before they saw the prototype?

Chapter 12

Analysis and Design: Seeking the Objects

In this chapter, we will present the barest outline of a software development methodology. For some readers, this will be simplistic and unsuitable. In our experience, however, there are many businesses out there with very small development teams that have very little software engineering experience, even though they have considerable technical and programming skill. Our goal here is to present a bare minimum of analysis and design method, so that we can be sure we have a common basis for discussing the issues of object-oriented analysis and design.

12.1 WHAT YOU WILL LEARN

In this chapter you will learn a very simple method for object discovery and a simple method of documenting this process.

12.2 Facing the Blank Page

So, you have some requirements. Maybe you even have some UI prototypes. How do you turn that into an object-oriented design for Java classes? How do you confront the paralyzing blank white of your whiteboard, terminal session, or easel?

The simplest way is to start with real-world objects. Stop thinking about everything you have read about object-oriented programming. Instead, ask yourself, "What are the real objects involved in this problem?"

In our case, the more you look at it, the simpler it gets. For the moment, the only real objects we have are people—the users—and accounts, that is, named pools of money. We know that users get accounts from "above," and that they may break those pools down into subaccounts, which they may own or delegate to other users.

At the broadest level, then, we seem to have two "classes" or types of real-world objects: Accounts and Users.

12.3 Using CRC Cards

So, we need two classes. But what goes into those classes? How do we go about putting the substance into this simplistic framework?

In their now (semi)famous paper presented at the object-oriented programming conference OOPSLA in 1989, Kent Beck and Ward Cunningham introduced a simple, practical design tool for object-oriented design based on a simple, practical 3x5 file card. The CRC cards for our classes are shown in Figures 12.1 and 12.2.

But we are getting a bit ahead of ourselves. These CRC cards are an end product of analysis. They are the starting point for coding. Let's talk a little bit about what is on these cards and how we came to that content.

12.4 Finding the Objects

The basic technique for doing OOA[1] with CRC cards is to start with a stack of blank cards. Assemble a design team (this may be one person, or this may

1. (object-oriented analysis)

Account	
a pool of dollars	
members	**collaborations**
* name - a String * owner - a User * amount - an SAMoney object * children - an ArrayList (of Accounts) * parent - an Account	* persistence (CRUD) * User

Figure 12.1 Account CRC card

User	
someone who manages budget dollars	
members	**collaborations**
* name - a String * home - an Account	* persistence (CRUD) * Account

Figure 12.2 User CRC card

be dozens).[2] The first step should always be the nomination of the real-world objects. Don't edit or critique at this point. If someone says "computer" as an object, write "Computer" on the top of a card and put it on the table. If someone says "Manager" write it on a card and put it on the table.

2. It is fun to gloss over such a complex topic with a single sentence! Obviously, the composition of a design team is a complicated matter. At the very least, a design team must include a representative from the programming team and a future user of the system. On small, simple projects, that may be all you need. On more complex or mission-critical systems, there will have to be additional representatives, such as people from Operations, Support, Training, Quality Assurance, and so on.

To take our example, suppose we have the following list of CRC cards after such an open brainstorming session:

- Database
- Capital Account
- Current Account
- CEO
- Computer
- CFO
- Director
- Keyboard
- Manager

Where do you go from here? Let's articulate a general principle.

The first principle. If we could teach a programmer only one thing about software design, it would be this idea: less is more. Or, to quote Antoine de Saint-Exupéry: "Perfection is achieved not when nothing can be added, but when nothing can be taken away." Or, to put it yet another way, always use the KISS[3] principle. The best object design is the smallest possible number of classes that model the real objects and meet all the requirements.

You are seeking simplifying abstractions.

