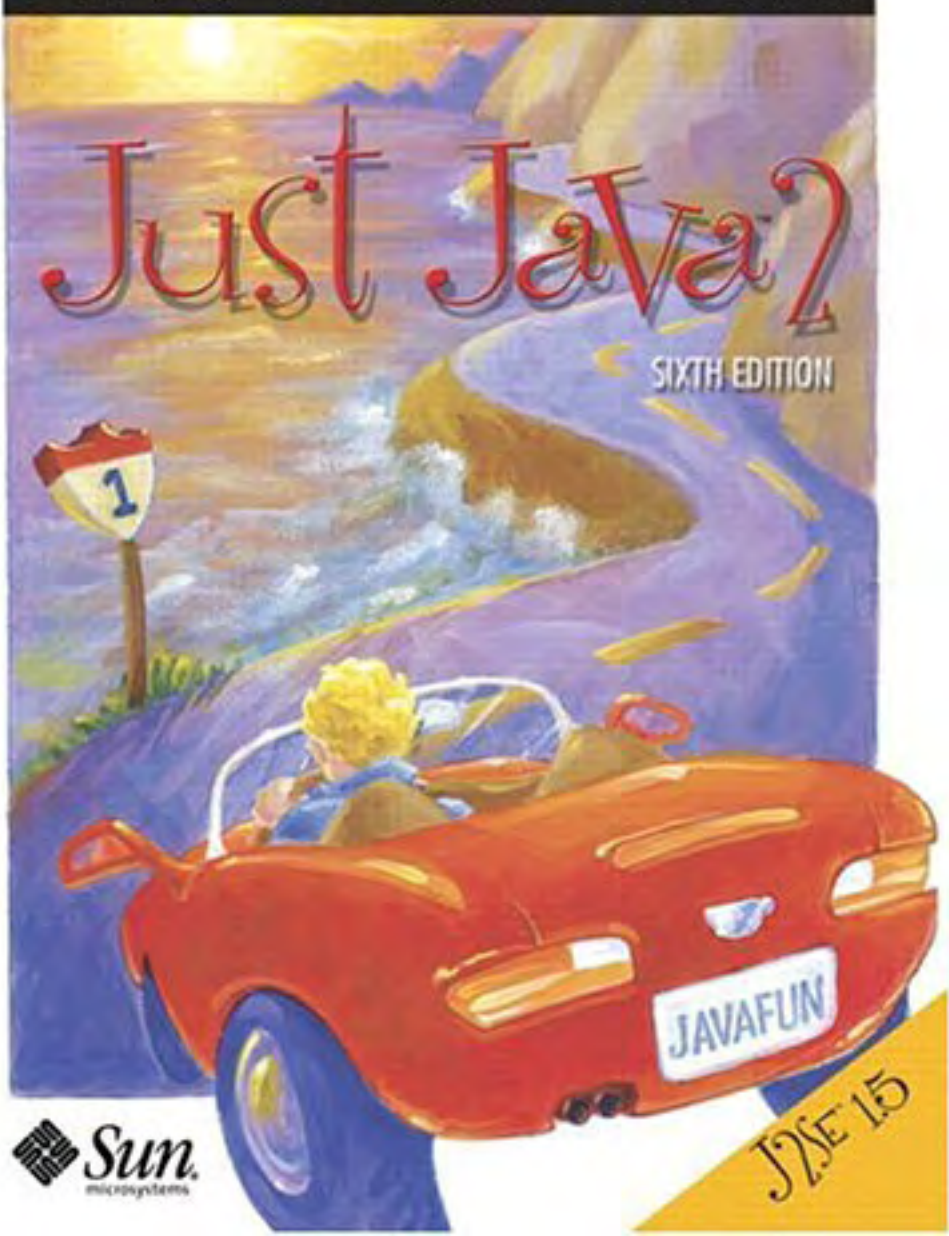
PETER van der LINDEN

Just Java 2

SIXTH EDITION



Sun microsystems

J2SE 1.5

Just Java 2 SIXTH EDITION
By PETER van der LINDEN

Publisher: Addison Wesley
Pub Date: June 21, 2004
ISBN: 0-13-148211-4
Pages: 848

The #1 introduction to J2SE 1.5 and enterprise/server-side development!

An international bestseller for eight years, Just Java(TM) 2 is the complete, accessible Java tutorial for working programmers at all levels. Fully updated and revised, this sixth edition is more than an engaging overview of Java 2 Standard Edition (J2SE 1.5) and its libraries: it's also a practical introduction to today's best enterprise and server-side programming techniques. Just Java(TM) 2, Sixth Edition, reflects both J2SE 1.5 and the latest Tomcat and servlet specifications. Extensive new coverage includes:

- New chapters on generics and enumerated types
- New coverage of Web services, with practical examples using Google and Amazon Web services
- Simplified interactive I/O with printf()
- Autoboxing and unboxing of primitive types
- Static imports, foreach loop construct, and other new language features

Peter van der Linden delivers expert advice, clear explanations, and crisp sample programs throughout—including dozens new to this edition. Along the way, he introduces:

- The core language: syntax, objects, interfaces, nested classes, compiler secrets, and much more
-

collections

- Server-side technology: network server systems, a complete tiny HTML Web server, and XML in Java
- Enterprise J2EE: Sql and JDBC(TM) tutorial, servlets and JSP and much more
- Client-side Java: fundamentals of JFC/Swing GUI development, new class data sharing details

Companion Web Site

All the book's examples and sample programs are available at http://afu.com.

Copyright

Library of Congress Control Number: 2004107483

Prentice Hall PTR offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact U.S. Corporate and Government Sales, 1-800-382-3419, corpsales@pearsontechgroup.com. For sales outside of the U.S., please contact International Sales, 1-317-581-3793, international@pearsontechgroup.com.

Acquisitions Editor: Gregory G. Doench

Editorial Assistant: Raquel Kaplan

Production Supervision: Julie B. Nahil

Editor: Solveig Haugland

Cover Designer: Nina Scuderi

Art Director: Gail Cocker-Bogusz

Preface

The first edition of Just Java was one of the earliest books to accompany the original release of Java in 1996. The launch of Java coincided with the explosion of interest in the web and the net which, in turn, drove technology forward at a frantic pace. People talked about "Internet time," which meant three things to me in Silicon Valley: there was immense pressure to rapidly create new hardware and software products; everyone wrote software to display stock prices on their desktops and cell phones; you were forgiven for not showering if you fell asleep at your desk after midnight and woke up there the next morning. Times have changed, but software productivity remains a big reason behind Java's popularity.

Over the last eight years Java has had six major releases, averaging one about every 18 months. With each of these releases, there has been a new edition of Just Java to describe and explain the technology. Table 1 shows how the language and libraries have improved.

Table 1. Java changes from JDK 1.0.2 to Java 2 v1.4

Release	Date	Content	See Just Java 6th ed.
JDK 1.0.2	Jan 1996	First general release of the language and libraries	Throughout the book
JDK 1.1	Feb 1997	Language changes:	
		Instance initializers	Chapter 5
		Array initializers	Chapter 9
		Nested classes	Chapter 12
		Library changes:	
		Delegation based event-handlers	Chapter 20
		I/O Readers and Writers	Chapter 17
		Object serialization	Chapter 18
JDK 1.2 (rebadged to Java 2)	Dec 1998	Language changes:	
		strictfp	Chapter 7
		Weak references	Chapter 10

Acknowledgments

I'm very grateful to the following people, who are some of the most talented and creative individuals you'll find:

Gilad Bracha

Michael Davidson

Jane Erskine

Marcus Green

Roedy Green

Trey Harris

Karsten Lentzsch

Bob Lynch

Aleksander Malinowski

Simon Roberts

Rick Ross, who provides first-class leadership at http://javalobby.org

Kerry Shetline

Robin Southgate

Lefty Walkowiak

All the cowboys and cowgirls down on the Java Ranch http://javaranch.com

The Limewire team

The unsung heroes and heroines of Sun's Java and Solaris groups

The editorial, marketing, and production teams at Prentice-Hall and Sun Microsystems deserve full appreciation:

Greg Doench, Chris Guzikowski, Julie Nahil, Raquel Kaplan, Nina Scuderi, and Solveig Haugland and her magic editing pixies.

Thanks too, to my wife, family, and friends—hey, if I wanted my study to look organized, I'd keep it that way, OK?

Part 1: Language

Chapter 1. What Can Java Do for Me?

Java has become a very popular programming language for the kind of modern software people want to write. Java began as a research project inside Sun Microsystems. The results were posted on the web, and Java took off like a Titan rocket. Sun wisely decided to nurture the market by sharing, and licensed the technology across the computer industry. Today, Java compilers and source code are available for free download from many different organizations. Just about the entire computer industry is backing Java enthusiastically with products and support. In this chapter we'll look at the reasons behind Java's popularity, and summarize the key features of the language.

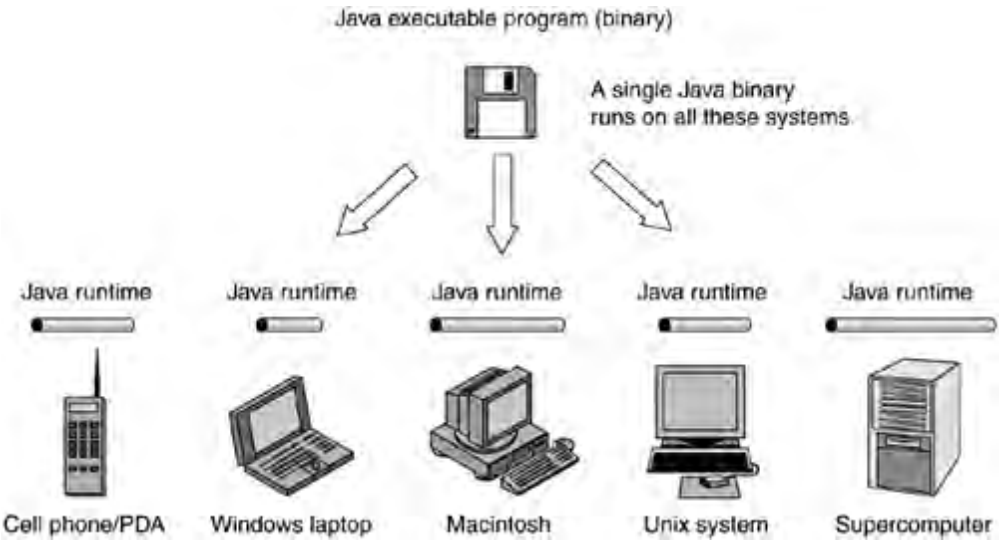
Java works well for web and server-based applications. It has great features like object-oriented programming, easy database access, and well-designed GUI support. The latest release, Java 2 version 1.5, has performance tweaks to speed up desktop programs. There are libraries for communicating across networks, and for encryption. It has strong security built in. Most programmers pick up Java quickly. Behind all this, Java is more than just the latest, most popular programming language. It is a way of creating applications that are independent of all hardware and all operating systems.

What Java Does for You: Software Portability

What is meant by "applications that are independent of all hardware and operating systems?" It means you can compile a Java program on any system and run the resulting binary executable file on the same or any other system—on a Macintosh, on Windows 98, NT, 2K, XP, on Solaris, Linux, BSD or any of the varieties of Unix, on IBM's mainframe operating systems, on cell phones, Personal Digital Assistants (PDAs), embedded processors, and even on smart cards (credit cards with a microprocessor and memory), as shown in Figure 1-1.

Figure 1-1. Future-proof software: Your Java application runs on every system. No more "requires Windows XP" or "Linux PPC only" , or "compiled for MacOS X version 10.5"; just "built with Java," and you're done.

[View full size image]



You will even be able to run on future operating system releases, like Microsoft's Longhorn. Java will be ported to all future operating systems of significance. It's also a fast way for new hardware to get software applications that will run on it.

Java unlocks software from being coupled to any specific OS and hardware platform. This new flexibility has made Java very popular with users, IT departments, and software vendors. All three benefit enormously from software portability.

Why Portability Matters

You might think that software portability does not affect you: your application software runs fine on your PC today and that's all you use. And that's true, right up until the time you want to consider a new or different system.

Say you're in the market to buy a new desktop system, and a friend shows you the video-editing, music, virus resistance, and digital picture capabilities of his Apple Macintosh. You consider switching to a Mac. Switching GUIs is a no-brainer—GUIs all do essentially the same things, and it only takes a day or two to re-train your fingers. The problem is the apps. You are faced with the "choice" of walking away from your investment in your existing PC-only software, or buying yet another Windows PC that has some compatibility with your previous system. You can easily switch hardware from Dell to H-P or IBM, but there's a software barrier to switching from Windows to something with fewer security problems, like Linux or MacOS. You've been locked in.

When your application programs are written in Java, you can upgrade OS and applications independently. You can try Linux and still use your familiar applications. You can move your existing Java programs to any new system, and carry on using them. This is why software portability matters to home users.

For businesses, the problem is worse and far more expensive. Even if your whole organization has standardized on, say, Microsoft Windows XP, there have been numerous releases over just the last decade and a bit: MS-DOS, Win 3.1, Win3.11, Win95A, Win95B, Win98, 98 SE, ME, NT 3.1, NT3.5, NT3.51, NT4, 2K, multiple service packs and required hot-fixes, XP, and Longhorn on the horizon. These platforms have subtle and different incompatibilities amongthem. Even applications running on a single platform have limited interoperability. Older versions of Microsoft Office cannot read files produced by default from the latest Microsoft Office, even when the files don't use any of the new features.

This is done deliberately, to force upgrades. If even one person in an office upgrades, everyone has to (or risk being cut off from reading new files).

Software portability is all about "future-proofing" your software investment. Rewrite it in Java, and that's the last port you'll ever need to do. Portability is the Holy Grail of the software industry. It has long been sought, but never before attained. Java brings the industry closer to true software portability than any previous system has.

Software portability for office applications

There is today an excellent alternative to costly and incompatible MS Office updates. You can download the free OpenOffice.orgsoftware and use it instead of MS Office.

MS Office Professional Edition 2003 costs $499 in the USA for the basic product—for one computer. OpenOffice.org has the same look and features, and is free for any number of computers. OpenOffice.org can read and save files in MS Office formats. You can even get the source code. Over

Language and Libraries

Let's spend a minute to review some software terms that are often taken for granted. If you are already familiar with this, just skip ahead until you reach something new. The terminology is spelled out in detail here to give a solid basis for the material that follows in the rest of the book.

What a language is

When people talk about a "programming language", they mean things like:

- How the language describes data,
- The statements that work on data, and
- The ways the two can be put together. The set of rules describing how you can put together programs in the language is called a grammar. The grammar for a programming language is written in mathematical language, and it defines how expressions are formed, what statements look like, and so on. We'll stay away from mathematics and explain things in English.

Java is a programming language in the same part of the languages family tree as C++, Pascal, or Basic. Java adopts ideas from non-mainstream languages like Smalltalk and Lisp, too. Java is a strongly typed language, meaning that the compiler strictly checks operations and will only allow those that are valid for that type of operand (you can't multiply two character strings, but you may append them) Java is object-oriented, meaning that the data declarations are very closely tied to the statements that operate on the data. Object-oriented means more than just tying data to statements, and we expand on that in chapters 2 and 5.

Statements are grouped into what other languages call functions, procedures, or subroutines. In an object-oriented language, we call the functions methods as they are the method or way to process some data. Methods can call other methods and can be recursive (call themselves). Program execution begins in a method with the special name main().Statements in Java look similar to statements in many other high-level languages. Here are some examples of Java statements:

```
y = y + 1;

if ( isLeapYear(y) )      // this is a comment

     febDays = 29;

else febDays = 28;
```

One Size Doesn't Fit All

It's quite an accomplishment to run the same binary executable on such widely different architectures as MacOS, Windows, Linux and a quarter of a billion cell phones. There's more information about this coming up in Chapter 2, but for now, let's make an over-simplification and say that Java binary executables are interpreted on each platform.

The Java binary format doesn't contain machine code instructions for any one computer. It contains a slightly higher level, more general, set of instructions known as byte code. To run a Java program, you actually run an interpreter. The interpreter reads the binary file containing the byte code and translates it into the machine code for the system where you are executing the program. The full and more complete story is in Chapter 2, remember.

It's also not easy to run software on everything from a smart card to a supercomputer cluster. That encompasses a wide range of hardware capabilities—virtual memory, hard disk, processor speed, file space, GUI abilities, memory size, and so on. Sun has achieved Java support on virtually every computing device[1] by defining different "editions." The smaller editions have a subset of the libraries in the bigger editions.

[1] I have a signet ring that runs Java programs! These rings were given out at the 1998 JavaOne conference. See www.ibutton.com.

There are three editions of Java, and the smallest one is further sub-divided. For the sake of clarity, they should just have been called the small, medium and large editions, but the Java marketing folks at Sun gave them these names instead:

Micro Edition (the "small" platform)

The Micro Edition is a very low-footprint Java run-time environment, intended for embedding in consumer products like cell phones and other wireless devices, palmtops, or car navigation systems. You will develop your code using J2SE, and then deploy onto the various small devices. The Micro Edition is also known as Java 2 Micro Edition or "J2ME" for short.

The J2ME environment is further subdivided into "profiles." There is a profile (Connected Limited Device Configuration, or CLDC) that defines the libraries available for PDA-type hardware. I have a handheld Sharp Zaurus PDA which runs the Linux operating system just like one of my desktop systems, so PDAs can be pretty capable these days. Another profile (the Mobile Information Device Platform, MIDP) is for wireless devices such as cell phones. Cell phones today are general purpose computers, with a few special peripherals. The smallest profile, which runs in just 128 Kbytes of memory, is intended for smart cards. It's called the Java Card API. The guy at Sun in charge of really dumb names (like J2SE SDK, CLDC and MIDP) must have been on vacation the week the API for Java cards got named "the Java Card API". This profile is allowed to omit support for floating-point arithmetic when the hardware doesn't have it.

The J2ME environment is enjoying enormous success and has shipped in many millions of cell phones already. The year 2002 marked the point at which the number of handheld computing devices sold exceeded the number of PCs sold, so Java's success in this sector has real momentum. The Zelos Group (a company of technology analysts) predicts that Java will run on 450 million handsets by 2007, which is 75% of those shipping that year. As cell phone and PDA makers continue to search for the elusive sweet spot that combines the two products, Java dominates software for one, providing a bridge to the other. One programmer commented "by learning to program in Java, you free yourself from

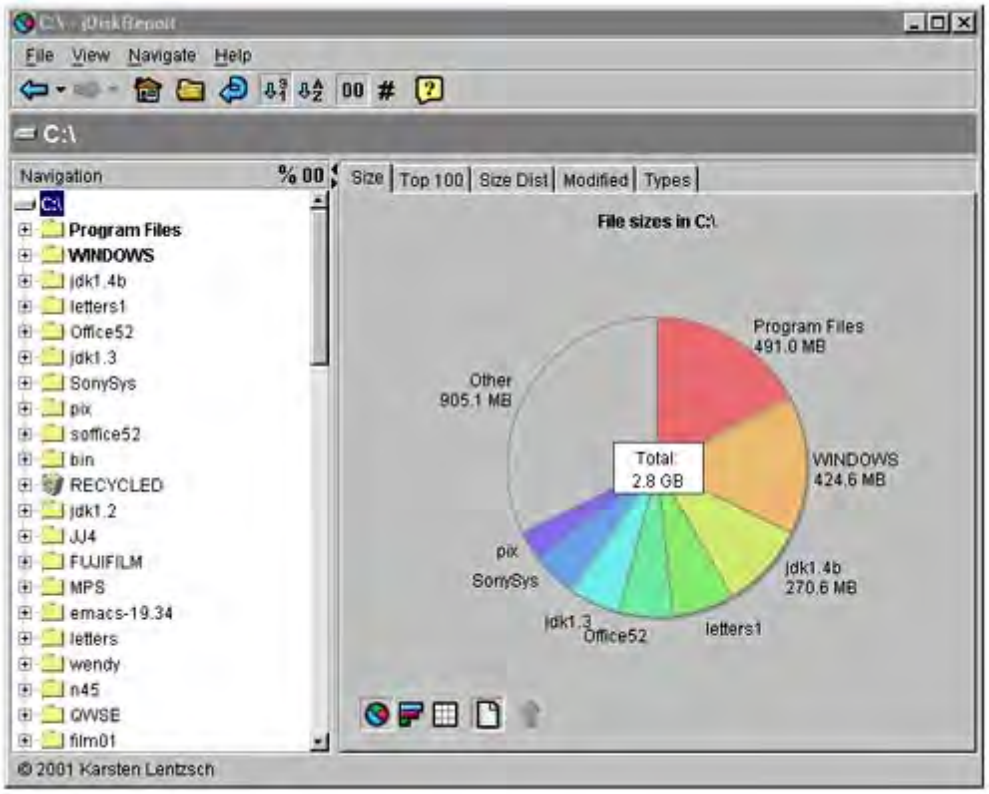
Some Light Relief—A Java Desktop Application

Do you have trouble keeping track of your disk usage? Do you sometimes need to free up space, but have no idea what to start deleting? Have a look at jDiskReport, which is a Java application written by expert programmer Karsten Lentzsch of Kiel in Germany.

jDiskReport is a free cross-platform graphical disk report utility. It lets you understand how much space the files and directories consume on your hard disks.