First of all, all the objects that represent technologies or implementation details should be removed. In our list, this would include "Database," "Computer," and "Keyboard." While it is likely that all three will be involved in the final product, they are not objects in the problem space. There is no theoretical reason why an OOA session cannot produce a manual, noncomputer solution. It is a common tendency to leap from problem analysis directly to technical solutions. "We can write that in Java," "We can store those in Oracle," "That could be an XML file." Statements like these are to be avoided at this stage. Those are details about the implementation. You haven't got a design to implement yet!

As we said, you are seeking simplifying abstractions. The next step, after culling cards that do not represent real objects in the problem space, is to group together the cards that have any attributes in common. If we look at our

3. An acronym for: *Keep It Simple, Stupid!*

remaining cards, we can quickly see that we have two cards that are accounts: "Capital Account" and "Current Account." These are both pools of money. Put them on top of one another on the table. Likewise, it is fairly obvious that "CEO," "CFO," "Director," and "Manager" are all people. Put them together on the table.

Remember that we are looking for simplifying abstractions. The grouped cards should all be obviously variant types of a generic class of objects. In our example, the one is a stack of Accounts, and the other is a stack of People, or, as we will call them, Users. Create new cards for these generic classes. Make a card with "Account" at the top and put it above the first stack. Make another card with "User" at the top and put it above the second stack.

There are two ways that this might simplify your design. For now, all cards below the abstract cards are "on probation." We are going to move on to define the attributes (data) and methods (behavior) of our abstract classes. If the abstract class can handle all use cases without having to treat any of the more specific classes differently, then the specific cards are discarded. If not, then all functionality that is common across the more specific types will be put on the abstract class card, and only those data and behaviors that are different will be put on the more detailed cards.

In the first case, the simplification is a reduction of several potential classes to a single class. This is always a good thing, when it is possible. In the second case, you are identifying potential inheritance relationships.[4]

12.5 Finding the Methods and Attributes

The next step is to start identifying the data and behavior that characterize your classes. Always put such items on the most abstract class first. The only time to add an attribute or method to a more specific class is when it applies to that class and only that class—in other words, only when it represents a difference between the general case and the specific case.[5]

---

4. We'll talk more about that later in the book. As it happens, all of our simplifications in this chapter are examples of the first case.

5. In complex cases, you may find an attribute or method that applies to several, but not all, of the specific cases. In such a case, a new abstract class below the main abstract class, but above all the specific classes that share that attribute or method, may be called for.

12.6 ESSENTIAL AND NONESSENTIAL

So far, we have walked you through a very simple example, and we have made sound choices at every step. In more complex cases, even the best of us will make mistakes. We will head down blind alleys. We will group things together that might belong in separate abstract categories, but should, perhaps, share an interface. These are not so much errors as judgment calls, and skill at recognizing them and making the correct decisions comes only with experience.

For now, the most important questions to ask include:

- **Do I need this class?**

  We are often tempted to create too many inherited classes. When we seek more generic, higher level abstractions, it is often possible to use only the more abstract class. Of course, it is possible to carry that tendency too far. If your methods contain a lot of "if's" to handle various subtypes, that might be a case where you should inherit and overload the method.

- **Should I get functionality by inheritance or composition?**

  Inheritance should be reserved only for cases where a class is a more specific variety of the base class. For example, you might have a Person class, and then you might have Employee and Customer classes inherit common attributes and methods from Person. This is frequently called an "is-a" relationship, as in "A User is a Person." If your proposed inheritance relationship makes sense phrased that way, it might well be a good candidate for inheritance.

  Composition is when you use a class as an attribute. To extend our example, you might have an Address class. You might be tempted to have Person inherit from Address. But a Person is *not* an Address. Try it: "A Person is an Address." Nope. Instead, you should just have an instance of the Address class as an attribute of Person. Such a relationship is often called a "has-a" relationship, as in "A Person has an Address." If the relationship makes sense phrased that way, it is a good candidate for composition. Another way to recognize that you've wrongly used inheritance is if you end up having a radically different class inherit from the same base class. For example, suppose you have a class, Building. Would it make sense for Building and Person to inherit from Address? Are Buildings and Persons more specific instances of the same general type of thing? No, they are not. Building and Person should get Address functionality by composition.