Figure 1-2 shows jDiskReport but really doesn't do justice to the program.

Figure 1-2. The jDiskReport: keeping track of hard disk space.



[View full size image]

The jDiskReport software is freely from Karsten's website at www.jgoodies.com. Download it and unzip the file. There is a readme.txt and a jar file. Go to the directory containing those files, and start the program with the command line

```
java -jar jdiskreport.jar
```

Chapter 2. Introducing Objects

Here's where we get to grips with a real Java program. Follow the steps in Appendix A to download a Java compiler. Then you can type in source code and test examples as we go along.

The chapter presents some basics of Object Oriented Programming (OOP). We'll explain exactly what language designers mean by a type, and relate that to the objects in OOP. We'll develop a type called Timestamp, then add a couple of other types to turn it into a complete program. Along the way, we'll clarify the super-important distinction between "an object" and "a reference to an object". The chapter ends with the development of a small but complete running Java program that displays a desktop clock.

Downloading and Compiling Java

If you use Solaris or an Apple computer running MacOS X, a Java compiler is already installed on your system. So you folks can skip right ahead to "Running a Java Program". Dell and H-P preload Java libraries on their Windows PCs, but you still need to download a compiler. If you are using Windows, Linux, or Solaris, Sun's website at java.sun.com is the easiest place to get a free Java compiler. There are other places too. IBM has some great Java compiler and tool downloads. If you are using something other than Linux, MacOS, Windows or Solaris, find the Java compiler by searching the manufacturer's website.

Downloading a Java compiler system

Follow the steps in Appendix A to download and install a Java Development Kit (JDK). The current version of the JDK is called "J2SE 1.5 SDK" i.e. Java 2 Standard Edition, version 1.5 of the Software Development Kit. Most people call it by the old, simpler name "JDK 1.5". If there is a newer version than 1.5, such as 1.5.1 or 1.6, get that instead.

Compiling a Java program

A lot of programmers use IDEs to develop their code, and there are many IDEs that support Java, including some free ones such as Sun's NetBeans, Eclipse, or the BlueJ environment. You can get a list by googling for "Java IDE". We'll avoid IDEs here and do everything from the command line, to keep the learning experience focused on Java. Create a separate work directory just for your files, and don't put your directory anywhere under the JDK installation directory.

```
mkdir c:\work

cd c:\work
```

After installing, you can run the Java compiler. Type this at the command shell:

```
javac
```

Assuming you installed the JDK and set the path correctly, you will get back a dozen lines of messages, starting with this line:

What Is a Class?

In a way, Object Oriented Programming (OOP) is a misnomer, as the fundamental things we're dealing with are classes. Perhaps it should have been called "Class-based Programming", but it's too late to change now. So what's a class, and what's an object? There are several different ways of coming at this, and we'll try a couple of them. If you are already familiar with OOP, then just skim through the chapter to pick up the Java specifics. Be sureto visit the "What is an Object?" section, which describes an object-oriented feature handled differently in Java (reference types).

Declaring a variable

To get a good understanding of OOP, you need to understand data types in procedural (non-OOP) languages. The idea of representing real world objects by types and variables in a computer is called abstraction. Here's a quick review. You'll be familiar with the idea of declaring a variable. In many languages, a variable declaration will look something like this:

```
int mm = 0;
```

Depending on the exact rules of a language, the keyword for the type might be "int" or "integer" or something similar, and it might come before or after the variable name. In Java, a variable is declared exactly as shown here: type name first, followed by variable name. When you write a declaration like:

```
int mm = 0;
```

you are:

- Allocating some storage, and initializing it to zero,
- Giving it a name, mm,
- Saying that it can only hold values that are compatible with the int type. You cannot put a string of characters into mm.

Simple types

What Is an Object?

The previous section focused on classes and ended with a small but complete user-defined class called Timestamp. We've already let the cat out of the bag that an object is a variable of a class type, not a primitive type. There are several other ways to say that:

- An object is an instance of a class
- An object belongs to a class
- An object belongs to a reference type

However, they all mean the same thing. This section fills in some more of the details on objects: how you declare them and how you create them (two different things), and how you use them.

What "reference type" really means

An object declaration looks exactly the same as a primitive variable declaration. You first write the type name (which is the class name), then follow it by the variable name (which is the object name). So it's just like a declaration of a primitive variable. Here is a declaration of a variable of class type:

```
Timestamp  tick;
```

The following example shows a qualified name that lets you declare an object of a type defined in another package:

```
java.util.Calendar  now;
```

The name Calendar is qualified by the package it is in, java.util.

The Java Language Specification talks about classes being reference types. It means memory references, as in a pointer or address. The variables in the previous examples, tick and now, look like they can hold an object. Don't be fooled!

Java Digital Clock Program

Here's the rest of the code that makes up the digital clock. The program is made up of three classes:

- TheTimestamp class that can give you the current hours, minutes, and seconds.
- A class called "ClockView". This holds everything related to the visual appearance of the clock on screen.
- A class called "clock" that has overall control, and is where execution starts.

For a small example like this, we could merge everything into one class, and make it a dozen lines shorter. By keeping the design elements in separate classes, you can form a better idea of how classes should be used in bigger programs and systems.

The main() routine where execution starts

We'll start with the class that has overall control, where execution starts. Every Java application ever written starts in a method called "main" that has this general appearance.

As the diagram indicates, the name and parameters of a method are together known as the signature. The signature of a method comes up later in some other class features. Those three qualifiers at the front of the main name "public static void" always appear on the front of the main routine, so just take them as given for now. We've already seen that "void" means "this method does not have a return value". Chapter 6 has an explanation of static if you want to peek ahead.

You write your main method, and put it in whichever of your classes is going to have overall control of the program.

The actual statements in our digital clock main routine are quite brief. Here they are in full (line numbers added on the left for ease of description):

```
1   public static void main(String[] args) {
2
```

Summary

You have now covered two of the four cornerstones of object-oriented programming: abstraction and encapsulation. This section summarizes what we've seen.

OOP is all about special support for class types and the operations on them. Objects are variables whose type is a class. Classes, objects, and methods belong together in a unbreakable way. The methods in a class can only be invoked on objects that belong to that class. If you don't have an object, you cannot invoke one of the object methods. You cannot write a method in class A that is invoked on an object of some unrelated class B. There is no way to get that past the compiler in Java.

There are no structures or records in Java. The most important way to group related things is to put them in a class. You also put classes together into a package. Packages are usually implemented as directories in the file system, though the JLS doesn't demand that. You can have nest packages within a package. This is called a sub-package.

The rest is just details (although there are a lot of them). We've touched on constructors and hinted about inheritance. This is a short summary, but the concepts are critical to object oriented programming and being an effective Java programmer. It's a good idea to turn the corner of this page down, and come back periodically to ensure the concepts stick.

Exercises

1. Write down an explanation of how a class, object, and method are related.
2. A rectangle can be defined by the length of two adjacent sides. Write a class that holds these two pieces of integer data, and provides useful rectangle-related operations, like calculating the area, updating the side data, and calculating the combined area of two rectangles. You'll write a method with a signature like this:

```
int combinedArea(secondRect sr) // the first rect is "this" of course
```

3. Rewrite the ClockView class (only) so the clock displays the time in Roman numerals. The Roman numerals from 1 to 12 are I, II, III, IV, V, VI, VII, VIII, IX, X, XI, and XII.
4. Rewrite the ClockView class so it uses the 12-hour clock and also displays AM or PM as appropriate.
5. Rewrite the ClockView class so that it includes tenths of seconds, and change the main routine in the clock class to update the display every tenth of a second. This code in Timestamp will obtain the tenths of a second from a Calendar object:

```
int mlsec = now.get(java.util.Calendar.MILLISECOND);
tenths = mlsec / 100;
```

You'll need to declare tenths as a field of Timestamp. You also need to put that number into the display of the clock in ClockView.

Some Light Relief—Napster and LimeWire

Everyone knows the story of Napster by this point. College student Shawn Fanning (baby nickname: "the napster" because of his sleeping habits) threw together some rickety old Windows software to transmit the titles of any music tracks you had on your PC to a central database.

Other users could search that database looking for titles they wanted. When they found a match, the Napster software would set up a peer-to-peer connection and allow direct transfer of the music bits from your PC to the unknown fan elsewhere. Your incentive to share was that you in turn would be able to get files from other people similarly sharing their titles.

That was the concept at least. In practice, the record companies and some bands took exception to freelance distribution. By this time, Fanning had parlayed his idea and some prototype software into a start-up venture backed by premier venture capitalist company Kleiner Perkins Caulfield and Byers. At its peak, Napster claimed it had 70 million users, and even if they only exaggerated by the industry standard factor (ten-fold), that's still a lot of users.

Napster's demise took a couple of years to wind through the legal system, but the central point never seemed that subtle to me: you cannot legally broker the wholesale transfer of other people's intellectual property. Naturally, the record companies conducted themselves with their usual rapacious shortsightedness. Instead of licensing the Napster software to supply what customers clearly wanted, and charging fees, they tried to sue Napster out of existence. It's a reprise of 1992 when they killed DAT (Digital Audio Tape) by encumbering it with anticopying hardware backed by law. Incredible.

Which brings us to the current situation. Napster is just about completely dead. Apple has taken the largest slice of the online music market by offering the service that everyone wanted all along: reasonably priced legal music downloads through its iTunes Music Store. The music you buy on iTunes can easily be played on iPods, which are a high profit margin product for Apple. Microsoft is poised to leverage its monopoly and belatedly bundle its imitation of Apple's Music Store. And a number of underground, peer-to-peer file-sharing distributed databases have replaced Napster's centralized model, most notably a service called BitTorrent.

When I tried the Napster software in its heyday, all the way back in the last millennium, my first thought was, "Why on earth didn't they write this in Java?" It was a simple network database lookup with peer file transfer capability front-ended by a simple GUI. Tailor-made for Java! Fanning was not familiar with Java, so he churned out his Windows-only software. Though Napster has now passed on to that great big recording studio in the sky, others still carry the conductor's baton.

Bearshare, Gnutella, and LimeWire are currently three popular applications for sharing files (including .mp3 music files) across the net. They use the Gnutella client protocol, which is a search engine and file serving system in one. It is wholly distributed. Anyone can implement it to share their content (any files) with others. The great thing about LimeWire is that it is written in Java. You can download the application from www.limewire.com and see for yourself. The main screen is shown in Figure 2-5.

Figure 2-5. LimeWire: Napster's successor

Chapter 3. Primitive Types, Wrappers, and Boxing

Literal Values

As we mention each of the eight primitive types, we'll show a typical declaration; say what range of values it can hold; and also describe the literal values for the type.

A "literal" is a value provided at compile time. Just write down the value that you mean, and that's the literal. Here's a snippet of code showing several literals:

```
int  i = 2;       // 2 is a literal

double d = 3.14;  // 3.14 is a literal

if ( c == 'j' )     // 'j' is a literal
```

Every literal has a type, just like every variable has a type. The literal written as 2 has type int. The literal written as 3.14 belongs to the type double. For booleans, the literals are the words false and true.

It is not valid to directly assign a literal of one type to a variable of another. In other words, this code is invalid:

```
int  i = 3.14;  // type mismatch! BAD CODE
```

The literal 3.14 is a double floating-point number literal, and cannot be assigned directly to an int variable. You can do it by forcing an explicit type conversion, known as a cast. The details are coming up shortly.

A language that allows assignment and conversion only between closely related types is called a strongly typed language. Java is a strongly typed language. Strong typing reduces the power of a language, but reduces some errors even more.

boolean

This is the data type used for true/false conditions. To speed up memory access, implementations don't pack boolean values into the theoretical one-bit minimum space, but put each boolean into a byte.

example declaration:

```
boolean b = false;
```

range of values: false, true

literals: false true

In the code below,

```
boolean found = false;

/* more code */

if (x == 99) found = true;
```

x is compared with value 99. If x matches 99, then the boolean found is set to true.

You cannot assign or cast a boolean value to any other type. However, you can always get the same effect by using an expression, like the following:

char

This type is a 16-bit unsigned quantity that is used to represent text characters. You'd use this to hold a single character. If you want to store several successive characters, such as a name or an address or phone number, you would use the predefined type String, which keeps track of multiple consecutive characters at once. String is a class in the java.lang package, not a primitive type. We'll get to String a little later.

The type char has been made 16 bits long because that's what you need to support character sets from every locale in the world. The USA can get by with just 7-bit ASCII, and Western Europe is fine with 8-bit ISO 8859-1 characters giving a few accented characters. Many other places such as China, Korea, India, and Japan need and use 16-bit character sets. So Java does too. If you are using a computer in the 8-bit western world, characters are automatically converted to/from 16 bits on the way in and out of Java.

Although char holds a text character, inside the JVM char is just a 16-bit binary number. So all the arithmetic operators are available on type char. But unlike all the other arithmetic types, char is unsigned—it never takes a negative value. You should only use char to hold character data or bit values. If you want a 16-bit quantity for calculations, don't use char, use short. If you ignore this advice, a cast (conversion) of a negative value into char will make the value magically become positive without the bits changing. If you cast that char value back to an int, the int will now have a positive value. That's a surefire way to introduce bugs into your code.

example declaration:

```
char testGrade;

char middleInitial;
```

range of values: a value in the Unicode code set 0 to 65,535

You have to cast a 32-or 64-bit result if you want to put it into a smaller result. This means that assignments to char must always be cast into the result if they involve any arithmetic. An example would be:

```
char c = (char) (c + 13); // the cast is required
```

literals: Char literals are always between single quotes. String literals are always between double quotes, so you can tell

int

The type int is a 32-bit, signed, two's-complement number, as used in virtually every modern CPU. It will be the type that you should choose by default whenever you are going to carry out integer arithmetic.

example declaration:

```
int i;
```

range of values: –2,147,483,648 to 2,147,483,647

literals: Int literals come in any of three varieties:

- A decimal literal, e.g., 10 or –256
- With a leading zero, meaning an octal literal, e.g., 077777
- With a leading 0x, meaning a hexadecimal literal, e.g., 0xA5 or 0Xa5

Uppercase or lowercase has no significance with any of the letters that can appear in integer literals. If you use octal or hexadecimal, and you provide a literal that sets the leftmost bit in the receiving number, then it represents a negative number. (The arithmetic types are stored in a binary format known as "two's-complement"—google for full details).

A numeric literal is a 32-bit quantity, even if its value could fit into a smaller type. But provided its actual value is within range for a smaller type, an int literal can be assigned directly to something with fewer bits, such as byte, short, or char. If you try to assign an int literal that is too large into a smaller type, the compiler will insist that you write an explicit conversion, termed a "cast."

When you cast from a bigger type of integer to a smaller type, the high-order bits are just dropped. Integer variables and literals can be assigned to floating-point variables without casting.

long

The type long is a 64-bit, signed, two's-complement quantity. It should be used when calculations on whole numbers may exceed the range of int. Using longs, the range of values is –263 to (263–1). Numbers up in this range will be increasingly prevalent in computing, and 264 in particular is a number that really needs a name of its own. In 1993, I coined the term "Bubbabyte" to describe 264 bytes. Just as 210 bytes is a Kilobyte, and 220 is a Megabyte, so 264 bytes is a Bubbabyte. Using a long, you can count up to half a Bubbabyte less one.

example declaration:

```
long nationalDebt;
```

range of values: –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

A word about casts (type conversion)

All variables have a type in Java, and the type is checked so that you can't assign between two things that are incompatible. You cannot directly assign a floating-point variable to an integer variable.

It is reasonable, however, to convert between closely related types. That is what a cast does. You cast an expression into another type by writing the desired new type name in parentheses before the expression, as follows:

```
float f = 3.142;
int i = (int) f; // a cast
```

Some numeric conversions don't need a cast. You are allowed to directly assign from a smaller-range numeric type into a larger range—a byte to int, or int to long, or long to float—without a cast. Assigning to a more capacious type is called a widening primitive conversion, and it does not lose any information about the overall magnitude of the numeric value. When you want to do an assignment that can potentially

byte

The byte type is an 8-bit, signed, two's-complement quantity. The reasons for using byte are to hold a generic 8-bit value, to match a value in existing data files, or to economize on storage space where you have a large number of such values. Despite popular belief, there is no speed advantage in arithmetic on shorter types like bytes, shorts, or chars—modern CPUs take the same amount of time to load or multiply 8 bits as they take for 32 bits.

example declaration:

```
byte heightOfTide;
```

range of values: –128 to 127

literals: There are no byte literals. You can use, without a cast, int literals provided their values fit in 8 bits. You can use char, long, and floating-point literals if you cast them.

You always have to cast a (non-literal) value of a larger type if you want to put it into a variable of a smaller type. Since arithmetic is always performed at least at 32-bit precision, this means that assignments to a byte variable must always be cast into the result if they involve any arithmetic, like this:

```
byte b1=1, b2=2;

byte b3 = b2 + b1; // NO! NO! NO! compilation error

byte b3 = (byte) (b2 + b1); // correct, uses a cast
```

People often find this surprising. If I have an expression involving only bytes, why should I need to cast it into a byte result? The right way to think about it is that most modern computers do all integral arithmetic at 32-bit or 64-bit precision (there is no "8-bit add" instruction on modern CPUs). Java follows this model of the underlying hardware.

An arithmetic operation on two bytes potentially yields a bigger result than can be stored in one byte. The philosophy for numeric casts is that they are required whenever you assign from a more capacious type to a less capacious type. This is termed a narrowing primitive conversion, and it may lose information about the overall magnitude of a numeric value. It may also lose some digits of precision, and sign information (when converting to char, because chars are always interpreted as positive values).

short

This type is a 16-bit, signed, two's-complement integer. The main reasons for using short are to match external values already present in a file or to economize on storage space where you have a large number of such values.

range of values: –32,768 to 32,767

literals: There are no short literals. You can use, without a cast, int literals provided their values will fit in 16 bits. You can use char, long, and floating-point literals if you cast them.

As with byte, assignments to short must always be cast into the result if the right-hand side of the assignment involves any arithmetic.

The last two primitive types, double and float, are floating-point arithmetic types. Before we look at the features of double and float, Limited Accuracy of Floating Point Types on page 49 describes some problems you may encounter in any language, when using floating-point arithmetic.

Limited Accuracy of Floating Point Types

Before we describe double and float, we need to point out two limitations of floating-point arithmetic. People are still getting Ph.D.s for probing the mathematics underlying floats, but the short story is:

- Floating-point numbers only hold a limited number of significant figures. The float type only holds six to seven significant figures. So you can hold the number 123,456 accurately, and you can hold the number 0.123456 pretty accurately, but it's certain that you cannot hold the number 123,456.123456 in a float variable accurately because that would require 12 significant figures. You can write the number in your program, and you'll actually get a number that is approximately the value you want, but not exactly equal. (You'll get 123456.125, in fact).
- Floating-point numbers may contain tiny inaccuracies that can mount up as you iterate through an expression. Don't expect ten iterations of adding 0.1 to a float variable to cause it to exactly equal 1.0F!

Floating-point numbers have these limitations in every programming language. It is inherent in the type. You are trying to represent an infinite quantity of numbers in a finite type. The only way this can be done is by picking points on the real-number continuum and representing those exactly. Then use those model values to represent approximations to all other real numbers. It's as though we could only store tenths, and so everything from 0.0 to 0.049 becomes 0.0. Everything from 0.05 to 0.149 becomes 0.1, and so on. With floats, we're working with millionths, not tenths. But it is still not perfectly accurate for most numbers. With this background about the limitations, let's review the two floating point types.

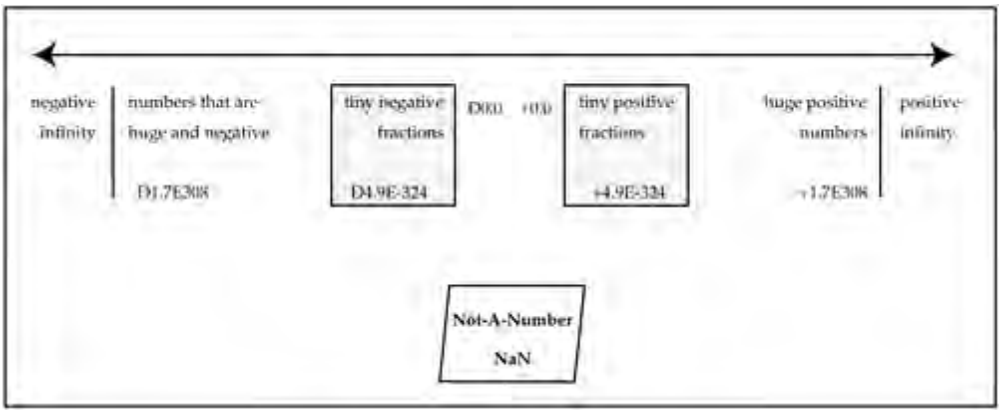
double

The type double refers to floating-point numbers stored in 64 bits, as described in the IEEE[1] standard reference 754. The type double will be the default type you use when you want to do some calculations that might involve decimal places (i.e., not integral values).

[1] IEEE is the Institute of Electrical and Electronic Engineers, a U.S. professional body.

range of values: These provide numbers that can range between about –1.7E308 to +1.7E308 with about 15 significant figures of accuracy. The exact accuracy depends on the number being represented. Double precision floating-point numbers have the range shown in Figure 4-1.

Figure 4-1. Type double should be used when calculations involve decimal places



[View full size image]

IEEE 754 arithmetic has come into virtually universal acceptance over the last decade, and it would certainly raise a few eyebrows if a computer manufacturer proposed an incompatible system of floating-point numbers now. IEEE 754 is the standard for floating-point arithmetic, but there are several places where chip designers can choose from different alternatives within the standard, such as rounding modes and extended precision. Java originally insisted on consistency on all hardware by specifying the alternatives that must be used. That is now loosened somewhat with strictfp.

Make mine a double: how large is 1.7E308?

The largest double precision number is a little bit bigger than a 17 followed by 307 zeroes. That's an amazingly huge number.

The volume of the observable universe is about (4pi/3)(15 billion light-years)3 = 1085 cm3. Protons, the fundamental particle found in the nucleus of an atom, have an average density (averaged over all space, not just on planets) of about 10-7 cm-3.

float

The type float refers to floating-point numbers stored in 32 bits, as described in the IEEE standard reference 754.

The justification for using single-precision variables used to be that arithmetic operations were twice as fast as on double precision variables. With modern extensively pipelined processors and wide data buses between the cache and CPUs, the speed differences are inconsequential. The reasons for using floats are to minimize storage requirements when you have a very large quantity of them or to retain compatibility with external data files.

Floats are best avoided if possible, because they have such limited accuracy. They are in Java because they are supported in hardware so it costs little, and provides compatibility with existing code and expectations.

range of values: The type float provides numbers that can range between about –3.4E38 to 3.4E38 (i.e., 340,000,000,000,000,000,000,000,000,000,000,000) with about six to seven significant figures of accuracy. The exact accuracy depends on the number being represented.

literals: The simplest way to understand what is allowed is to look at examples of valid float literals, as follows:

```
1e1f  2.f  .3f  3.14f  6.02e+23f
```

A suffix of "F" or "f" is always required on a float literal. A common mistake is to leave the suffix off the float literal, as follows:

```
float cabbage = 6.5;
```

```
Error: explicit cast needed to convert double to float.
```

The code must be changed to the following:

```
float cabbage = 6.5f;
```

Object Wrappers for Primitives

Each of the eight primitive types we have just seen has a corresponding class type, predefined in the Java library. For example, there is a class java.lang.Integer that corresponds to primitive type int. These class types accompanying the primitive types are known as object wrappers and they serve several purposes:

- The class is a convenient place to store constants like the biggest and smallest values the primitive type can store.
- The class also has methods that can convert both ways between primitive values of each type and printable Strings. Some wrapper classes have additional utility methods.
- Some data structure library classes only operate on objects, not primitive variables. The object wrappers provide a convenient way to convert a primitive into the equivalent object, so it can be processed by these data structure classes. One example of a data structure class that only operates on objects is the class java.util.SortedSet, which keeps a collection of objects in sorted order for you.

Why use object wrappers instead of primitive types directly?

There is one more big reason why we wrap primitive types. There is a "joker" reference type that can hold a pointer to any type of object at all. This is the type java.lang.Object. Code like this is perfectly legal (and common in some kinds of program):

```
Timestamp t = new Timestamp();

java.lang.Object o = t;
```

There is no corresponding primitive type that can hold any kind of primitive variable. Therefore, if you want to have an algorithm that operates on arbitrary or unknown variables, make it work with variables of type java.lang.Object. If the arbitrary values are primitives, wrap them and use java.lang.Object references to manipulate them.

You might wonder what kind of processing you can do to an object, if its true type (and therefore its methods) are concealed from you. You can hold it in a data structure such as a stack, queue, set, etc.

Autoboxing and Unboxing

Autoboxing is a feature that came into Java with the JDK 1.5 release. It recognizes the very close relationship between primitive variables and objects of their corresponding wrapper type, and provides a little extra help from the compiler. Autoboxing says that you can convert from one to the other without explicitly writing the code. The compiler will provide it for you.

In releases before 1.5, you could get an int value into a java.lang.Integer like this:

```
int i = 27;

Integer myInt = new Integer(i);
```

With autoboxing, you can make the assignment directly, and the compiler looks at the types of the variables, and says "yes, I know how to wrap (or "put a box around") an int to get an Integer object". Then it allows the expression, and generates the code to do it. So you may write this instead:

```
int i = 27;

Integer myInt = i;          //  autobox!
```

Unboxing is the same concept going the other way, from a primitive-wrapper object to a primitive type. Here's an example with a wrapper object for the Double type:

```
Double dObj = 27.0;  // autobox

double d = dObj;     // unbox, gets value 27.0
```

Boxing isn't just for variable creation or initialization. You can use it wherever the context expects a primitive and you have the corresponding wrapper. It gets automatically wrapped for you. Here's an example showing how we can use a Boolean object where a boolean primitive is expected:

Performance Implications of Autoboxing

The autoboxing feature was introduced to Java for programmer convenience, particularly in the Collections classes that provide ready-made data structures for use on Objects. Where the intent is obvious, you no longer have to write it all out explicitly.

Just because you don't write the code explicitly, it doesn't mean the code goes away. The compiler generates actual instructions for the implied type promotions between a primitive and its object wrapper.

Those actual instructions have a performance cost at run-time, even though you don't see them at compile time. Take this autoboxing, for instance:

```
Integer myInt = i; //  autobox!
```

The compiler will treat that the same way as if you had written a constructor call:

```
Integer myInt = new Integer(i);
```

When a constructor is invoked, it causes a chain of calls to constructors in successively higher parent classes back until a constructor in java.lang.Object is invoked. The parent class of Integer is java.lang.Number, and the parent of Number is java.lang.Object. So there are three constructor calls, a memory allocation, and an assignment in that innocuous-looking autoboxing.

CPU cycles get cheaper every year, but they're not completely free or infinitely fast. Be aware of the cost of autoboxing, and avoid doing it unnecessarily or in a loop. Autoboxing costs no more and no less than the equivalent explicit statements. But it's easy to inadvertently overlook the cost of statements that aren't written explicitly.

No methods on primitives in Java

The C# language goes one step further than Java in autoboxing—you can invoke methods on primitives!

java.lang.Object

We'll finish this chapter by describing a couple of much-used classes that we have seen informally already: Object and String.

The class java.lang.Object is the ultimate parent of every other class in the system, and it has half-a-dozen methods which therefore can be invoked on any and all objects. (An object is able to invoke the methods in all its parent classes — remember how ClockView objects could call setVisible() because they got it from their parent JFrame). Following are some of the members Object has. You can skim through this now, jumping ahead to the description of type String, and return if you need more information about something specific in Object.

```
public class Object {

     public java.lang.Object();  // constructor


     public java.lang.String toString();


     public boolean equals(java.lang.Object);

     public native int hashCode();


     public final Class getClass();


     protected native java.lang.Object clone()

                    throws CloneNotSupportedException;


     // methods relating to thread programming

     public final native void notify();

     public final void wait() throws InterruptedException;

}
```

The "throws SomeException" clause is an announcement of the kind of unexpected error return that the method might give you. Object does have a few more methods than shown here. The API documentation has the full description.

java.lang.String

To complete this chapter, here is the standard Java class java.lang.String. That's obviously a class type, contrasting with the primitive types with which we started the chapter. You will use String instances a lot—whenever you want to store some characters in a sequence or do human-readable I/O.

As the name suggests, String objects hold a series of adjacent characters, similar to an array. Strings have methods to extract substrings, to put into lower case, to search a String, to compare two Strings, and so on. Arrays of char have none of these. String is a very convenient class and you'll use it extensively in many programs.

literals: A string literal is zero or more characters enclosed in double quotes, like these two lines:

```
"That'll cost you two-fifty \n"

"" // empty string
```

The empty String is different from the null pointer. The empty String is a pointer to a String object which happens to have zero length. You can invoke all the compare, search, extract methods on an empty String. They do little, but they don't cause errors.

Strings are used so frequently that Java has some special built-in support. Everywhere else in Java, you use a constructor to create an object of a class, such as:

```
String drinkPref = new String( "I like tea" );
```

String literals count as a shortcut for the constructor. So this is equivalent:

```
String drinkPref = "I like tea";
```

Language Support for String Concatenation

There is another String feature with special built-in compiler support: concatenation, or joining of two Strings. Whenever a String is one operand of the "+" operator, the system does not do addition. Instead, the other operand (whatever it is, object or primitive) will be converted to a String, and the result is the two Strings appended together. If the operand is an object, it is converted to a String by calling its toString() method. The toString() method of Object is described on page 58; String has its own special version of this.

Concatenation is a piece of "magic" extra operator support for type String. The "+" operator used to be the only operator that could be applied to an object, until unboxing was brought into JDK 1.5. You will use this String "+" feature in many places. Here are a few examples:

- To print out a variable and some text saying what it is:

```
System.out.println( "x has value " + x
                    + " and y has value " + y );
```

- To break a long String literal down into smaller strings and continue it across several lines:

```
 "Thomas the Tank Engine and the naughty "
+ "Engine-driver who tied down Thomas's Boiler Safety Valve"
+ "and How They Found Pieces of Thomas in Three Counties."
```

- To convert the value to a String (concatenating an empty String with a value of a primitive type is a Java idiom):

```
int i = 256;
```

String Comparison


Just a reminder about String comparisons. Compare two Strings like this:

```
if ( s1.equals(s2) )
```

not like this:

```
if (s1 == s2)
```

The first compares string contents, the second, string addresses (this is another artifact of reference types, remember?). Failing to use equals() to compare two strings is probably the most common single mistake made by Java novices.

Tip: Using intern() on Strings

There is one exception to rule of comparing Strings by calling equals(). The exception has been put in place as a performance optimization. String has a method called intern(). You can call intern() on one of your strings, to put it into a private program-wide pool that the String class maintains. You get back a pointer to that string in the pool. Each string is only in that pool once. If you later call intern() on a string that is already in the pool, you get the shared version back to use. This works because string contents never change after creation.

All String literals and String-valued constant expressions are interned for you automatically. That ensures that you don't have two copies of a string literal with the same contents. You can call intern() on your own strings in addition, if you wish. There is some cost to interning a string, so only do it when the number of string comparisons in your program is a lot more than the number of string creations.

The key reason to use intern() is that all strings returned from intern() can be compared for equality by

Some Light Relief—Hatless Atlas

Here's a cheery song that was written by top programmer David H. Zobel and circulated on the Internet for more than a while. It's sung to the tune of Twinkle, Twinkle, Little Star, but to sing it you have to know how some programmers pronounce the shifted and control characters on a keyboard. The "^" character above the "6" is often pronounced "hat" because it looks like a little hat. Some people give the name "huh" to "?", and "wow" to "!". Both of those are a lot shorter than more conventional names.

The song is called Hatless Atlas and it goes like this.

^ < @ < . @ *	Hat less at less point at star,
} " _ # |	backbrace double base pound space bar.
- @ $ & / _ %	Dash at cash and slash base rate,
!(@ | = >	wow open tab at bar is great.
; ' + $? ^?	Semi backquote plus cash huh DEL,
, # " ~ |) ^G	comma pound double tilde bar close BEL.

This song can be enjoyed on more than one level. While the theme is not totally transparent, key elements are revealed. The bare-headed strong man relishes nature ("point at star"), and then enjoys the full hospitality of a tavern ("Wow, open tab at bar is great"). Soon he finds that the question of payment does arise, after all; hence, the veiled reference to the Treasury lending policy ("slash base rate") and the finality overshadowing that closing lament "bar close BEL"!

I like to think that in the years to come wherever programmers gather in the evening, after the pizza is all eaten and a sufficient quantity of beer has been drunk, a piano may start to play softly in the corner. Quietly, one member of the group will sing, and then more and more of the programming staff will join in. Any systems analysts who haven't yet quaffed themselves unconscious might sway unsteadily with the beat. Soon several choruses of Hatless Atlas will roll lustily around the corners of the room. Old-timers will talk of the great bugs they have overcome and the days of punching clocks, cards, and DOS machines.

Or maybe we'll all stay home and watch reruns of Star Trek: Kirk Violates the Prime Directive with Xena, Warrior Princess instead. Who knows?

Chapter 4. Statements and Comments

Huh? What is it, Lassie? What is it, girl?

You say someone has left the braces off their if statement, and that it's likely to be a maintenance problem? Good girl, Lassie!

Statements are the way we get things done in a program. Here are the most common statements in Java.

Organizing statements	{ /*statements*/ }	and	emptyStatement ;
Expression statement	someExpression ;		
Selection statements	`if` `switch`		

Organizing Statements

Wherever you can legally put one statement, you can put a block of several statements. We've seen blocks many times. Use them to group statements in a loop or an "if" clause, or to declare a variable that will be used only within the block. This is called a local variable because it is local to the block and the name cannot be seen outside the block. A block looks like this:

```
for (;;) {
    int timeMsecs = 500;      // local variable
    Thread.sleep(timeMsecs);
}
```

This makes the entire block be the subject of the "for" loop. In this example, the for statement causes the block to be repeated in an infinite loop. It can be stopped by an exception error condition or by something external to the loop, like closing the window containing the program, or stopping the Thread it is running in.

Empty statement

Java has the empty statement, which is simply a semicolon by itself. The empty statement does nothing, and is used to make things clearer. In the example below, we are saying "if the condition is true, do nothing; otherwise, invoke the method."

```
boolean noRecordsLeft = /*some calculation*/
if (noRecordsLeft)

      ; // empty statement

else
      alertTheMedia();
```

The code is written this way to make it clearer. If you rewrite it so the "else" part becomes the "then" part to avoid an

Expression Statements

Certain kinds of expression are counted as statements. You write any expression, put a semicolon after it, and voila, it's an expression statement. In particular, an assignment, method invocation, the creation of a object by calling a constructor (looks like a method call, and the keyword "new" is used so we can tell them apart), and pre-increment, post-increment, and decrement are all expressions that can be statements.

```
a = b;                // assignment

w.setSize(200,100);   // method invocation

new WarningWindow(f); // instance creation

++i;                  // pre-increment
```

An expression statement is executed by evaluating the expression.

If the expression is an instance creation, you usually save a reference to it so that you can access fields and invoke methods on that object. For example, you have:

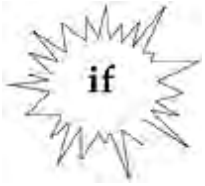
```
foo = new WarningWindow(f); // instance creation
```

not:

```
new WarningWindow(f); // instance, but no ref saved
```

However, there are certain classes we'll see later that you can instantiate for which you don't necessarily need to save a reference. The two main examples are Threads and inner (nested) classes. They are fine doing their work without further input from you. So, while you usually keep a reference to a newly instantiated class, you might occasionally see no reference kept.

Selection Statements



The general form of the "if" statement looks like this:

```
if ( Expression ) Statement  [ else  Statement ]
```

Statement Notes

✓ The Expression must have boolean type. It can be a Boolean object which will be unboxed to a boolean. Requiring a boolean (in contrast to an integer) has the delightful effect of banishing the old "if (a=b)" problem, where the programmer mistakenly taps the "=" key once instead of twice, and does an assignment instead of the intended comparison, (a==b).

✓ If that typo is written in Java, the compiler will give an error message that a boolean is needed in that context—unless a and b are booleans. For this reason, people sometimes write code like if (false==b) because the compiler will output an error message for the typo if (false=b)

✓ The Statement can be any statement, in particular a block statement, { /*more code*/ }, is normal.

The general form of the "switch" statement is impossible to show in any meaningful form in a syntax diagram with less than about two dozen production rules. That tells you something about how badly designed the statement is right there. If you look at Kernighan

Looping Statements

The "for" statement looks like this:

for (Initial; Test; Increment) Statement

There are new versions of "for" to use with collection classes, arrays and enumerated types:

for (SomeType varOfSomeType : SomeCollectionVariable) Statement

for (SomeType varOfSomeType : ArrayVariableOfSomeType) Statement

for (SomeType varOfSomeType : EnumSet) Statement

These versions of "for" allow you to cycle easily through all elements in a collection or an array or an enumeration, or subranges of an enumeration. They are covered in Chapter 16, "Collections," Chapter 9, "Arrays," and Chapter 6, "Static, Final, and Enumerated Types."

Statement Notes

✓

Initial, Test, and Increment are all expressions that control the loop. The semicolons are required, but any or all of the expressions are optional. The Test expression must be a boolean or a Boolean. A typical loop will look like this:

```
for( i=0; i<100; i++ ) { /*loop
body*/  }
```

An infinite loop will have blank Initial Test and Increments, like this:

```
for (;;)
```

Transfer of Control Statements

A "return" statement looks like this:

```
return;
```

or

```
return  Expression;
```

Statement Notes

✓ "Return" (with no accompanying expression) is only used in a method that has a return value of void. It gets you out of the method and back to where you were called from. The code below is an example:

```
void resetFields(Customer c) {

   if (c==null) return;

   // otherwise, go on to reset the
fields of c

}
```

✓

Comments

Java has comment conventions similar to those of C++. Two slashes together, "//" make the rest of the line a comment, as follows:

```
i = 0;  // the "to end-of-line" comment
```

Comments starting with "/*" make all the characters up to and including the first "*/", be a comment. It might stretch over several lines:

```
/* the "regular multiline" comment

   goes here.

 */
```

Commenting out code

Since comments do not nest in Java, to comment out a big section of code, you must either put "//" at the start of every line, or use "/*" at the front and immediately after every embedded closing comment, finishing up with your own closing comment at the end.

You can also use the following around statements you want to temporarily delete:

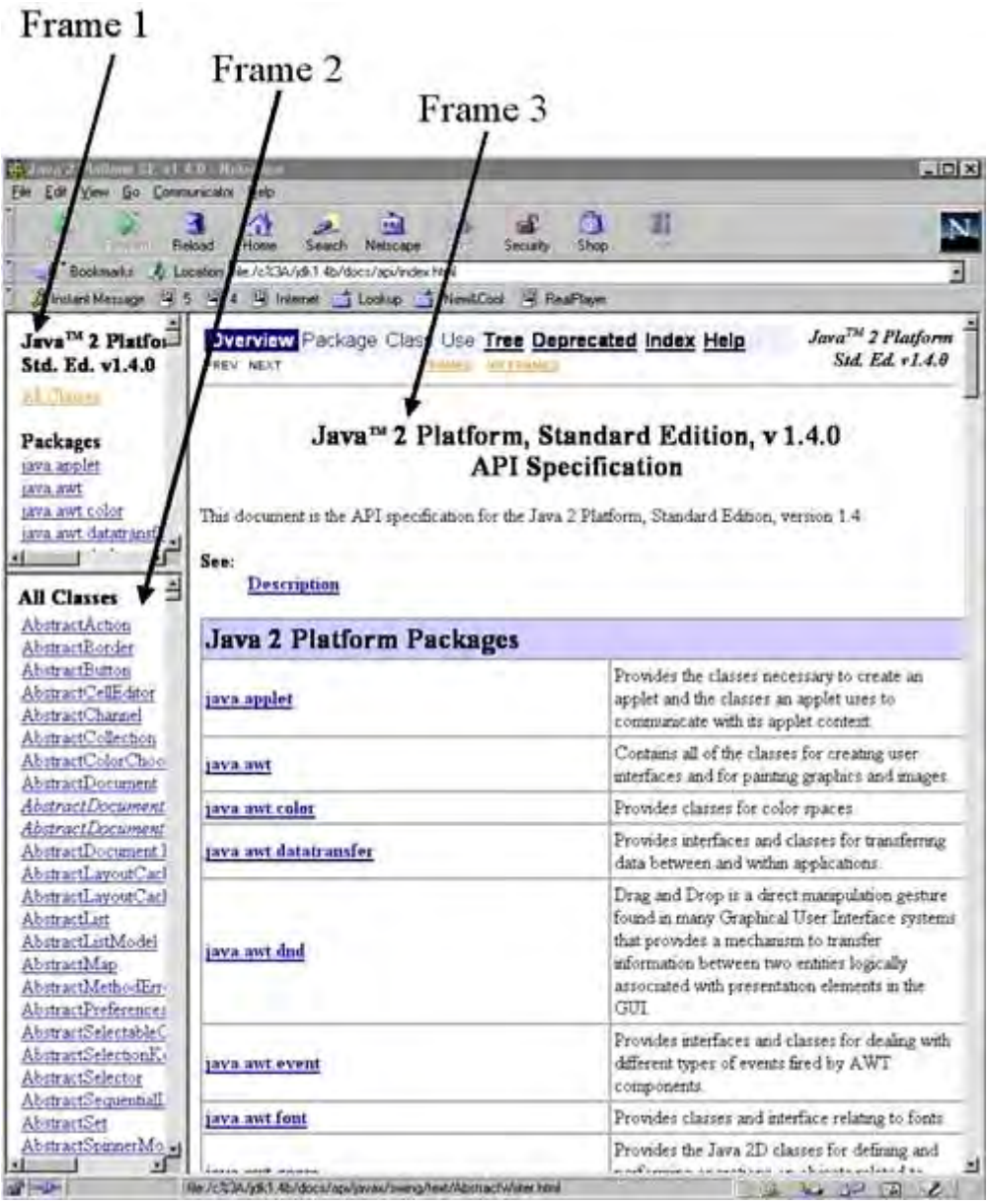
```
if ( false ) {

    ...

}
```

Reading the Java API

This is a very, very important section! The Java Application Programmer Interface (API) is a vast collection of libraries. Luckily, it is well documented, and the documentation is easily accessed using a browser starting at http://java.sun.com/docs/index.html

You can also download the html and install it locally. I prefer to download it because local access is faster than web access, and we're mostly Type A individuals in programming. However you do it, start up your favorite browser and browse the URL for the Java API for the release you installed on your computer. When the page comes up, set a bookmark for it. Your browser should be displaying something like Figure 4-1.