- **Does this attribute or method belong here?**

  If you find yourself specifying nearly identical methods in more than one class, this should make you ask if the classes should have a common base class from which they should inherit, or if there should be a new unrelated class that they all share by composition.

  If the functionality is the same for a set of classes, and the classes are specific instances of a more general type, the method should be on the general class. For example, a `changeName()` method should probably be on Person, not on Employee or Customer, because the functionality is the same for all three classes. By contrast, a `changeEmployeeNumber()` method should be only on Employee. It should not be on Person, because not all Persons are Employees. There may also be methods that are common to both Employee and Customer types, but are radically different in implementation. For example, a `changePassword()` method might change a password in a system-wide LDAP server for an Employee, but might just change a record in a Web site database for a Customer. This is easily done by writing separate methods in each class.

  But should you add a `changePassword()` method on Person? If you want to be able to call the method when treating either a Customer or an Employee as a Person, then you should. But you don't have to implement the method on Person. You can declare `Person.changePassword` as abstract, and then, if you call the method on a Person, it will call the correct method based on what type of Person (Employee or Customer) the Person is. Note that if a class contains any abstract methods, the class itself must be declared abstract and it cannot then be instantiated. Also note that this is often best accomplished not through abstract classes, but through interfaces (see Eckel, pp. 321–322).

These are by no means the only considerations that come to bear on what classes to create and how to arrange and implement them, but they do represent a good start. They are a foundation on which you can build best practices out of your own experience and environment.

Whole books have been written on the topics of object-oriented analysis and object-oriented design. CRC cards are only one part of an array of techniques that can be applied to OOA/OOD. The Unified Modeling Language (UML) is popular in many MIS circles. UML consists of a variety of different diagrams which are used to model parts of an object-oriented design. They are:

Am I Mature? Or Are You My Mommy?

Let us point you at one more business buzzword link. Even though we think this particular site and their work are being ill-applied by many well-intentioned IT managers, there is still a great deal of value in the Carnegie Mellon Capability Maturity Model (`http://www.sei.cmu.edu/cmm/`). At the very least it provides an objective way to assess the level of process sophistication you have in your organization.

The CMM defines five levels of maturity:

1. Initial
2. Repeatable
3. Defined
4. Managed
5. Optimizing

If we may grossly oversimplify (and why should we stop now?), "Initial" means you do things differently every time. You just make your best guess about what the right thing to do is, and you do it. "Repeatable" means that you have hit upon a method that appears to work, and you use it consistently. "Defined" means that somebody has written it down. "Managed" means that the process is actively maintained and supervised in an effort to adapt it to changing circumstances. "Optimizing" means that measurements ("metrics") are made that objectively assess the process, and ensure that continuous improvement takes place and can be so proven.*

What we have shown you in this chapter probably falls in the Repeatable category, a long way from the engineering and management nirvana of Optimizing.

---

* The problem that seems to come up with this system is that very bad processes may be very mature and very good processes may be relatively immature. Obviously, however, an Optimizing process must be steadily moving towards the good.

- Class Diagram
- Sequence Diagram
- Collaboration Diagram
- Use Case Diagram
- Activity Diagram
- Component Diagram
- Deployment Diagram

Using the simple but effective technique of CRC cards can be a good place to start, but you may soon want to move up the OOA/OOD ladder to use tools like Umbrello[6] to make UML diagrams, and perhaps to use the whole UML toolset.[7] Many organizations that we know of will pick and choose various techniques and tools. No matter how far down the road of formal software engineering you go, you must at least make some effort to have a repeatable process that incorporates continuous improvement.

12.7 Analysis Paralysis

The catchy phrase "analysis paralysis" has become a cliché. (And how could it not, being so catchy?) What it refers to, of course, is the tendency to become bogged down in details; or the tendency to refuse to start implementation until you are certain that your design is "right."