Figure 4-1. The API documentation



[View full size image]

Exercises

1. Write the if statements that call different methods do0_9(), do10_99(), and do100_999() based on the value of i being in the range 0-9, 10-99, or 100-999. Don't forget to allow for other values of i, including negative values.
2. Write a switch statement that corresponds to the if statement in question 1. Which is more compact? Which is easier to read?
3. Write a method that counts the number of "1" bits in an int value. You can get successive bits from the end of an int by checking to see if the int is an odd number. An int is an even number if a division by two followed by a multiplication by two gives you back the original number (truncation of a fraction gives a smaller result for an odd number). If it is an odd number, the least significant bit is 1, otherwise it is zero. Then divide the int by 2 to get the next bit. Don't do this check-and-divide-by-two more than 31 times. Why shouldn't you do the shift more than 31 times with an int?
4. Write another method that counts the number of "1" bits in a long value. Is there any way to share code between the two? (Hint: a long is two ints wide.) Would you do so in practice? Explain why or why not. Then look up the javadoc API and reimplement it using java.math.BigInteger.bitCount().

Some Light Relief—MiniScribe: The Hard Luck Hard Disk

Most readers will know the term "hard disk," which contrasts with "floppy disk," but how many people know about MiniScribe's pioneering efforts in the fields of very hard disks, inventory control, and accounting techniques?

MiniScribe was a successful start-up company based in Longmont, Colorado, that manufactured disk drives. In the late 1980s, Miniscribe ran into problems when IBM unexpectedly cancelled some very big purchasing contracts. The venture capitalists behind MiniScribe, Hambrecht & Quist, brought in turnaround expert Q.T. Wiles to get the company back on track.

Wiles mercilessly drove company executives to meet revenue targets, even as sales fell still further. In desperation, the beleaguered executives turned to outright record falsification. It must have seemed so easy. Over the space of a couple of years they came up with an impressive range of fraudulent techniques for making a failing company have the appearance of prospering.

The Miniscribe executives started off with the easy paper-based deceit, like:

- Counting raw inventory as finished goods (with a higher value).
- Anticipating shipments before they were made to customers.
- Counting imaginary shipments on non-existent ships.

When they were not found out, they graduated to more brazen activities like parading inventory past the accountants twice, so it would be counted twice, and shipping obsolete product to a fake customer called "BW." "BW" stood for "Big Warehouse" and was a MiniScribe storage building. And so it went, with smaller crimes leading to bigger crimes, just the way your kindergarten teacher warned it would.

Miniscribe employed more than 9,000 people worldwide at the height of its fortunes, so this was no fly-by-night, two-guys-in-a-garage undertaking. This was a fly-by-night, 9,000-guys-in-a-Big-Warehouse undertaking. The companies that supplied Miniscribe were doing less and less business with them and were finding it hard to get paid. One analyst surveyed the entire computer industry and found only one large MiniScribe customer. At the same time, MiniScribe was issuing press releases talking about "record sales."

The most breathtaking coup, though, was the brick scam. Desperate to show shipments on the books, executives brought in their assistants, spouses, and even children for a crazy weekend of filling disk shipping boxes with house bricks. They also created a special computer program called "Cook book" (these guys were well aware of what they were doing) that facilitated shipping the bricks to good old "BW" and recognizing the "revenue" from that customer. These bricks were surely the ultimate hard drive.

Chapter 5. OOP Part II—Constructors and Visibility

All the way back in Chapter 2, we briefly mentioned two concepts that are used a lot with classes: constructors (to create new objects), and access control (to deliberately restrict or increase the visibility of classes and things in classes). This chapter is where we deliver the full details on these two topics. Constructors are usually described as being like methods "with a few differences" and method is the object-oriented name for a function or procedure. So there's a section on methods and parameter passing in here too. We look at the ways you can organize groups of classes into packages, and how you can store packages in jar files. At the end of this chapter, you'll have enough knowledge to read and write basic Java object-oriented programs.

Polymorphism Is a Long Word for a Short Topic

In Chapter 8, we cover inheritance and polymorphism. Don't be put off by the term polymorphism. It's just a long word meaning "how child classes provide specialized versions of the methods inherited from the parent". You'll need to read and understand the advanced OOP chapter be an effective Java programmer. The good news is that object-oriented programming is based on a few simple ideas. Here are the key OOP ideas that we have met so far:

Object-oriented programming: key ideas so far

- A class is another name for a datatype.
- An object is a variable belonging to a class datatype.
- The methods and data defined in a class implement the operations of the type. The compiler permits only these operations on objects of this type. You have to provide a specific object when you call a method. The statements are executed on that specific object.
- Every class has a parent class, and has full access to the data and operations of its parent (access to your parent's stuff is called "inheritance").

We also saw that Java object variables are really just pointers to objects, "references" in Java terminology. This is a big difference between Java and most other object-oriented programming languages. Some implications that flow from this design decision are summarized in the next section.

Reminder: Java object variables are really references to objects

- All objects are accessed through references (these are, effectively, pointers).
- Declaring what looks like an object variable actually gets you a variable that can hold a reference to an object.
- When you declare that variable, it contains a null value that does not point to any object. To start using the object, you must first make the variable point to an object. You can make it point to an existing object, or you can create a new object for it to reference.

We have reinforced the point that object variables don't start out pointing to an object. It's time to describe in detail how you create a new object when you want one.

Creating New Objects: Constructors

You call a constructor to create a new object. A constructor is a special kind of method that you write as part of a class. It looks like an ordinary method, but a constructor magically allocates memory for a new instance of an object, then executes the statements that you write in the constructor body. Your statements typically initialize the fields of the new object.

Like any method, a constructor needs to be given a name. A constructor also needs to indicate that its return type is "an object of the class that this constructor belongs to". The Java design team chose to roll these two needs together, by adopting the conventions that:

- A constructor has the same name as the class it belongs to.
- A constructor is written without an explicit return type. In some sense, its name is the return type.

The purpose of a constructor is to allocate and initialize a newly created object. Here's the Timestamp class with the obvious constructor:

```
class Timestamp {
        int hrs;
        int mins;
        int secs;

        // a constructor returns a new Timestamp object
        // implicitly allocates memory for the object
        // explicitly initializes the fields of the object
        Timestamp() {
            java.util.Calendar now =
                        java.util.Calendar.getInstance();
            hrs = now.get(java.util.Calendar.HOUR_OF_DAY);
            mins = now.get(java.util.Calendar.MINUTE);
            secs = now.get(java.util.Calendar.SECOND);
```

More About Methods

Methods are the OOP name for functions. A method is always declared inside a class; methods can't exist outside a class. A method has this general form:

[View full size image]

The parts marked "opt" in the previous example are optional, and we will deal with these in due course. An example method may look like this:

```
void setValue(int i) {

     something = i;

}
```

The main routine, where execution starts, is another example:

Arguments and parameters

What's the difference between an argument and a parameter? Here's a bit of terminology which is frequently ignored. A parameter is any variable declared within the method signature. The list of parameters is enclosed in parentheses. Each parameter consists of a type name followed by a variable name. In the code fragments shown previously, i is an int parameter to setValue(), and args is a String array parameter to main(). Parameters are sometimes called "formal parameters".

Arguments only appear in calls to a method. An argument is the actual value used in a particular call to a function. If we invoke setValue twice like this:

Variable-Arity Methods

Arity is a computer science term meaning the number of arguments a method or operator takes. It was originally a joking term, taken from "polarity" meaning the number of poles something has, so by extension "arity" must mean "number of". Well, just like your mother told you if you pulled a face it would stick like that, the term arity stuck.

In some languages, methods may have variable-arity which means their last or only argument is actually a list of arguments. Congratulations; with JDK 1.5, Java is now one of the languages with variable-arity! This is another hack that was put in Java because some engineers really liked C's printf() function, and wanted Java to have the same convenience.

Luckily, variable-arity (also called var-args by some) has been implemented in a very straightforward way in Java. Here's the code you write to tell the compiler "this is a variable-arity method" in Java:

```
public PrintWriter format(String format, Object... args)
```

This declaration actually comes from the API class java.io.PrintWriter. The three dots tell the compiler "args is a sequence of Objects", so this is a variable-arity method. Inside the body of format(), the args parameter is regarded as type "array of Object", i.e. it is exactly as though the method were declared:

```
public PrintWriter format(String format, Object[] args)
```

Where you see a difference is in calls to variable-arity methods. If the last parameter is declared with those three dots known as an ellipsis, then you can call the method with a variable number of objects as that last parameter. They will be automatically assembled into an array for you.

So you can make a call like this:

```
myPW.format( myFormatStr, t1, t2 );
```

Packages

There are several keywords that control the visibility of a class or the members of a class. Access Modifiers on page 103 summarizes these modifiers (private, no keyword, protected, and public). Some of these access modifiers apply to packages, so we'll start with the "how and why" of Java packages.

What a package is

When you're just writing a few programs for your personal use, you can put the classes in any old directory and use any old names. When you have a team of twenty programmers working on five different software products that interact, you need a better way to organize your files than "putting them in any old directory."

Java uses the term package to mean a collection of related classes that form a library, and which are kept together in the same directory. Strictly speaking, packages and classes do not have to be directories and files. The Java Language Specification is written to allow implementations to store code in a database or any other kind of repository, and some IDEs do just that. However, "package equals directory" and "class equals file" is the simplest implementation, and the best way to understand the topic.

Package naming rules

A package name needs to match the directory name where the source and class files are stored. So a package called "java.lang" must have its class files in a directory where the last part of the pathname is java\lang (on Windows) or java/lang (on Unix, Linux, the Mac, etc). Java uses the computer's filesystem to organize and locate packages.

The requirement means that if you know the name of a class, you also know where to find it (as long as you know where to begin looking). Classes that are part of the same package will be in the same directory. It simplifies things for the compiler-writer, and gives the programmer less to remember.

Packaging classes

How do you tell the compiler that a class D is part of some package, like a.b.c ? It's simplicity itself—just put the source in file a/b/c/D.java and write the first few lines in the file like this:

```
package a.b.c;

class D  {

          /* more code */

}
```

How the JDK Finds Classes

Unlike many other language or compiler systems, Java doesn't link all your code into one big, freestanding program. Instead, it leaves everything in separate classfiles. Packages, and hence directory paths, impose some order on that. The JDK always knows where to find the Java run-time library classes, but the run-time class loader still needs to know where to find your classes.

The first point to note is that you can zip all your class files into a jar file, so you don't really need umpteen levels of directory on the customer's system. A jar file (Java Archive) works exactly like a zip file and can be used to store a whole directory tree in a single file. You can put other files that the application needs, such as image files, sound files, and translations for Strings, in the jar, too. Then you have to put the jar file (or the complete directory tree) where Java can find it, or tell Java the possible places to look for it. You can even set a jar file up so it will execute when its icon is double-clicked.

How to create a jar file

You can create your own jar files by using standard WinZIP, other software, or the jar utility that comes with the JDK. The jar program has a command line that looks like this in general form:

```
jar [ options ] [manifest] destination input-file [input-files]
```

The options that jar takes are similar to those of tar—the Unix tape archive utility—but the formats are different, and tar files are not used in Java. To create a compressed archive of all the class files and .jpg files for package a.b.c, you would use the commands

```
cd parent-of-a

jar cvf myJarFile.jar a/b/c/*.class a/b/c/*.jpg
```

Don't forget that once you put some code in a jar, it's not enough to recompile when you change something. You must also rebuild the jar file with the new .class; otherwise, you'll continue to get the old version of the program. Don't encrypt the jar file. Since JDK 1.2 there has been an option to update (replace a file in) an existing jar file. The example below shows how to replace the file Foo.class in archive myJarFile.jar.

Access Modifiers

It's important to know the information in this section, but you can skim through it on the first reading. It explains how we can use keywords to make things more widely or narrowly visible.

There are three or four keywords that are applied to class members. These keywords are collectively called the access modifiers. By default, a member is only visible to classes in its own package. Certain keywords make them visible inside the class (only), or inside the inheritance hierarchy, or to classes in other packages.

Access modifiers for a class

Class declarations can be nested inside other classes. It's a design choice you'd make when the nested class is a helper or utility for the top level class. E.g. the top level class java.lang.Character contains a nested class called java.lang.Character.UnicodeBlock.

```
package java.lang;

public class Character {   /* a top level class */

      /* lots of code */

      public class UnicodeBlock { /*a nested class*/  }

}
```

That nested class organizes and gives names to several chunks of Unicode characters. You've got your Basic Latin block, your Extended Latin block, your Greek block, your Cyrillic block, and so on all the way to the Chinese and Japanese character blocks. The UnicodeBlock class was put inside Character because it provides some services intimately tied to Character. However it is public so anyone can use it (e.g. to test if a certain character is in the range of Greek characters).

Most of the classes you write will be top level (not nested) classes. Top level classes are always visible to other classes in the same package. They are only visible in other packages when you prefix the class definition with the access modifier "public". The term "visible" means you can see and access the resource (call a method, read a field, whatever it is). Here are the two cases of visibility of a top level class:

- If you don't use any access modifier

Exercises

1. Distinguish between class and object, member and field, primitive type and class type.
2. Write a sample program that demonstrates the compiler will not let you access a private member from another class, but that you can write a public accessor function to set the value of a private field. (Compile the first part, compile and run the second part.)
3. Write a class with a static block, some constructors, and instance initializers that print out the order in which these are executed.
4. Describe why a setSomething() method is often given protected visibility.

Some Light Relief—It's Not Your Father's IBM

Probably the biggest single supporter of Java is IBM. The largest computer company on the planet demonstrates its support with products, with research, with free downloadable code, and with open participation in the user community. IBM's Java initiative mirrors its Linux initiative. The company supports these two technologies for the same reasons: software compatibility across product lines, freedom from proprietary operating systems that IBM doesn't own, and recognition that these technologies have become a major force in IT.

IBM announced in 2000 that it would invest $1 billion in Linux, and did exactly that. In 2004, IBM announced a plan to migrate all its internal desktop PCs to Linux! This isn't "the standard is whatever we sell" IBM of a generation ago. The company is showing a new face that is lively, cooperative, and engaging. For example, to show how the Linux operating system is viable across all platforms—from large enterprise servers to the smallest embedded devices—IBM demonstrated a wristwatch-sized Linux device at a San Jose conference in August 2000 (see Figure 5-1). The wristwatch had wireless capability, and could be used to read email. It ran Linux 2.2 and the X window system!

Figure 5-1. The IBM Linux wristwatch

Now, granted a wristwatch running Linux is not the big deal today that it was in 2000, but it's a measure of IBM's commitment to Linux. On the Java side, IBM has released a large amount of useful software through its Alphaworks early release research program. One Java package that you can download from Alphaworks is the Robocode software.

Robocode is an intriguing "learn Java while having fun with a game" system. Recreational software is one of the last things I would have expected to come out of IBM! The game part is shown in the screenshot in Figure 5-2. There is a

Chapter 6. Static, Final, and Enumerated Types

Enumerated types were brought into Java with the JDK 1.5 release. They are not a new idea in programming, and lots of other languages already have them. The word "enumerate" means "to specify individually". An enumerated type is one where we specify individually (as words) all the legal values for that type.

For a type that represents t-shirt sizes, the values might be small, medium, large, extraLarge. For a bread flavors type, some values could be wholewheat, ninegrain, rye, french, sourdough. A DaysOfTheWeek enumerated type will have legal values of Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday.

The values have to be identifiers. In the USA, ladies' dress sizes are 2, 4, 6, 8, 10, 12, 14, 16. As a Java enumeration, that would have to be represented in words as two, four, or any other characters that form an identifier, such as size2, size4 etc.

When you declare a variable that belongs to an enumerated type, it can only hold one value at a time, and it can't hold values from some other type. A t-shirt size enum variable can't hold "large" and "small" simultaneously, just as an int can't hold two values simultaneously. You can't assign "Monday" to a t-shirt size variable. Though enumerated types aren't essential, they make some kinds of code more readable.

enum is a new keyword

Although JDK 1.5 introduced extensive language changes, "enum" is the only new keyword brought into

What Field Modifier static Means

We have seen how a class defines the fields and methods that are in an object, and how each object has its own storage for these members. That is usually what you want.

Sometimes, however, there are fields of which you want only one copy, no matter how many instances of the class exist. A good example is a field that represents a total. The objects contain the individual amounts, and you want a single field that represents the total over all the existing objects of that class. There is an obvious place to put this kind of "one-per-class" field too—in a single object that represents the class. Static fields are sometimes called "class variables" because of this.

You could put a total field in every object, but when the total changes you would need to update every object. By making total a static field, any object that wants to reference total knows it isn't instance data. Instead it goes to the class and accesses the single copy there. There aren't multiple copies of a static field, so you can't get multiple inconsistent totals.

Static is a really poor name

Of all the many poorly chosen names in Java, "static" is the worst. The keyword is carried over from the C language, where it was applied to storage which can be allocated statically (at compile time). Whenever you see "static" in Java, think "once-only" or "one-per-class."

What you can make static

You can apply the modifier static to four things in Java:

- Data. This is a field that belongs to the class, not a field that is stored in each individual object.
- Methods. These are methods that belong to the class, not individual objects.
- Blocks. These are blocks within a class that are executed only once, usually for some initialization. They are like instance initializers, but execute once per class, not once per object.
- Classes. These are classes that are nested in another class. Static classes were introduced with JDK 1.1.

We'll describe static data, static methods in this chapter. Static blocks, static classes we'll deal with later.

What Field Modifier final Means

This section looks at final, which makes something constant. Why was the word "const" or "constant" not chosen? Because "final" can also be applied to methods, as well as data, and the term "final" makes better sense for both.

A class or a class member (that is, a data field or a method) can be declared final, meaning that once it is given a value it won't change. We will look at what it means for a class or a method not to change in Chapter 8. A couple of final data declarations are:

```
final static int myChecksum = calculateIt();

final Timestamp noon = new Timestamp(12, 00, 00);

final int universalAnswer = 42;
```

When a reference variable is declared final, it means that you cannot change that variable to point at some other object. You can, however, access the variable and change its fields through that final reference variable. The reference is final, not the referenced object.

JDK 1.1 introduced the ability to mark method arguments and variables local to a method as final, such as:

```
void someMethod(final MyClass c, final int a[]) {

     c.field = 7;       // allowed

     a[0] = 7;          // allowed

     c = new MyClass(); // final means this line is NOT allowed

     a = new int[13];   // final means this line is NOT allowed

}
```

Programmers rarely use this, because it clutters up the signature and makes the parameter names harder to read. That's a pity. Marking a declaration as final is a clue to the compiler that certain optimizations can be made. In the case of final primitive data, the compiler can often substitute the value in each place the name is used, in an optimization known as constant propagation. As my friend, talented compiler-writer and builder of battle robots Brian Scearce pointed out, that in turn may lead to other optimizations becoming possible.

Why Enumerate a Type?

Here's the older approach of simulating enumerations:

```
class Bread {

    static final int wholewheat = 0;

    static final int ninegrain = 1;

    static final int rye = 2;

    static final int french = 3;

}
```

You would then declare an int variable and let it hold values from the Bread class, e.g.

```
int todaysLoaf = Bread.rye;
```

Drawbacks of using ints to enumerate

Using final ints to represent values in an enumeration has at least three drawbacks.

- All the compiler tools (debugger, linker, run-time, etc.) still regard the variables as ints. They are ints. If you ask for the value of todaysLoaf, it will be 2, not "rye". The programmer has to do the mapping back and forth mentally.
- The variables aren't typesafe. There's nothing to stop todaysLoaf getting assigned a value of 99 that doesn't correspond to any Bread value. What happens next depends on how well the rest of your code is written, but the best case is that some routine notices pretty quickly and throws an exception. The worst case is that your computer-controlled bakery tries to bake "type 99" bread causing an expensive sticky mess.
- Use of integer constants makes code "brittle" (easily subject to breakage). The constants get compiled into

Statements Updated for Enumerations

Two statements have been changed to make them work better with enumerations: the switch statement, and the for statement.

Using a for statement with enums

Language designers want to make it easy to express the concept "Iterate through all the values in this enumeration type". There was much discussion about how to modify the "for" loop statement. It was done in a clever way—by adding support for iterating through all the values in any array.

Here is the new "get all elements of an array" syntax added to the "for" statement.

```
// for each Dog object in the dogsArray[] print out the dog's name

for (Dog dog : dogsArray ){

    dog.printName();

}
```

This style of "for" statement has become known as the "foreach" statement (which is really goofy, because there is no each or foreach keyword). The colon is read as "in", so the whole thing is read as "for each dog in dogsArray". The loop variable dog iterates through all the elements in the array starting at zero.

Zermelo-Fränkel set theory and you

Keywords don't have to be reserved words. So keywords don't have to be prohibited for use as identifiers. A compiler can look at the tokens around a keyword and decide whether it is being used as a keyword or an identifier. This is called context-sensitive parsing.

I would have preferred an actual keyword "in" rather than punctuation, but the colon has been used this way in Zermelo-Fränkel set theory by generations of mathematicans. It's just syntactic sugar. And if we've always done something this way, hey, don't not go for it.

Using the new foreach feature, we can now write a very brief program that echoes the arguments from the command

More Complicated Enumerated Types

Since enums are effectively classes, you can do pretty much everything with them that you can do with a class. In particular, you can provide one or more constructors for an enum type! That may seem a little weird to you (it does to me), because you never call an enum constructor anywhere in your code, or use the "new" operator to instantiate an enum. The definition of an enum brings the class enum constants into existence. You always work with static fields of an enum, not instances.

You might want to write a constructor when you have enumerations with a close relationship to numeric values. An example is an enumeration of hens' egg sizes, shown in Table 6-1. In the U.S., these are the names and weights associated with eggs:

Table 6-1. U.S. names and weights associated with eggs

Name	**weight per dozen**
Jumbo	30 oz
Extra large	27 oz
Large	24 oz

Adding constructors to your enums

We're going to create an enum class called egg, with enum constants of jumbo, etc. You can tie arbitrary data to each enum constant by writing a constructor for the enum class. In other circumstances, a constructor has the same name as the class, so you'd expect it to be called egg here. And indeed it is, but that's not the full story. You declare an enum constant (jumbo, etc) and that name declaration is regarded as a call to your constructor. You may pass data values as arguments to the call. Arguments are passed in the usual way, by enclosing the comma-separated list in parentheses.

Putting it together, the enum now looks like this:

```
enum egg {

            // the enum constants, which "call" the constructor

            jumbo(30.0),

            extraLarge(27.0),
```

Some Light Relief—The Haunted Zen Garden of Apple

I've always made a point of exploring new and unfamiliar places, particularly when I'm trying a new shortcut and forgot to bring the map with me. My colleague Lefty is just the opposite. Lefty (who I taught to count on his fingers in binary—you can count up to 1023 that way) likes to stick to paths so well beaten they are practically pulverized.

Lefty and I were on the Apple Computer campus in Cupertino, California. If you know the area, we were in the Bandley 3 building, and late for a meeting in nearby De Anza 7. You can almost see these two buildings from each other, so I was practically sure that if we jogged through the parking lots as the crow flies, we'd get to De Anza 7 in the shortest amount of time.

"I hope you're not trying one of your shortcuts," accused Lefty as he sprinted behind me, vaulting over fences and bushes. "Don't be ridiculous!" I retorted. We would have arrived almost on time too, if my shortcut hadn't taken us through a small and mysterious grove of trees at the side of the De Anza 7 parking lot. Lefty ran into the grove, then straight into a large waist-high stone.

Figure 6-2. The Haunted Zen Garden of Apple (De Anza 7 in background)

The big southpaw went down with a curse that would curl the hair of a software architect. While he threshed on the ground, mouthing incoherent criticism of me and shortcuts, I examined the stone with interest. It appeared to be a tombstone covered in Japanese writing.

I tried to redirect his attention to the mysterious stone "Check this out Lefty," I noted, "It's just like that slab on Jupiter in 2001!". His reply cannot be printed in a text that family members may read. Looking around more widely, I was very surprised to see that the grove was an abandoned Zen garden. My first thought was to wonder if the tombstone meant

Chapter 7. Names, Operators, and Accuracy

This chapter covers more of the language basics. The information here follows a progression: first, the basics of names and identifiers, then how names are connected in expressions by operators, followed by the rules to evaluate expressions.

The last part of the chapter describes practical limits on accuracy, casting and conversions. If you are new to floating point arithmetic, some of this may come as a bit of a shock. The chapter finishes up by presenting the standard class java.lang.Math.

Keywords

Keywords are reserved words, which means they cannot be used as identifiers. Java now has 50 keywords ("enum" became a keyword in JDK 1.5). These are the keywords.

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

The keywords const and goto are reserved words, even though they are not used anywhere in Java. This allows future expansion, or (unlikely but claimed) better diagnostic messages if a programmer uses them in a way they might be used in C++.

The words "true" and "false" look like keywords, but technically they are boolean literals. Similarly,"null" looks like it should be in the keyword list, but it is technically the null literal for reference types. The classification of these three words as literals, not keywords, doesn't have any subtle side effects, and it keeps the type system a bit cleaner.

Punctuation Terminology

You might find it useful to refer back to this table in the chapters ahead.

These characters are called parentheses: ( ). They go around expressions and parameter lists.

These characters are called square brackets: [ ]. They go around array indexes.

These characters are called braces or curly braces: { }. They indicate the start and end of a new block of code, or an array literal.

These characters are called angle brackets: < >. They go around generic parameters.

The ellipsis token ... came into Java in JDK 1.5. It indicates the last argument to a method is a sequence of objects.

Names

What is the difference between an identifier and a name? An identifier is just a sequence of letters and digits that don't match a keyword or the literals "true," "false," or "null." A name, on the other hand, can be prefixed with any number of further identifiers to pin-point the namespace from which it comes. An identifier is thus the simplest form of name. The general case of name looks like the following:

```
package1.Package2.PackageN.Class1.NestedClass2.NestedClassM.memberN
```

Since packages can be nested in packages, and classes nested in classes, there can be an arbitrary number of identifiers separated by periods, as in:

```
java.lang.System.out.println( "goober" );
```

That name refers to the java.lang package. There are several packages in the java hierarchy, and java.lang is the one that contains basic language support. One of the classes in the java.lang package is System. The class System contains a field that is an object of the PrintStream class, called out. PrintStream supports several methods, including one called println() that takes a String as an argument. It's a way to get text sent to the standard output of a program.

By looking at a lengthy name in isolation, you can't tell where the package identifiers stop and the class and member identifiers start. You have to do the same kind of evaluation that the compiler does. Since the namespaces are hierarchical, if you have two identifiers that are the same, you can say which you mean by providing another level of name. This is called qualifying the name. For example, if you define your own class called BitSet, and you also want to reference the class of the same name that is in the java.util package, you can distinguish them like this:

```
          BitSet myBS = new BitSet();
java.util.BitSet theirBS = new java.util.BitSet();
```

java.util qualifies the class name BitSet.

Identifiers

Identifiers are the names provided by the programmer, and can be any length in Java. They must start with a letter, underscore, or dollar sign, and in subsequent positions can also contain digits.

A letter that can be used for a Java identifier doesn't just mean uppercase and lowercase A–Z. It means any of the tens of thousands of Unicode letters from any of the major languages in the world including Bengali letters, Cyrillic letters, or Bopomofo symbols. Every Unicode character above hex 00C0 is legal in an identifier. Legal Java identifiers shows some example valid Java identifiers.

Table 7-1. Legal Java identifiers

calories	Häagen_Dazs	déconnage
_99	Puñetas	fottío
i	$__	p

The more accented characters you use in your variable names, the harder it is for others to edit them and maintain the code. So you should stick to the simple ASCII characters.

When a field or method can be forward-referenced

A forward reference is the use of a name before that name has been defined, as in the following:

```
class Fruit {
     void setWeight() { grams = 22; } // grams not yet declared

     int grams;
}
```

The example shown compiles with no problem. You can declare fields in any order, with one exception. The rule of thumb is that the compiler needs to see a field before it is used in the initialization of another field:

Expressions

There's a lengthy chapter in the Java Language Specification on expressions, covering many cases that would be interesting only to language lawyers. What it boils down to is that an expression is any of the alternatives shown in Table 7-2.

Table 7-2. Expressions in Java

Expression	Example of expression
A literal	245
this object reference	this
A field access	now.hh
A method call	now.fillTimes()
An object creation	new Timestamp(12, 0, 0)
An array creation	new int[27]
An array access	myArray[i][j]
Any expression connected by operators	now.mins / 60
Any expression in parens	(now.millisecs * 1000)

You evaluate an expression, to get a result that will be:

- A variable (as in evaluating this gives you an object you can store into), or
- A value, or
- Nothing (a void expression). You get nothing as the result when you call a method with a return value of void.

An expression can appear on either side of an assignment. If it is on the left-hand side of an assignment, the result designates where the evaluated right-hand side should be stored. Here's an example:

Arrays

This is just a confirmation that Java has arrays. When you see a pair of square brackets like this [ ], it always means you're dealing with an array. The full description of arrays has its own chapter, Chapter 9.

An example array is the set commandline arguments, which is passed to the main() routine as an array of Strings.

```
void main(String[] args) { /* code */ }
```

The individual elements in an array can be any type: primitive or reference type. The array as a whole is an object in Java. That means array types are reference types, and an array variable is really a reference to an array object.

Operators

An operator is a punctuation mark that says to do something to two (or three) operands. An example is the expression "a * b". The "*" is the multiplication operator, and "a" and "b" are the operands. Most of the operators in Java will be readily familiar to any programmer.

One unusual aspect is that the order of operand evaluation in Java is well-defined. For many older languages, the order of evaluation was been deliberately left unspecified to allow the compiler-writer more freedom. In other words, in C and C++, the following operands can be evaluated and added together in any order:

```
i + myArray[i] + functionCall();
```

The function may be called before, during (on adventurous multiprocessing hardware), or after the array reference is evaluated, and the additions may be executed in any order. If the functionCall() adjusts the value of i, the overall result depends on the order of evaluation. The trade-off is that some programs give different results depending on the order of evaluation. A professional programmer would consider such programs to be badly written, but they exist nonetheless.

The order of evaluation was left unspecified in earlier languages so that compiler-writers could reorder operations to optimize register use. Java makes the trade-off in a different place. It recognizes that getting consistent results on all computer systems is a much more important goal than getting varying results a trifle faster on one system. In practice, the opportunities for speeding up expression evaluation through reordering operands seem to be quite limited in many programs. As processor speed and cost improve, it is appropriate that modern languages optimize for programmer sanity instead of performance.

Java specifies not just left-to-right operand evaluation, but the order of everything else, too, such as:

- The left operand is evaluated before the right operand of a binary operator. This is true even for the assignment operator, which must evaluate the left operand (where the result will be stored) fully before starting any part of evaluating the right operand (what the result is).
- An array access has this general form:

Associativity

Associativity is one of those subjects that is poorly explained in most programming texts. In fact, a good way to judge a programming text for any language is to look for its explanation of associativity. Silence is not golden.

There are three factors that determine the ultimate value of an expression in any algorithmic language, and they work in this order: precedence, associativity, and order of evaluation.

Precedence says that some operations bind more tightly than others. Precedence tells us that the multiplication in a + b * c will be done before the addition, i.e., we have a + (b * c) rather than (a + b) * c. Precedence tells us how to bind operands in an expression that contains different operators.

Associativity is the tie breaker for deciding the binding when we have several operators of equal precedence strung together. If we have 3 * 5 % 3, should we evaluate it as (3 * 5) % 3 or as 3 * (5 % 3) ?

The first is 15 % 3, which is 0. The second is 3 * 2, which is 6! Multiplication and the "%" remainder operation have the same precedence, so precedence does not give the answer. But both operators *, % are left-associative, meaning when you have a bunch of them strung together you start associating operators with operands from the left. Push the result back as a new operand, and continue until the expression is evaluated. In this case, (3 * 5) % 3 is the correct grouping.

There is no extra charge for parentheses. Use parentheses generously in expressions, so that future programmers (and the compiler) can avoid imposing its own view about how you wanted the operands grouped.

Associativity is a terrible name for the process of deciding which operands belong with which operators of equal precedence. A more meaningful description would be, "Code Order For Finding/Evaluating Equal Precedence Operator Textstrings." This is the "COFFEEPOT property" mentioned in Table 7-3.

Note that associativity deals solely with deciding which operands go with which of a sequence of adjacent operators of equal precedence. It doesn't say anything about the order in which those operands are evaluated.

Order of evaluation, if it is specified in a language, tells us the sequence for each operator in which the operands are evaluated. In a strict left-to-right language like Java, the order of evaluation tells us that in (i=2) * i++, the left operand to the multiplication will be evaluated before the right operand, then the multiplication will be done, yielding a result of 4, with i set to 3. Why isn't the auto-increment done before the multiplication? It has a higher precedence after all. The reason is because it is a post increment, and so by definition the operation is not done until the operand has been used.

In C and C++, this expression is undefined because it modifies the same value more than once within an expression. It is legal in Java because the order of evaluation is well defined.

How Accurate Are Calculations?

The accuracy when evaluating a result is referred to as the precision of an expression. The precision may be expressed either as number of bits (64 bits), or as the data type of the result (double precision, meaning 64-bit floating-point format).

The precision of an expression

In Java, the precision of evaluating an operator depends on the types of its operands. Java looks at the types of the operands around an operator and picks the biggest of what it sees: double, float, and long, in that order of preference. Both operands are then promoted to this type, and that is the type of the result. If there are no doubles, floats, or longs in the expression, both operands are promoted to int, and that is the type of the result. This continues from left to right through the entire expression.

A Java compiler follows this algorithm to compile each operation:

- If either operand is a double, do the operation in double precision.
- Otherwise, if either operand is a float, do the operation in single precision.
- Otherwise, if either operand is a long, do the operation at long precision.
- Otherwise, do the operation at 32-bit int precision.

In summary, Java expressions end up with the type of the biggest, floatiest type (double, float, long) in the expression. They are otherwise 32-bit integers.

Limited significant figures in floating-point numbers

The precision tells you how many bits you get in your answer. But if your calculation involves floating point, you also have to be wary about the limited accuracy of the answer. Accuracy is not just the range of values of a type, but also (for real types) the number of significant figures that can be stored. The type float can store some numbers exactly, but in general you can only count on about six to seven digits of significant figures. When a long (which can hold at least 18 places of integer values) is implicitly or explicitly converted to a float, some precision may be lost there too. Here's an example showing loss of precision when floating a long and casting back.

```
public class inexact2 {
```

Widening and Narrowing Conversions

This section provides more details on when a cast is needed, and also introduces the terminology of type conversions. This is explained in terms of an assignment between one variable and another, and exactly the same rules apply in the transfer of values from actual parameters to formal parameters.

When you assign an expression to a variable, a conversion must be done. Conversions among the primitive types are either identity, widening, or narrowing conversions.

- Identity conversions are an assignment between two identical types, like an int to int assignment. The conversion is trivial: just copy the bits unchanged.
- Widening conversions occur when you assign from a less capacious type (such as a short) to a more capacious one (such as a long). You may lose some digits of precision when you convert either way between an integer type and a floating point type. An example of this appeared in the previous section with a long-to-float assignment. Widening conversions preserve the approximate magnitude of the result, even if it cannot be represented exactly in the new type.
- Narrowing conversions are the remaining conversions. These are assignments from one type to a different type with a smaller range. They may lose the magnitude information. Magnitude means the largeness of a number, as in the phrase "order of magnitude." So a conversion from a long to a byte will lose information about the millions and billions, but will preserve the least significant digits.

Widening conversions are inserted automatically by the compiler. Narrowing conversions always require an explicit cast.

Expressions are evaluated in one of the canonical types (int, long, float or double). That means if your expression is assigned to a non-canonical type (byte, short, or char), an identity or a narrowing conversion will be required. If a narrowing conversion is required, you must write a cast. The cast tells the compiler, "OK, I am aware that the most significant digits are being lost here. Just go ahead and do it." Now it should be clear why we have to use a cast in:

```
byte loNibble = (byte) (byteMe & 0x0F);
```

Each operand in the expression "byteMe & 0xOF" is promoted to int, then the "and" operation is done yielding a 32-bit int result. The variable receiving the expression is an 8-bit byte, so a narrowing conversion is required. Hence, the cast "(byte)" is applied to the entire right-hand side result to mollify the compiler.

What Happens on Overflow?

When a result is too big for the type intended to hold it because of a cast, an implicit type conversion, or the evaluation of an expression, something has to give! What happens depends on whether the result type is integer or floating point.

Integer overflow

When an integer-valued expression is too big for its type, only the low end (least significant) bits get stored. Because of the way two's-complement numbers are stored, adding one to the highest positive integer value gives a result of the highest negative integer value. Watch out for this (it's true for all languages that use standard arithmetic, not just Java).

There is only one case in which integer calculation ceases and overflow is reported to the programmer: division by zero (using / or %) will throw an exception. To "throw an exception" is covered in Chapter 10. It means the flow of control is changed to go to an error-handler that you provide.

There is a class called Integer that is part of the standard Java libraries. It contains some useful constants relating to the primitive type int.

```
public static final int   MIN_VALUE = 0x80000000; // class Integer

public static final int   MAX_VALUE = 0x7fffffff; // class Integer
```

There are similar values in the related class Long. Notice how these constants (final) are also static. If something is constant, you surely don't need a copy of it in every object. You can use just one copy for the whole class, so make it static.

One possible use for these constants would be to evaluate an expression at long precision, and then compare the result to these int endpoints. If it is between them, then you know the result can be cast into an int without losing bits, unless it overflowed long, of course.

Floating-point overflow

When a floating-point expression (double or float) overflows, the result becomes infinity. When it underflows (reaches a value that is too small to represent), the result goes to zero. When you do something undefined like divide zero by zero, you get a NaN. Under no circumstances is an exception ever raised from a floating-point expression.

Some Light Relief—Furby's Brain Transplant

The smash hit toy of Christmas 1998 was Furby. Furby was designed by Californian toy inventor Dave Hampton. Part of Dave's motivation to develop Furby was a reaction against the previous "virtual pet," the Japanese Tamagotchi. Tamagotchis are bland, inert lumps of plastic. Dave knew he could do better, and he created a fur-covered toy that sings, farts, and wobbles—sometimes all at once. No kid could resist that. In 1998, Furby sold over two million units, and they were actually being rushed by air from mainland China factories to satisfy U.S. market demand! Toys have a limited life. Furby was red hot in 1998, warm in 1999, and by 2000 you could buy one for $9.99 in Target.

Furby is an animated doll about 7 inches high. It looks like a Gremlin from the 1984 movie of that name, and toy distributor Hasbro had to settle an infringement claim from Warner Brothers. Inside, Furby is packed with a rich assortment of devices. It has a microphone, a loudspeaker, infrared transmitter/receiver, light detector, speech generation chip, CPU, EEPROM, and RAM. It has a motor and various cams to animate ears, eyes, body, etc. Obviously, it was imperative to take one of these apart and reprogram it for more useful tasks.

Unfortunately, at some point in the past, Furby architect Dave Hampton had been rudely surprised by "potty-mouth Barney," and he was determined that no one would pull the same stunt with Furby. There was another reason to make it difficult to reverse-engineer Furby: to prevent other toy companies from copying the technology. The world of children's toys is (apparently) a cutthroat, rapacious, dog-eat-dog world, where intellectual property is Napstered on a daily basis, and only the strong survive. The main defense against reverse-engineering Furby was a brittle epoxy shell that completely encased Furby's CPU, ROM, RAM, audio data, and the I/O interfaces such as driver transistors and an analog-digital convertor.

The epoxy made it impossible to clamp a logic analyzer onto the CPU, read the bus traffic, and dump out the control program. It is impossible to chip or grind off the epoxy without destroying the components underneath. Certain U.S. government labs are equipped with the right acids and neutralizers to break into equipment like this. But I don't know anyone at the CIA or NSA, and this didn't seem like the right project to introduce myself. Furby's software and sound data is not accessible for reading, writing, disassembly, replacement, or even examination. There are no exposed data/address buses, interrupt lines, or I/O lines other than those that directly drive the peripherals. Conclusion: Reprogramming Furby would require junking the existing CPU and fitting another CPU and memory—effectively, a Furby brain transplant.

So I issued the "Hack Furby" challenge from my website at afu.com (where I also keep the Java Programmers FAQ). The Hack Furby challenge offered a cash prize for the first person to reverse-engineer Furby or retrofit it with a different CPU. It was similar to challenges issued in the early days of aviation. Almost all the early aviation milestones—Bleriot's flight across the channel, Alcock and Brown across the Atlantic, Lindbergh's solo Atlantic

Chapter 8. More OOP—Extending Classes

We now come to the main part of object-oriented programming, covering inheritance and polymorphism. Despite the unusual names, they describe some clean concepts. You need a solid understanding of inheritance and polymorphism to program in an object-oriented language and to use some of the Java library routines.