This is where using a "spiral" development model can pay off. By doing frequent small releases, you can expose subtle design flaws at an earlier stage in development. Often, you can (to trot out another trendy term) "refactor" a small part of your design or implementation. If you have clean object interfaces, this can often be done with minimal disruption because a good object model hides implementation details within classes.

In most cases it is best, once you have the use cases and requirements, to proceed to a prototype object model and learn by doing.

6. `http://uml.sourceforge.net/index.php`

7. `http://www.uml.org/`

12.8 Real Software Engineering

Let's take a moment here and ask a fundamental question. Is this the best way to make software? And there is another fundamental, but subtly and importantly different question: Is this the right way to make software?

There are techniques and methods of Software Engineering that do approach the ideal of "zero defects." NASA uses such procedures for manned spacecraft. Coders for medical devices do likewise. The outline method we have suggested here doesn't come close to such methods. So, is what we have described the best way to make software? No, it is not. So why don't we all use those zero defect methods? That is easy to answer: cost. It is expensive. Virtually no MIS shop on the planet would be willing to pay the price it takes to get that stability and certainty. The price isn't just dollar cost, either. The Space Shuttle, for example, has computers that still use magnetic core memory, a technology that was old in the 1970s. Why? Because the restrictions imposed by their change control systems would essentially require the entire shuttle to be redesigned and retested if they made such a change.[8]

But this isn't an either-or. You do not have to apply either a full-fledged software engineering methodology, or use nothing at all. Instead, you have to apply some design, development, and maintenance processes that improve the probability of success and reduce the cost of failure. When we recommend version control, requirements gathering, use cases, and CRC cards, we are giving you a bare-bones set of methods that will help to write fairly successful software at reasonable cost in reasonable amounts of time.

To some of you, this will be old news. If you are at level 2 or above on the Capability Maturity Model (see the sidebar in Section 12.6), then you already have some process. But you would be surprised how many business out there do not even have source code control in place. To some of you, what we suggest here will be primitive compared to processes you already have. The point is, no one's level of control and process is "right" (to us, that means "cost-justified") for all cases. But using no method at all is a risk too great for any business.

8. An exaggeration to be sure, though maybe not as much as you might think, but you get our point.

12.9 CORE CLASSES

So, let's meet our core Java classes. Here they are, in all their glory (Examples 12.1, 12.2).

12.10 REVIEW

We have discussed a simple approach to object-oriented analysis and design through the use of CRC cards. The ideal outcome is a design with the smallest possible number of classes that model real-world objects while meeting all the requirements.

12.11 WHAT YOU STILL DON'T KNOW

We could list the names of a number of formal software engineering methodologies, but we won't bother. If this chapter has served as your only introduction to object-oriented analysis and software engineering, let's just say you have a lot of reading to do. But beyond that, there is something you need that is much more subtle and difficult to pin down: *experience*. The only way to get good at analysis and design is to do it. It helps to do it in conjunction with experienced people, because they can save you time and pain in acquiring your experience. This chapter is the simplest of foundations. The books give you knowledge. Experience gives you wisdom.

12.12 RESOURCES

Kent Beck and Ward Cunningham, "A Laboratory for Teaching Object-Oriented Thinking", in *OOPSLA'89 Conference Proceedings*, New Orleans, Louisiana, October 1–6, 1989. The special issue of *SIGPLAN Notices* 24, no. 10 (October 1989) is also available online at `http://c2.com/doc/oopsla89/paper.html#cards`.

More on the Capability Maturity Model can be found at `http://www.sei.cmu.edu/cmm/`.

Information on the Unified Modeling Language can be found at `http://www.uml.org/`.