As we have seen several times, inheritance means basing a new class on a class that is already defined. The new class will extend the existing class in some way. Just as inheritance in real life is "what you get from a parent," inheritance in OOP is "what you get from a parent class." Every class has one immediate parent class. This is either a parent that you name explicitly or the parent that you get implicitly. The implicit parent you get if you don't name one explicitly is java.lang.Object.

```
class A { /*code*/ }
```

is equivalent to:

```
class A extends java.lang.Object { /*code*/ }
```

The class java.lang.Object is the ultimate parent class of all classes. The phrase "class A extends B" is how you indicate that A is a child of class B. A child class is also known as a subclass or subtype. A subclass can access all the

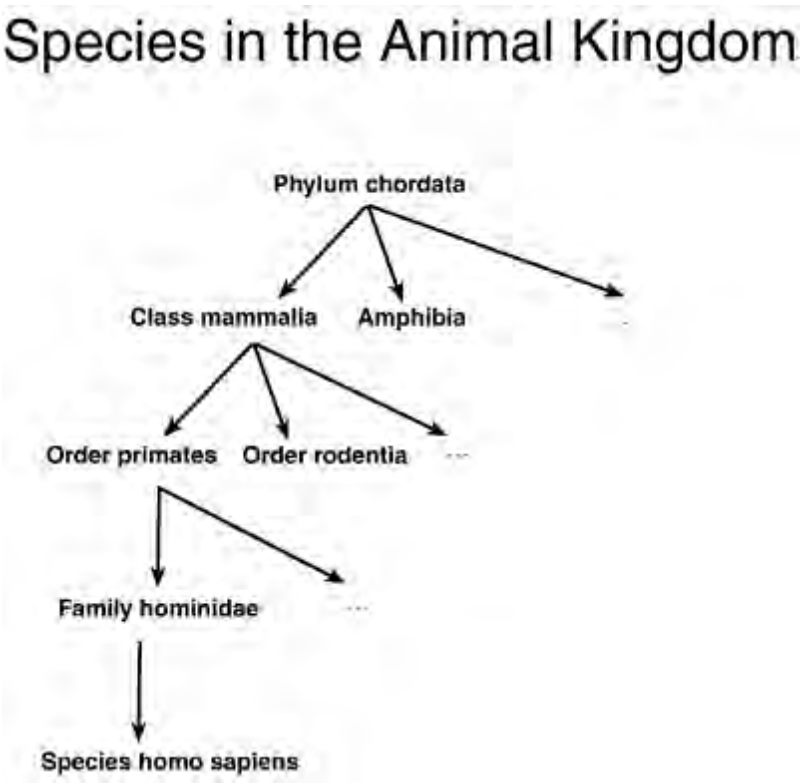
Inheritance

In this section we are going to look at a real-world example of type inheritance, and turn that into code fragments to give you a solid understanding of the big picture. Then we'll look at a more substantial example of inheritance in OOP, and present that in terms of a complete program you can type in and run.

Real-world example of inheritance

To see what inheritance means in practice, consider a real-world example of the Linnaean taxonomy of the animal kingdom, as shown in Figure 8-1.

Figure 8-1. A real-world example of an inheritance hierarchy



Animals are classified by whether they have a spinal cord or not. Developing the example with code, we have:

```
class Animal {   }

class Chordata extends Animal { final boolean spine=true; }
```

Polymorphism

Polymorphism is a complicated name for a straightfoward concept. It is Greek for "many shapes," and it merely means using the same one name to refer to different methods. "Name reuse" would be a better term. There are two types of polymorphism in Java: the really easy kind (overloading) and the interesting kind (overriding).

Overloading

The really easy kind of polymorphism is called overloading in Java and other languages, and it means that in any class you can use the same name for several different (but hopefully related) methods. The methods must have different signatures, however, so that the compiler can tell which of the synonyms is intended.

Here are two overloaded methods:

```
public static int parseInt(String s) throws NumberFormatException

public static int parseInt(String s, int radix)

                                  throws NumberFormatException
```

These methods come from the class java.lang.Integer class wrapper for the primitive type int. The first method tries to interpret the String as an int, and return the corresponding binary int value. The second method does the same thing, but uses an arbitrary base that you specify. You could parse a hexadecimal String by supplying 16 as the radix argument.

The return type and the exceptions that a method can throw are not looked at when resolving same-named functions in Java.

The I/O facilities of a language are one typical place where overloading is used. You don't want to have an I/O class that requires a different method name depending on whether you are trying to print a short, an int, a long, and so on. You just want to be able to say "print(thing)."

Overriding

The second, more complicated kind of polymorphism, true polymorphism, is resolved dynamically at run-time. It occurs when a subclass class has a method with the same signature (number, type, and order of parameters) as a method in the superclass. When this happens, the method in the derived class overrides the method in the superclass. Methods cannot be overridden to be more private only to be more public.

The Class Whose Name Is Class

Here's where the terminology admittedly can get a little confusing. We saw a few pages back that every object has some Run Time Type Information associated with it. The RTTI for any object is stored in an object whose class is called "Class." Class could certainly use a better, less self-referential name, like ClassInformation or just RTTI.

A big use of Class is to help with loading new classes during program execution. A lot of other languages load an entire program before they start execution. Java encourages dynamic loading. Classes only need to be loaded on demand when they are first referenced.

When you compile a class called, say, Fruit, the compiler creates the bytecode file Fruit.class. When the first reference to a Fruit type or object is encountered at run-time, the JVM asks the java.lang.ClassLoader class to find and load that class file. Typically, the ClassLoader transforms the fully qualified name into a file name and looks for that in certain places locally (and for applets, it looks across the net back to the server). It will look in the directories on the class path; it will look in the lib\ext directory in the run-time library; it will look at any jar files you have named, and so on. If the class file is not found, an error is reported. Otherwise the JVM loads the bytecode instructions into memory and automatically creates a Class object to represent the class.

Class provides the information for checking casts at run-time, and it also lets a program introspect all the members of a class—the data fields and the methods. These are systems programming activities, unlikely to occur in your programs, but providing Class makes it easy to program them without stepping outside the Java system. You can safely skim over the rest of this section and return when you have a specific need to look at RTTI.

To access the run-time type information for a class, you need an object of type Class. You can get one in three ways. Here's some code showing the alternatives:

```
Object o = new Timestamp();
```

```
Class which = o.getClass(); // getClass is a method in Object
```

or

```
Class which = Class.forName("Timestamp"); // forName is a static method
```

Exercises

1. What are the four attributes that distinguish object-oriented programming? What are some advantages and disadvantages of OOP?
2. Give three examples of primitive types and three examples of predefined Java classes (i.e., object types).
3. What is the default constructor, and when is it called? What is a no-arg constructor?
4. Describe overriding, and write some code to show an example.
5. Consider the following three related classes:

```
class Mammal {}
class Dog extends Mammal { }
class Cat extends Mammal { }
```

There are these variables of each class:

```
Mammal m;
Dog d = new Dog( );
Cat c = new Cat( );
```

Which of these statements will cause an error at compile time, and why? Which of these statements may cause an error at run-time, and why?

Some Light Relief—The Nerd Detection System

Most people are familiar with the little security decals that electronic and other high-value stores use to deter shoplifters. The sticker contains a metallic strip. Unless deactivated by a store cashier, the sticker sets off an alarm when carried past a detector at the store doors.

These security stickers are actually a form of antenna. The sticker detector sends out a weak RF signal between two posts through which shoppers will pass. It looks for a return signal at a specific frequency, which indicates that one of the stickers has entered the field between the posts.

All this theory was obvious to a couple of California Institute of Technology students Dwight Berg and Tom Capellari, who decided to test the system in practice. Naturally, they selected a freshman to (unknowingly) participate in the trials. At preregistration, after the unlucky frosh's picture was taken but before it was laminated onto his I.D. card, Dwight and Tom fixed a couple of active security decals from local stores onto the back of the photo.

The stunt card was then laminated together, hiding the gimmick, and the two conspirators awaited further developments. A couple of months later they caught up with their victim as he was entering one of the stores. He was carrying his wallet above his head. In response to a comment that this was an unusual posture, the frosh replied that something in his wallet, probably his bank card, seemed to set off store alarms. He had been conditioned to carry his wallet above his head after several weeks of setting off the alarms while entering and leaving many of the local stores.

The frosh seemed unimpressed with Dwight and Tom's suggestion that perhaps the local merchants had installed some new type of nerd detection system. Apparently though the comment got the frosh thinking, because on the next occasion when he met Dwight he put him in a headlock until he confessed to his misdeed. Moral: Never annoy a computer programmer.

Chapter 9. Arrays

In this chapter, we introduce arrays and describe how to use them. Arrays in Java have pretty much the same features as arrays in many languages—the ability to store multiple variables all of one type, and access them by an index value.

Understanding and Creating Arrays

There are some neat features that flow from Java's rule that arrays are objects. That's a good place to start reviewing arrays.

Arrays are objects

When Java says arrays are objects, it means array types are reference types, and your array variable is really a reference to an array. What looks like the declaration of an array:

```
int day[];
```

is actually a variable that will point to an array-of-ints when you create that array. Notice that the array size is not mentioned in the declaration. When we finally fill in the pointer, it can point to any size of int array, and in the course of execution you can assign it different values to point to different arrays. You can't suddenly make an int array point to a char array, of course.

Here are some ways in which arrays are like objects:

- They are objects because the language specification says so ("An object is a class instance or an array", section 4.3.1).
- Array types are reference types, just like object types.
- Arrays are allocated with the "new" operator, similar to constructors.
- Arrays are always allocated in the heap part of memory, never in the stack part of memory. Objects follow the same rule.
- The parent class of all arrays is Object, and you can call any of the methods of Object, such as toString(), on an array.

On the other hand, here are some ways arrays are not like objects:

-

Arrays of Arrays

We'll discuss how the terminology of arrays is not universally consistent among all programming languages. Java uses the terminology consistently, though. Then we'll show how to declare arrays of arrays in Java.

Array terminology

The language specification says there are no multidimensional arrays in Java, meaning the language doesn't use the convention of Pascal or Ada to put several indexes into one set of subscript brackets. Ada allows multidimensional arrays like this:

```
year : array(1..12, 1..31) of real;          Ada code for
                                  multidimensional array.

year(i,j) = 924.4;
```

Ada also allows arrays of arrays, like this:

```
type month is array(1..31) of real;          Ada code for array of
                                  arrays.

year : array(1..12) of month;
year(i)(j) = 924.4;
```

Perhaps Ada and Pascal are a b it obscure these days, but I wanted to show you the different terminology and different design choices these languages have made, compared with Java. In some cases the terminology in one language is in conflict with the terminology in another language. Make sure everyone has a consistent view of the terminology before you start designing your software.

What "multidimensional" means in different languages

The Java language only has arrays of arrays, and it only calls these arrays of arrays.

- The Visual Basic language only has multidimensional arrays, and only calls them multidimensional arrays.
-

Have Array Brackets, Will Travel

There is a quirk of syntax in that the array declaration bracket pairs can "float" to be next to the element type, to be next to the data name, or to be in a mixture of the two. The following are all valid array declarations:

```
int a [] ;

int [] b = { 5, 2, 3 } ;


char c [][] = new char[12][31];

char[] d [] = { {1,1,1,1}, {2,2,2,2} }; // creates d[2][4]

char[][] e;


byte f [][][] = new byte [3][3][7];

byte [][] g[] = new byte [3][3][7];


short [] h, i[], j, k[][];
```

If array brackets appear next to the type, they are part of the type, and apply to every variable in that declaration. In the code above, "j" is an array of short, and "i" is an array of arrays of short. If the array brackets are next to the variable name, they apply only to that variable name.

Allowing (encouraging, actually) array brackets next to the type name is done so declarations of functions returning arrays can be read more easily. Here is an example of how returning an array value from a function would look following C rules. (You can't return an array in C, but this is how C syntax would express it if you could.)

int funarray()[] { ... } Pseudo-C code

Here are the alternatives for expressing it in Java (and it is permissible in Java), first following the C paradigm:

int ginger ()[] { return new int[20]; } Java code

A much better way is to express it as shown in Figure 9-1.

The Math Package

Let's introduce another of the standard classes. This one is called java.lang.Math and it has a couple of dozen useful mathematical functions and constants, including trig routines (watch out—these expect an argument in radians, not degrees), pseudorandom numbers, square root, rounding, and the constants pi and e.

There are two methods in Math to convert between degrees and radians:

```
public static double toDegrees(double); // new in JDK 1.2

public static double toRadians(double); // new in JDK 1.2
```

You'll need these when you call the trig functions if your measurements are in degrees.

You can review the source of the Math package at $JAVAHOME/src/java/lang/Math.java and in the browser looking at the Java API.

How to invoke a Math.method

All the routines in the Math package are static, so you can invoke them using the name of the class, like this:

```
double probability = java.lang.Math.random(); // value 0.0..1.0
```

All the classes in java.lang are automatically imported, so you can write this as:

Some Light Relief—Think Big (and Small)

One early Java technology was applets—a Java program that runs in a browser. Someone who sets up a website can serve up executable Java programs referenced from HTML pages. When a user browses such a page, the Java code is downloaded to his or her system, along with the HTML text and images. A JVM inside the browser (safely and securely) executes the applet on the user's system.

You write an applet by extending the class javax.swing.JApplet, supplying your own child versions of some of the methods in the base class. At one time applets were very popular on the Internet, and they are still widely used for some applications on company intranets. They are also widely used at some educational sites.

Applets were wounded in the "browser wars" of the late 1990s. Microsoft crushed all competition in web browsers by bundling the Internet Explorer application, and paying off large ISPs to use Explorer. Eventually Netscape went out of business, and then people had to download complicated browser plug-ins to use applets. Many decided it was more trouble than it was worth. Pulling this kind of stunt is illegal when you're a monopoly. Microsoft settled out of court by paying $750 million to Netscape's owner AOL. But Netscape was still out of business, and there was still no competition in the browser market, and there was still no native support for applets.

People write applets to do all kinds of things, usually focused on a graphical and/or dynamic presentation of data from the server. There's one particularly good applet which you can easily find on the web. Known as the "Powers of 10" demo, it was written by Matthew J. Parry-Hill, Christopher A. Burdett, and Michael Davidson of the National High Magnetic Field Laboratory at Florida State University. The applet is based on the book "Cosmic View: The Universe in 40 Jumps" written by brilliant Dutch engineer and educator, Kees Boeke. Figure 9-2 shows a screen shot of the Powers of 10 applet.

Figure 9-2. Powers of 10 interactive Java-based tutorial at
http://micro.magnet.fsu.edu/primer/java/scienceopticsu/powersof10/index.html

Chapter 10. Exceptions

When people say "it's the exception that proves the rule" they mean prove in the sense of "test", as in proving grounds. They also mean exception in the sense of "infringement". In other words, you're really hearing "it's the infringements that test whether the rule is any good or not", but never mind that. In Java, the word "exception" relates to an unwanted error condition, and how we recover from it.

In this chapter, we'll cover the purpose of exceptions, what kinds of things cause them, and how to handle them when they happen. We'll start with a look at two data structures used by the run-time system, because some knowledge of these will give you much better insight into exceptions. We'll end the chapter with a description of the assert statement added to Java in JDK 1.4.

Exceptions and asserts are two ways of dealing with unexpected and unwanted error situations. Exceptions give you a chance to recover from the error and continue program execution. Assert statements provide a way to stop a program dead when it notices some serious inconsistency (like a checksum being incorrect). You can configure at run-time whether you want this "stop on error" behavior or not.

Run-time Internals: The Heap

We made the point in Chapter 2 that the purpose of reference variables is to allow automatic memory management. Many languages allow storage to be allocated dynamically (at run-time). This is good, because it allows a program to adapt to the size of the data it is trying to process, rather than being compiled with a fixed upper limit.

Memory is allocated implicitly by creating an object

In low-level languages like C, the programmer asks for a block of memory with a call to one of the malloc routines. In more structured languages like Java, you get memory by instantiating an object. That object can be an array of arbitrary size, limited only by the maximum value that an int index can hold, or by process size or by the virtual storage on the system.

Memory is a scarce resource, and you should free it up (return it to the system) when it is no longer needed. This can be done by the programmer explicitly managing memory resources, or by the system keeping track of what is in use and freeing up memory that is no longer in use. When the system has responsibility for freeing unused memory the process is known as garbage collection, although garbage recycling would be a more accurate term.

Since its first incarnation as the Oak language, Java has included garbage collection. Automatic garbage collection takes the burden of a challenging and error-prone task away from the programmer, leading to better memory utilization and higher reliability software.

A heap of memory

The heap is the run-time data structure used to support dynamic memory management. It is a large storage area that can allocate memory when the run-time library requests it, and can accept previously allocated memory back for deallocation and returning to the pool. Why is it called a heap? In English, a heap is a quantity of things lying in any old order on top of one another. That's what a heap is in a run-time library. It starts as a large segment of memory given to the process when it starts. As the process runs, it allocates memory, gives it back, allocates some more, and pretty soon you have a heap. The heap itself keeps track of what is in use and what is free. In C, you manage the heap manually. In Java, it is done for you.

Your programs start with a default total heap size, and you can change that maximum using a compiler flag like "-Xmx64m" for the JDK ("64m" means 64 MB). There are several other compiler flags to give guidance to the heap manager. You would only use them if you are tuning an application to make it fit on hardware that may be slightly too small. Search at java.sun.com for the document "Garbage Collector Ergonomics" if you want more information on these flags.

Memory leaks—rare in Java

Garbage Collection

Many programming languages support heap-based memory allocation. All objects in Java are allocated on the heap; no objects are ever allocated on the stack (an additional special-purpose storage data structure). Different pieces of storage can be allocated from and returned to the heap in no particular order, leading to problems of fragmentation—you have enough total storage free to satisfy a large allocation request, but it is not in the usable form of one contiguous chunk.

Solving the fragmentation problem

Heap fragmentation is resolved by the run-time system periodically reorganizing the heap, and possibly relocating some live (in-use) objects. All this will be transparent to the application, apart from the time it takes. Most algorithms require the application to stop while the heap is being reorganized.

Languages with dynamic data structures (structures that can grow and shrink in size at run-time) must have some way of telling the underlying run-time when they need more memory. C does this with the malloc() library call. Java does this with the "new" operator.

Conversely, you also need some way to indicate memory that is no longer in use (e.g., threads that have terminated, objects that are no longer referenced by anything, variables that have gone out of scope, etc.) and hand it back to the run-time system for reuse. C and C++ require explicit deallocation of memory; C does this with the free() library call, C++ uses delete(). The programmer has to say what memory (objects) to give back to the run-time system, and when. In practice, this has turned out to be an error-prone task. It's all too easy to create a "memory leak" by not freeing memory before overwriting the last pointer to it. It can then neither be referenced nor freed, and is lost to further use for as long as the program runs. The process address space grows bigger and bigger, and the process gets slower and slower as it is swapped out to make room for other tasks. Java takes a different approach to reclaiming memory.

To avoid the problems of explicit memory management, Java takes the burden off the shoulders of the programmer and puts it on the run-time storage manager. One subsystem of the storage manager is the "garbage collector." The automatic reclaiming of memory that is no longer in use is known as "garbage collection" in computer science. Java has a thread that runs in the background whose task is to do garbage collection. It looks at memory, and when it finds objects that are no longer referenced, it reclaims them by telling the heap that memory is available to be reallocated.

The costs and benefits of garbage collection

Taking away the task of memory management from the programmer gives him or her one less thing to worry about, and makes the resulting software much more reliable in use. It may take a little longer to run compared with a language like C++ with explicit memory management, because the garbage collector has to go out and look for reclaimable memory rather than simply being told where to find it. On the other hand, it's much quicker to debug your programs and get them running in the first place. Most people would agree that in the presence of ever-improving hardware performance, a small performance overhead is an acceptable price to pay for more reliable software.

What is the cost of making garbage collection an implicit operation of the run-time system rather than a responsibility of the programmer? It means that at unpredictable times, a potentially large amount of behind-the-scenes processing will suddenly start up when some low-water mark is hit and more memory is called for. This has been a problem with past

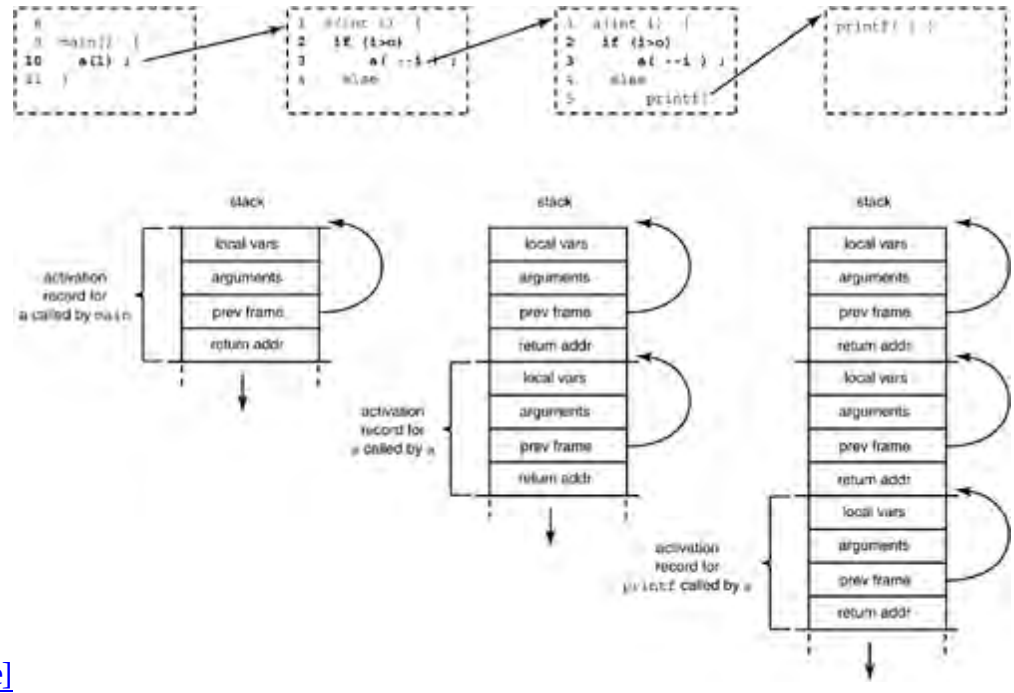
Run-time Internals: The Stack

The stack is a run-time data structure that provides the storage space needed for method calls and returns. All modern block-structured languages use stacks, and many processors have some hardware support for them. Unfortunately in recent years, the terminology has got a little looser; marketing people started using stack to mean any layered system service, such as the TCP/IP library, or an application server and container. In this chapter, stack means the LIFO (Last In First Out) data structure that provides space for local variables, and implements the method-calling conventions.