Example 12.1 The `Account` class

```
package net.multitool.core;

import net.multitool.util.*;
import java.util.*;
import java.sql.*;

public class
Account
{
  private String name;                      // A name to identify this account
  private User owner;                       // The user assigned to this account
  private SAMoney total;                    // Total amt originally allocated to
                                            //   this account
  private SAMoney balance;                  // amt remaining unallocated to any
                                            //   subaccounts
  private Account parent;                   // The account which contains this
                                            //   account as a child
  private HashMap children;                 // The collection of subaccounts,
                                            //   by name
  private static Connection dbConn = null;  // JDBC connection
  private ArrayList payments;               // TODO: unimplemented
  private SAMoney unspent;                  // TODO: unimplemented

  /**
   * Create an account, with a pool of dollars to budget.
   * Use this constructor to create the master account.
   * Use createSub to create children of this account.
   */
  public
  Account(String name, User owner, String total)
    throws NumberFormatException
  {
    this.name = name;
    this.owner = owner;
    this.total = new SAMoney(Double.valueOf(total).doubleValue());
    this.balance = new SAMoney(Double.valueOf(total).doubleValue());
                                            // N.B. must not be the same object
    this.parent = null;
    this.children = new HashMap();
  }

  // Static that connects to the DB and either returns the top account,
  // or creates it for us.
  public static Account getTopAccount() throws SQLException {
  Account topAccount = null;

  dbConn = DriverManager.getConnection("jdbc:postgresql:budgetPro?user=mschwarz");
```

```
if (dbConn != null) {
  // We have a database connection.
} else {
  // We don't and we must create a top account.
}

return topAccount;
}

// Simple getter; returns the name.
public String
getName() { return name; }

// Simple getter; returns the total pool of money that this account represents.
public SAMoney
getTotal() { return total; }

// Simple getter; returns the balance.
public SAMoney
getBalance() { return balance; }

// Simple getter; returns the parent account.
public Account
getParent() { return parent; }

// Simple getter; returns the owner of this account, as a User object.
public User
getOwner() { return owner; }

// Census - how many children.
public int
size() { return children.size(); }

/**
 * Get to all the children, via an iterator.
 */
public Iterator
getAllSubs()
{
  return children.values().iterator();
}

/**
 * Create a new subaccount (i.e., child)
 * given a name and an amount.
 * The child is connected to the parent, and
 * the parent's balance is reduced by the amount
 * allocated to the child.
 */
```

```
  public Account
  createSub(String name, String amt)
    throws NumberFormatException
  {
    Account acct = new Account(name, owner, amt);

    // Reduce the parent's unallocated funds.
    balance = balance.subtract(acct.getTotal());

    // Connect the accounts to each other.
    acct.parent = this;
    children.put(name, acct);

    return acct;

  } // createSub

  /**
   * Looks up and returns the account with the given name.
   */
  public Account
  getSub(String name)
  {
    return (Account) children.get(name);

  } // getSub

} // class Accoun
```

The Umbrello UML modeller is an Open Source tool for creating the various UML diagrams. You can find it at `http://uml.sourceforge.net/index.php`. We also recommend their online documentation as a good brief introduction to UML and to Umbrello. It can be found from the main Umbrello page, or directly at `http://docs.kde.org/en/HEAD/kdesdk/umbrello/`.

12.13 EXERCISES

1. Imagine a public library. Carry out the CRC nomination process for a system to track library members and the collection. What list of objects do you come up with? What abstract classes do you find? Which did you discard and why?
2. Extend the purpose of the library program to include generating mailings to members with overdue materials. Did you add classes? Did you add methods and/or members? To which classes did you add them?

Example 12.2 The `User` class

```
package net.multitool.core;

import net.multitool.util.*;
import java.util.*;

public class
User
{
  private String name;
  private Account home;        // TODO: implement

  public
  User(String username)
  {
     name = username;
  }

  public String
  toString()
  {
     return name;
  }

} // class User
```

3. A new requirement is added. The system must allow for books, audio recordings, and movies to be checked out for different lengths of time. Did you add classes? Did you add methods and/or members? To which classes did you add them?