When you call a method, some housekeeping data, known as an activation record or stack frame, is pushed onto the stack. The activation record contains the return address, the arguments passed to the function, the space for local variables, intermediate calculations, and so on, for that invocation of that method in that thread of control. When you return from a function, the activation record is popped from the stack. The next function call will push another record into the same space.

Figure 10-1 shows how each method call results in a new activation record being pushed onto (added to) the stack. Stacks are only ever meant to be used for data, never for storing code.

Figure 10-1. Stacks in Java



[View full size image]

Stack cracking—impossible in Java

If you have any pointers back into the old activation record on the stack, memory can be corrupted, as

Exceptions

We'll cover the purpose and use of exceptions following this order. The one sentence summary is "Exceptions are like software interrupts—they are generated by error conditions like division by zero, and they divert the flow of control to a place where you have said you will handle this kind of error."

First, we'll look at the basics of:

- Why exceptions are in the language.
- What causes an exception (implicitly and explicitly).

Once an exception has occurred, you'll want to know how to take steps to deal with it:

- How to handle ("catch") an exception within the method where it was thrown.
- Handling groups of related exceptions.

You'll also need to know what happens if you do not provide code to handle each type of exception. You can skip this on a first reading if you just want the exception basics. These sections have information saying how methods tell the compiler about exceptions they might generate but do not handle:

- How the exception propagates if not handled in the method where it was thrown.
- How and why methods declare the exceptions that can propagate out of them.
- Fancy exception stuff.

The purpose of exceptions

Exceptions are for changing the flow of control when some important or unexpected event, usually an error, has occurred. They divert processing to a part of the program that can try to cope with the error, or at least die gracefully.

There are many possible errors that can happen in non-trivial programs: these range from "unable to open a file" to "array subscript out of range" to "no memory left to allocate" to "division by zero." It would obscure the code greatly to check for all possibilities at all places where they may happen. Exceptions provide a clean way for you to write general

The Assert Statement

Introduced with Java 1.4, the assert statement helps to debug code and also troubleshoot applications after they have been deployed. You write an assert statement stating some very important condition that you believe will always be true at this point in your program. If the condition is not true, the assertion throws an Error (a Throwable thing that is not intended to be caught). You can do that already in Java. The part that is new is that assertion statements let you choose at run-time whether the assertion checking is on or off.

Two key steps in using assert

There are two key pieces to using asserts.

- First, you sprinkle assert statements at a few critical places in your program. For example, after calculating a checksum, you could assert that the calculated value equals the stored checksum. You only use assert statements for fatal errors—something has gone so wrong that the only thing to do is stop before more data disappears or whatever.
- The second half of assert statements is that you control whether the assert statements are in effect, or not, at run-time. There is a command line option to the JVM that enables or disables whether the assertions are executed, and this can be applied to individual packages and even classes.

The usual scenario is that you keep the assert statements on during debugging and testing. After testing, when you are confident that the application works correctly, you no longer need to do all that checking. So you disable the assert statements, but leave them compiled in the source code. If the application later hits a problem, you can enable asserts and rerun the failure case to see if any assertions are untrue. This can be done in the field or over telephone support.

An assert statement looks like either of these alternatives:

```
assert booleanExpression;
```

```
assert booleanExpression : Expression2;
```

If the boolean is not true, a java.lang.AssertionError is thrown. The second form of the statement allows you to write an expression that will be passed to the constructor of the AssertionError. The expression is supposed to resolve to a message about what went wrong. You should just pass a String. An expression of other types (such as int) is allowed to accommodate those crazies who want to number their error messages or label them with a character instead of using self-identifying strings.

Further Reading

There is more information on the assert statement, including a couple of clever idioms, at the Sun site java.sun.com/j2se/1.5/docs/guide/lang/assert.html. One of the idioms is code to prevent a program from running at all if assertions are turned off.

Some Light Relief—Making an Exception for You

In June 2003, NASA launched two robot geologist missions to Mars. After a space voyage of six months, the Spirit Rover was the first to arrive safely and start its exploration, looking for signs of water and other geological features. For two and a half weeks on the surface, Spirit sent a series of stunning pictures. Suddenly and inexplicably, it dropped contact with mission control.

The team back at California's Jet Propulsion Lab who designed and built the rovers were mystified and despondent. The Spirit mission manager, Jennifer Trosper, talked to her husband on the phone. "I asked him first how his day was. He said it was okay. And then he asked me how my day was going," recalled Trosper. "Well I think I'm personally responsible for the loss of a $400 million national asset," she confessed. I hate those kind of "rainy Monday" days, don't you?

The Rover is a reprogrammable embedded system that is directed by remote control instructions over wireless from Earth. If Spirit didn't make contact, it wouldn't get instructions for more tasks, or even to diagnose the failure. Things perked up a little the next day when JPL sent a command, and Spirit acknowledged it before once again falling silent.

The Spirit rover is built from off-the-shelf hardware and software. Wind River's VxWorks real-time embedded OS runs on top of a radiation-hardened RAD6000 CPU chip from Lockheed-Martin. This chip is based on the same Power-PC CPU that IBM uses in its RS/6000 Unix workstations. It's a 32-bit RISC processor clocked at 33MHz with 120MB main memory, which is plenty for the tasks it has to do. Mars is a hostile environment for anything mechanical, so the rover has a 256 MB flash memory filesystem instead of disk.

The flash filesystem stores data files (such as photo files), and also executable programs. It's designed so that when the system boots up, some filesystem data is copied from flash into a main memory cache. Obviously the amount of RAM reserved for the cache places an overall limit on the size and number of files the RAM disk can hold. JPL was aware of this limit, and had calculated that regular operations would stay well within it.

Everything about the software was designed to be maintained over radio links from planet Earth. Indeed, the mission team had uploaded a completely new software revision a week after the June 2003 launch, as the Spirit rover raced towards Mars. You never want to delete the old software until you've had a chance to check that the new software works in all circumstances, so the flash memory now held two sets of executables.

The software upload fixed the bugs that had been identified, and mission control made a note that they'd have to delete the old files after the new ones had been shown to work on the ground. Six months later, Spirit landed on Mars, and started collecting data. Each set of data, each image, each instrument reading was stored in a new file in the flash file system. On Martian day 15 of the mission, the ground team uploaded in two parts a utility that would clean out the obsolete files and free up a lot of room in the flash filesystem.

Unhappily, the second half of the upload failed, and was rescheduled for the next communications window between Earth and Mars, four days later. In the meantime, Spirit continued taking pictures and measurements, and storing the results in the rapidly filling filesystem. Before the file delete utility could be uploaded, the portion of RAM dedicated to the flash filesystem filled up completely.

Chapter 11. Interfaces

Interfaces are an important concept in Java. In this chapter, we'll explain what interfaces do, and how you use them. Here's a summary of what interfaces do, so you can see where we're headed.

In a few words, an interface is similar to a class that only abstract methods. You can't instantiate an abstract class; its purpose is to force any subclasses to implement its methods. That allows other classes to rely on those methods being present in the subclasses.

An interface does the same job as a class, but without the requirement that there is a parent/child relationship. Interfaces are reference types, like classes and arrays. Interface types are used as parameters, and an actual class with that behavior will be used as an argument. A class associates itself with an interface by saying it implements an interface, thereby promising that it provides all the methods that the interface specifies.

Let's start by describing the problem that interfaces solve.

What Problem Does an Interface Solve?

We've seen in previous chapters how classes can be related in a hierarchy, with the common behavior higher up the tree. An example is given in Figure 11-1.

Figure 11-1. Where should the refuel() method go?

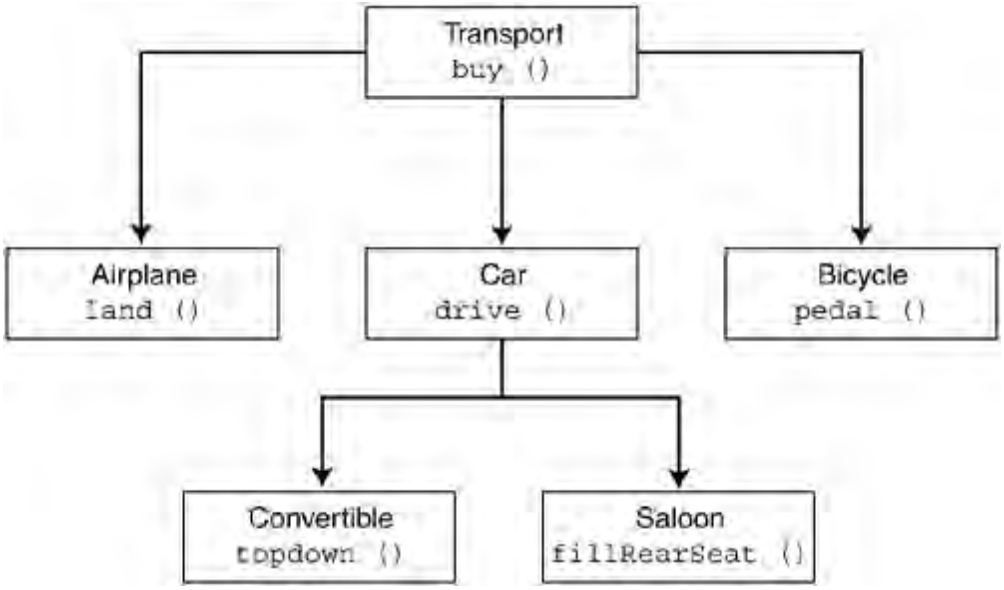


Figure 11-1 shows some classes relating to means of transport. Each box is a class, labeled with the class name, and with an example method that belongs in that class. The parent class is called "Transport" and has a method called buy(). Whatever kind of vehicle you have, you have to buy it before it becomes yours. Putting the method in the parent class means that all subclasses inherit it.

Transport has three child classes. The class called "Car" has a method called drive(), and this is inherited by its two subclasses, Convertible and Saloon. Whatever kind of car it is, the way you move it is to drive(). It belongs in Car, not Transport, because you ride a bicycle or fly an airplane, not drive them.

Now let's imagine we want to keep refuelling information in this class hierarchy. We want to add a method called refuel() that fills the fuel tank of each vehicle. The class that keeps track of our supply depot will call the refuel method on vehicle objects. Where is the right place to add refuel() in the tree of classes?

We cannot add a refuel() method in the transport class, because Bicycle would inherit it, and bicycles are not refuelled. We can add the method individually to Airplane and Car and any other unrelated Classes, like Generator, which represent something you can refuel.

That's good, but it causes problems for our supply depot class. The supply depot class will call an object's refuel() method. What should be the type of the things we pass to it for refuelling? We cannot pass a Transport, because not all transport objects have a refuel() method, and some non-Transport things (like Generator or Pump) need refuelling.

Interface java.lang.Comparable

Interfaces are used frequently in the Java run-time library. Here is the source of interface Comparable in the java.lang package up to JDK 1.4. (We'll show the JDK 1.5 code, which is different, soon).

```
public interface Comparable {

   public int compareTo(Object o);

}
```

All classes that implement Comparable have to provide a compareTo() method. It allows two objects of the same class to be put in order with respect to one another, that is, they can be compared.

If you read the API documentation, you will see that implementations of compareTo() must return a negative, zero, or positive int, depending on whether "this" is smaller, equal to, or greater than the object parameter. There isn't any way for the compiler to enforce this level of meaning, and if you make compareTo() do anything different for one of your classes, this code will be broken anywhere you use that class in place of a Comparable.

Here's a complete example that implements the Comparable interface. To keep it simple, the example is a class that "wraps" an int as an object.

```
class MyInt implements Comparable {

    private int mi;

    public MyInt(int i) { mi = i; }      // constructor

    public int getMyInt() { return mi; } // getter


    public int compareTo(Object other) {

        MyInt miOther = (MyInt) other;   //cast Obj param back to MyInt

        return (this.mi - miOther.mi);

    }

 }
```

Interfaces Versus Abstract Classes

While you use an interface to specify the form that something must have, it does not actually provide the implementation for it. In this sense, an interface is a like an abstract class. The abstract class must be extended in exactly the manner that its abstract methods specify.

An interface differs from an abstract class in the following ways:

- An abstract class is an incomplete class that requires further specialization in child classes. The subclasses are closely related to the parent. An interface doesn't have any overtones of specialization that are present with inheritance. It merely says, "We need something that does 'foo' and here are the ways that users should be able to call it."
- A class can implement several interfaces at once, whereas a class can extend only one parent class.
- Interfaces can be used to support callbacks (inheritance doesn't help with this). This is a significant coding idiom. It essentially provides a pointer to a function, but in a type-safe way. Using Interface Callbacks for GUI Event Handlers on page 239 explains callbacks.

Here's the bottom line: You'll probably use interfaces more often than abstract classes. Use an abstract class when you want to initiate a hierarchy of more specialized classes and provide a partial implementation with fine control over what is private, public, protected, etc. Use an interface when you need multiple inheritance of design to say, "This class has behavior (methods) A, B, and C."

An interface can be extended, but only by another interface. It is legal for a class to implement an interface but only have some of the methods promised in the interface. You must then make it an abstract class that must be further extended (inherited) before it can be instantiated.

Granting Permission Through an Interface—Cloneable

When we looked at the Object class in Chapter 3, we saw that it has this method:

```
protected native Object clone() throws CloneNotSupportedException;
```

Java supports the notion of cloning, meaning to get a complete bit-for-bit duplicate of an object. Cloning, perhaps surprisingly, does not invoke any constructor.

Not every class should support the ability to clone. If I have a class that represents a set of objects that are unique in the real world, such as the ships in a company fleet, the operation of cloning doesn't make sense. You buy new ships, but you can't get exact duplicates in every detail (including, say, registration number) of an existing ship. However, methods (including the clone() method) in the class java.lang.Object are inherited by all classes.

So Java places a further requirement on classes that want to be cloneable: They must implement the cloneable interface to indicate that cloning is valid for them. The cloneable interface is this:

```
public interface Cloneable { } // completely empty
```

The empty interface as a marker is not the most common use of interfaces, but you do see it in the system libraries. To have a class that can be cloned, you must state that the class implements the Cloneable interface.

```
public class ICanBeCopied implements Cloneable { ...
```

Why wasn't the method clone() of Object made part of the Cloneable interface? That seems like the obvious place to put it, instead of in Object. The reason is that we have two conflicting aims.

-
  - We want to provide a default implementation of clone so that any cloneable class can automatically inherit it. A

What Protected Really Means

The "protected" access modifier says the member is visible to any classes in the same package, and to subclasses everywhere. The same package part is clear enough: your code is never in the same package as java.lang.Object.clone(). The "subclasses everywhere" part is more accurately expressed as a protected member can be accessed from within a class through references that are of at least the same type of the class. Consider this example, which uses a protected field called "teeth", which is simpler to follow than the clone() method:

```
package animals;

class Mammal {protected int teeth;  }


package pets;

class Cat extends Mammal { }


class Dog extends Mammal {

    Cat housemate = new Cat();

    int cats_teeth = housemate.teeth; // NO!! field is protected.

}


class Dalmatian extends Dog { }
```

Cat and Dog both inherit the protected field teeth from Mammal. Since Mammal is in a different package, the subclass rule applies. That stipulates that code in the Dog class can access the protected teeth field only through a reference that is Dog or a subtype of Dog, like Dalmatian.

Code in the Dog class cannot access the protected field using a reference to Cat. Cat can access the protected field using a reference to Cat. If you have a reference to Mammal, and if you can successfully cast it to Dog, you can use that Dog value inside Dog to access the field. You can't use the Mammal reference inside Dog to access the field.

This rule is aimed at making sure that protected instance members are accessed only by the part of the class hierarchy they belong to, and not by siblings. It's a funny rule because unlike public, private, or package access, a line of code that accesses a protected member can be valid in one class and not valid in the next class you write in the same package. Even when both classes are subtypes of the class with the protected member! The rule doesn't apply to static protected variables and methods. Those are accessible from any child class, since there is no "object" through which to access them.

Using Interface Callbacks for GUI Event Handlers

So far we've discussed examples of the Comparable and Cloneable interfaces. These interfaces solve compile time issues of forcing a class to have certain behavior. There is an additional way that an interface can be used, to obtain more dynamic behavior. It forms the basis of GUI programming in Java. We have already seen most of this; the only piece that's new is the way that the object-which-implements-an-interface tells the servicing class about itself.

In our earlier example, we called a service method and passed an instance of our handler object as an argument:

```
Depot.service( myPlane );
```

Inside the body of service, the method myPlane.refuel() was immediately called back, and that really did the servicing. The piece that is new for GUI callbacks is this. We don't pass our handler object to the service routine each time a service is needed. Instead, we pass the handler object in a one-time registration with the service. The service saves the pointer to our handler. From then on, whenever an event happens, the service looks at its list of saved references. It will call back to every handler object that has registered with it. You will write the handler object so that it does whatever needs to happen when that part of the GUI is tweaked. If it is a handler for a button marked "quit", the handler will quit the program. If it is a handler for a file open dialog, it will open the file.

This one time registration of the callback target is not such a big change. But it allows the run-time library to take the initiative on deciding when a call back to an event handler must occur, rather than the handler deciding that. Let's look at this in terms of code.

1. The run-time library defines an interface that promises a method called itHappened() like this:

```
 interface ActionListener {

public void itHappened();

}
```

   The run-time library makes calls to the method promised by the interface.Your code implements the interface.

2. In your application code, provide a class that implements the interface:

The Class Double

The following is the declaration of class java.lang.Double. This class implements the Comparable interface, which has been updated to be a generic interface. Apart from the appetizer in this chapter, we defer further explanation of generics until Chapter 15.

```
public final class java.lang.Double extends java.lang.Number
                                    implements java.lang.Comparable {
     // constructors
     public java.lang.Double(double);
     public java.lang.Double(java.lang.String)
                         throws java.lang.NumberFormatException;
     public static final double POSITIVE_INFINITY = 1.0 / 0.0;
     public static final double NEGATIVE_INFINITY = -1.0 / 0.0;
     public static final double NaN = 0.0d / 0.0;
     public static final double MAX_VALUE = 1.79769313486231570e+308;
     public static final double MIN_VALUE = longBitsToDouble(1L);
     public static final java.lang.Class TYPE=
                                    Class.getPrimitiveClass("double");


     public byte byteValue();
     public short shortValue();
     public int intValue();
     public long longValue();
     public float floatValue();
     public double doubleValue();


     public int compareTo(java.lang.Double);
     public int compareTo(java.lang.Object);
     public boolean isInfinite();
```

Exercises

1. Describe, without excessive handwaving, two common uses for interfaces.
2. Given the SupplyDepot, Transport, and CapableOfBeingRefuelled classes/interfaces described earlier in this chapter, add a main routine to Airplane. Then add fields, parameters, and code to SupplyDepot and elsewhere so that a CapableOfBeingRefuelled can tell the service routine how much fuel it takes. Make sure that amount of fuel is decremented at the Depot, and incremented in the object that implements CapableOfBeingRefuelled.
3. Look at the Cloneable interface in javadoc. Take any class that you have written and make it cloneable by making it implement the Cloneable interface.
4. Override Object.clone() to do a shallow copy for your class, and also keep count of the number of objects of your class that have been created. Don't forget to count those created via a constructor, too.
5. Write some code to clone an object of your class. Change your version of clone() to do a deep copy for your class. Run the clone program again, and make it print out enough information that you can tell the difference between a shallow clone and a deep clone.

Some Light Relief—The Java-Powered Toaster

First prize for the "most entertaining Java application of 2001" goes to Robin Southgate. For his final year project as an Industrial Design student at Brunel University in England, Robin designed and built a bread toaster. Not just any toaster though. Robin's toaster is powered by a Java program that dials a weather service, retrieves the forecast, and singes the outlook onto the toast. Examples of the toast are shown in Figure 11-3.

Figure 11-3. Java-powered toaster

Toast on a sunny day

Toast on a rainy day

Robin's design integrates a standard domestic toaster with the Tiny InterNet Interface (TINI) microcontroller from Dallas Semiconductor. The TINI is a $20 microcontroller chip set that supports an incredible software development platform. TINI contains a Java virtual machine, a web server, and a TCP/IP network stack running on top of a real time operating system, all in less than half a Mbyte of flash RAM. You program the chip and control the I/O to its peripherals completely in Java. The peripherals can include ethernet, a parallel port, a wireless network interface, as well as the usual RS232 serial port and I2C bus.

Robin modified the toaster so that it works automatically. When a piece of toast is put in, the microcontroller wakes up and dials out through a modem on the serial port to remotely access the weather information. The Java code then condenses the forecast into a choice between "sunny," "overcast," "rain," or "snow," and chooses the appropriate baffle to move in front of the toast heater element. The baffle is made of polytetrafluorethylene (more commonly known as "teflon" or PTFE) which is both food-safe and heat resistant at toaster temperatures. The baffle has a hole in the shape of the weather icon, exposing that area of the toast to more radiant heat than adjacent masked areas. The toast pops up in about 30 seconds with the weather icon burned onto it.

When he started on the project, Robin appealed to the TINI engineering mailing list for advice. About half the engineers made a lot of fat-headed suggestions, like using cocoa powder for toner to print on the bread. Another proposed using a CO2 laser to reduce cooking time into the 1-2 second range. They just didn't seem to be taking the project seriously. Other engineers could see the value of Java-powered toast, and gave Robin guidance on how he could refine his design.

The project involved sensing toaster operation, communicating with a remote site, decoding the data returned, moving the baffles, and controlling the toaster element. The prototype shown in Figure 11-4 needed special attention to switch the high current for the heating element safely. TINI is an excellent choice for this kind of embedded design. It has a built-in serial port that can trivially drive an external modem. There are readily available modules to monitor current,

Chapter 12. Nested Classes

This chapter covers all aspects of nested classes. In compiler terminology, nesting means "putting something inside another thing". Everyone who has read this far will be comfortable with nesting methods and data fields inside classes. In this chapter we explain the different ways a class can be nested inside another class, and what it all means.

Introduction to Nested Classes

Nested classes are another in the (ever-lengthening) list of "Java features that were introduced to make one specific thing quicker to write". There's a trade-off though. While nested classes make event-handlers simpler to write, they also add complexity to the compiler and to the features that programmers must learn.

Nested classes will make more sense when we have covered event-handlers for the GUI (Chapter 20), so you could skim the rest of this chapter until then. For now, just note that there are ways you can nest classes inside each other like Russian dolls, and postpone an in-depth reading until you need it.

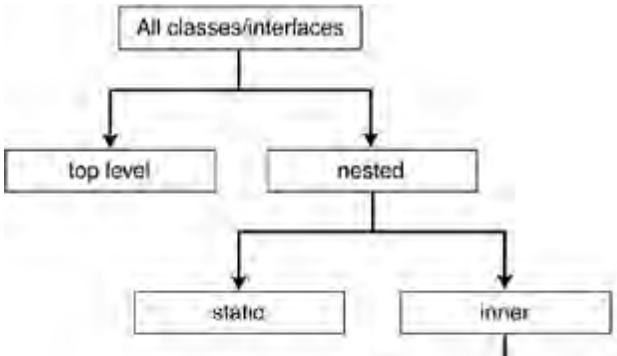
The name "nested class" suggests you just write a class declaration inside a class, and you're done! Actually, there are four different kinds of nested classes specialized for different purposes, and given different names.

All classes are either:

- Top-level or nested
- Nested classes are:
  - Static classes or inner classes
  - Inner classes are:
    - Member classes or local classes or anonymous classes

Figure 12-1 represents the hierarchy and terminology of nested classes in a diagram.

Figure 12-1. The hierarchy of nested classes



Nested Static Classes

Nested static classes are the simplest of the nested classes to write, understand, and use. Table 12-1 shows how to create a nested static class. You simply write the declaration of your class in the usual way, give it the modifier static, and put it inside a top-level class.

Table 12-1. Creating a nested static class

Java term	Description	Example code

Nested static class	`class Top {`
	
	`    static class MyNested {`
	`}`
	
	`}`
		A nested class that is declared "static."
		
		This acts like a top-level class.

Qualities of a nested static class

You recall that static means "there is only one of these". A static thing is not tied to an individual instance object. If you think about it, there is only one of any top-level class (though it might have a large number of instances). A static nested class acts exactly like a top-level class. The only differences are:

- The full name of the nested static class includes the name of the class in which it is nested, e.g. Top.MyNested. Instance declarations of the nested class outside Top would look like:

```
Top.MyNested myObj = new Top.MyNested();
```

- The nested static class has access to all the static methods and static data of the class it is nested in, even the private members.

If it helps, think of the "static" as having more to do with labelling the field of the top-level class, than with the nested

Inner Member Classes

Java supports an instance class being declared within another class, just as an instance method or instance data field is declared within a class. It is called an inner class, and the class is associated with each instance of the class in which it is nested.

The three varieties of inner class are member class, local class, and anonymous class, each one being a refinement of the one before it. We will look at the special features of each of these kinds of classes and how to use them in the pages ahead. Information on member class is shown in Table 12-2.

Table 12-2. Member class

Java term	Description	Example code

Member class	This is an inner class.
	
	It is a nested class that is not declared "static." It is an instance member of a class.
		`class Top {`
		`  class MyMember {`
		`  /* code */`
		`  }`
		`}`

The comment saying /* code */ is a reminder that in practice the classes will have bodies with methods and fields.

A member class is like a nested static class without the keyword static. That says, "The class appears in every instance".

Qualities of an inner member class

Just as any instance member in a class can see all the other members, the scope of an inner class is the entire parent in which it is directly nested.

That is, the inner class can reference any members in its parent. The parent must declare an instance of an inner class before it can invoke the inner class methods, assign to data fields (including private ones), and so on.

The following code shows an example of use.

Inner Local Classes

Inner local classes are perhaps the simplest of the nested classes to write, understand, and use. The following table shows how to create an inner local class. You simply write the declaration of your class in the usual way, and put it inside (local to) a block. The block almost always instantiates an object of the inner class.

Information on the local class is shown in Table 12-3.

Table 12-3. Local class

Java term	Description	Example code

Local class	This is an inner class.
	
	It is local to a block, typically within a method.
		`void myMethod() {`
		`  class MyLocal {`
		`  /*code*/`
		`  void something() { }`
		`  }`
		`  MyLocal m = new MyLocal();`
		`  foo( m );`
		`}`

Qualities of an inner local class

A local class is an inner class. It is a class that is declared within a block of code, so it is not a member of the enclosing class. Typically, a local class is declared within a method.

A local class is not visible outside the block that contains it, but the block can create an object of the inner local class and give away a reference to it. Then the object lives on, and its methods can be invoked by whoever holds the reference.

Where to use an inner local class

Figure 12-2 shows a local class. This inner class is the event handler for a button. The overwhelming use of local classes is to create an object that can be called back as an event handler. The value that local classes add is that you can write the handler close to the code that creates the button (or whatever) that generates the events.

Inner Anonymous Classes

It's possible to go one step further from a local inner class to something called an anonymous class. An anonymous class is a refinement of inner classes, allowing you to write the definition of the class, followed by the instance allocation.

The anonymous class is explained in Table 12-4.

Table 12-4. Anonymous class

Java term	Description	Example code

Anonymous class	This is an inner class.
	
	It is a variation on a local class.
	
	The class is declared and instantiated within a single expression.
		`void foo () {`
		
		`  JFrame jf = new JFrame();`
		`  jf.addEventHandler(`
		`  new EventAdapter() {`
		
		`myOverridingMethod(){`
		`  /* some code */`
		`  } // end method`
		`  } // end anon class`
		`  );`
		`}`

Qualities of an inner anonymous class

Instead of just nesting the class like any other declaration, you go to the

```
new SomeClass ()
```

How Inner Classes Are Compiled

You might be interested to learn that inner classes are completely defined in terms of a source transformation into corresponding freestanding classes, and that is how the Sun compiler compiles them. In other words, first it extracts them and pretends they are top-level classes with funny names. Then it fixes up the extra this pointers that are needed, and compiles them. The following is an inner class called pip:

```
public class orange {
     int i=0;
     void foo() {  }

     class pip {
          int seeds=2;
          void bar() { }
     }
}
```

Here's how the compiler transforms an inner class into JDK 1.0 compatible code. If you understand this process, a lot of the mystery of inner classes will disappear. First, extract the inner class, make it a top-level class, and prefix the containing-class-and-a-dollar-sign to its name.

```
class orange$pip {
     int seeds=2;
     void bar() { }
}
public class orange {
     int i=0;
     void foo() {  }
}
```

The Class Character

The following is the declaration of class java.lang.Character. The important point here is to notice all the routines for classifying a character as an identifier, a digit, etc:

```
public final class java.lang.Character

        implements java.io.Serializable, java.lang.Comparable {

    public java.lang.Character(char); // constructor
```

```
     public static final int MIN_RADIX = 2;

     public static final int MAX_RADIX = 36;

     public static final char MIN_VALUE = '\u0000';

     public static final char MAX_VALUE = '\uffff';

     public static final java.lang.Class TYPE = Class.getPrimitiveClass("char");

     public static final byte UNASSIGNED;//about 20 other Unicode types

     public static final byte UPPERCASE_LETTER;

     public static int getType(char); // returns the Unicode type


     public char charValue();

     public int compareTo(java.lang.Character);

     public int compareTo(java.lang.Object);

     public static int digit(char, int);

     public boolean equals(java.lang.Object);

     public static char forDigit(int, int);

     public static int getNumericValue(char);

     public int hashCode();
```

Exercises

1. Describe the three different kinds of inner class. Find an example of one kind of inner class in the Java run-time library source code. Describe it.
2. What is a static nested class? Give an example of where you might use one.
3. Find an example of a local inner class in the Java run-time library, and explain why it is used there.

Some Light Relief—The Domestic Obfuscated Java Code Non-Competition

Readers of my book Expert C Programming will be aware of the International Obfuscated C Code Competition (IOCCC). It's an annual contest run over Usenet since 1984 to find the most horrible and unreadable C programs of the year. Not horrible in that it is badly written, but in the much subtler concept of being horrible to figure out what it does and how it works.

The IOCCC accepts entries in the winter, which are judged over the spring, and the winners are announced at the summer Usenix conference. It is a great honor to be one of the dozen or so category winners at the IOCCC, as many very good programmers turn their talents to the dark side of the force for this event. If you know C pretty well, you might be interested in figuring out what this IOCCC past winner does:

```
main() {printf(&unix["\021%six\012\0"],(unix)["have"]+"fun"-0x60);}
```

Hint: It doesn't print "have fun."

Here, in the spirit of the IOCCC, are two Java programs that I wrote for April Fool's Day a few years back. You should be pretty good at reading Java code at this point, so I won't spoil your fun.

This program looks like one big comment, so it should compile without problems. When you run it, it greets you! But how?

```
/*   Just Java

     Peter van der Linden

     April 1, 1996.

\u0070\u0075\u0062\u006c\u0069\u0063\u0020

 \u0050\u0076\u0064\u004c\u0020\u0031\u0020\u0041\u0070\u0072\u0039\u0036

  \u002a\u002f\u0020\u0063\u006c\u0061\u0073\u0073\u0020\u0068\u0020\u007b

   \u0020\u0020\u0070\u0075\u0062\u006c\u0069\u0063\u0020\u0020\u0020\u0020

    \u0073\u0074\u0061\u0074\u0069\u0063\u0020\u0020\u0076\u006f\u0069\u0064
```

Part 2: Key Libraries

Chapter 13. Doing Several Things at Once: Threads

Multithreading is not a new concept in software, but it is new to come into the limelight. People have been kicking around experimental implementations for 20 years or more, but it is only in the last few years that desktop hardware (especially desktop multiprocessors) became powerful enough to make multithreading popular.

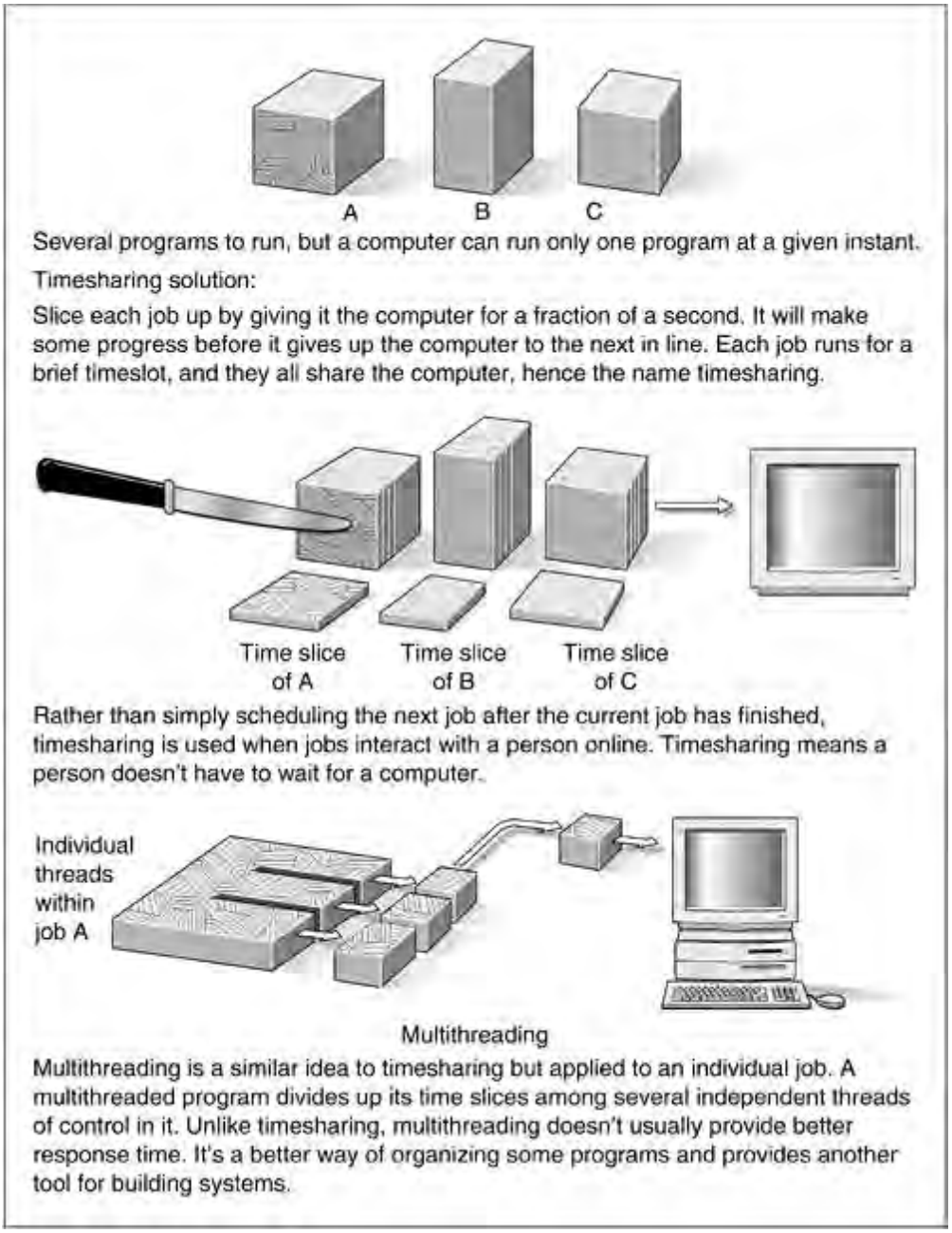
There is a POSIX[1] document 1003.4a (ratified June 1995) that describes a threads API standard. The threads described by the POSIX model and the threads available in Java do not completely coincide. The Java designers didn't use POSIX threads because the POSIX model was still under development when they implemented Java. Java threads are simpler, take care of their own memory management, and do not have the full generality (or overhead) of POSIX threads.

[1] POSIX is an operating system standard heavily weighted to a common subset of Unix.

What Are Threads?

A computer system can give the impression of doing several things simultaneously, like print from one window, compile in another, while downloading music in a third. The OS runs each process for a few milliseconds, then saves the process state, switches to the next process, and so on. Threads simply extend that concept from switching between several different programs to switching between several different functions executing simultaneously within a single program, as shown in Figure 13-1.

Figure 13-1. An explanation of processor time sharing and multithreading



A thread isn't restricted just to one method. Any thread in a multithreaded program can call any series of methods that could be called in a single-threaded program. You often have one thread that waits for input from a GUI and another thread that processes the input when it arrives.

Two Ways to Obtain a New Thread

There are two ways to obtain a new thread of control in Java. Either extend the Thread class, or write a class to implement the java.lang.Runnable interface and use it in the Thread constructor. The first way can be used only if your class doesn't already extend some other class (because Java disallows multiple parents).

1. Extend class java.lang.Thread and override run():

```
class Plum extends Thread {
public void run() { /* more code */ }
}

Plum P = new Plum();
P.start();
```

   or

2. Implement the Runnable interface (the class Thread itself is an implementation of Runnable), and use that class in the Thread constructor:

```
class Mango implements Runnable {
public void run() { /* more code */ }
}

Mango m = new Mango();
```

The Lifecycle of a Thread

We have already covered how a thread is created, and how the start() method inherited from Thread causes execution to start in its run() method. An individual thread dies when execution falls off the end of run() or otherwise leaves the run method (through an exception or return statement).

If an exception is thrown from the run method, the runtime system prints a message saying so, the thread terminates, but the exception does not propagate back into the code that created or started the thread. What this means is that once you start up a separate thread, it doesn't come back to interfere with the code that spawned it.

Priorities

Threads have priorities that you can set and change. A higher priority thread executes ahead of a lower priority thread if they are both ready to run.

Java threads are preemptible, meaning that a running thread will be pushed off the processor by a higher priority thread before it is ready to give it up of its own accord. Java threads might or might not also be time-sliced, meaning that a running thread might share the processor with threads of equal priority.

A slice of time

Not guaranteeing time-slicing might seem a somewhat surprising design decision as it violates the "Principle of Least Astonishment"—it leads to program behavior that programmers find surprising (namely threads can suffer from CPU starvation). There is some precedent in that time-slicing can also be missing in a POSIX-conforming thread implementation. POSIX specifies a number of different scheduling algorithms, one of which (round robin) does do time-slicing. Another scheduling possibility allows a local implementation. In the Solaris case of POSIX threads, only the local implementation is used, and this does not do any time-slicing.

Many people thought that the failure to require time-slicing in the Java scheduler was a mistake that will be fixed in a future release. But it hasn't happened for so many years, that now it probably won't ever.

Since a programmer cannot assume that time-slicing will take place, the careful programmer assures portability by writing threaded code that does not depend on time-slicing. The code must cope with the fact that once a thread starts running, all other threads with the same priority might become blocked. One way to cope with this would be to adjust thread priorities on the fly. That is not recommended because the code will cost you a fortune in software maintenance.

A better way is to yield control to other threads frequently. CPU-intensive threads should call the yield() method at regular intervals to ensure they don't hog the processor. This won't be needed if time-slicing is made a standard part of

Thread Groups

A Thread group is (big surprise!) a group of Threads. A Thread group can contain a set of Threads as well as a set of other Thread groups. It's a way of bundling several related threads together and doing certain housekeeping things to all of them, like starting them all with a single method invocation.

There are methods to create a Thread group and add a Thread to it. You can get a reference to your Thread group for later use:

```
private ThreadGroup mygroup;

mygroup=Thread.currentThread().getThreadGroup();
```

Thread groups exist because it turned out to be a useful concept in the runtime library. There seemed no reason not to just pass it through to application programmers, as well. The designers of the new java.util.concurrency package suggest that thread groups were never much use, and might be deprecated in future. So avoid them.

How many threads?

Sometimes programmers ask, "How many threads should I have in my program?" Ron Winacott of Sun Canada has done a lot of thread programming, and he compares this question to asking, "How many people can I take in my transport?"

The problem is that so much is left unspecified. What kind of transport is it? Is it a motorbike or a jumbo jet? Are the people children, or 250-pound pro wrestlers like Ric Flair? How many are needed to help get to where you want to go (e.g., driver, radio operator, navigator, tail gunner, nanny, observer)? In other words, what kind of program is it, what's the workload, and on what hardware are you running it?

The bottom line is: Each thread has a default stack size of 400K in the JDK current release. It will also use about 0.5K to hold its internal state, but the stack size is the limiting factor. A 32-bit UNIX process (UNIX is the most capable of all the systems to which Java has been ported) effectively has a 2 GB user address space[3] , so in theory, you could have around 5,000 threads. In practice, you would be limited by CPU availability, swap space, and disk bandwidth before you got up there. In one experiment, I was able to create almost 2,000 threads before my desktop system ground to a halt. That was just to create them; I'm not making any claims about them doing any useful work.

[3] The kernel has the other 2 GB of the 4 GB that can be addressed using 32 bits.

Now, back to the real question. Overall, there is no unique correct answer. How many is "reasonable"? There is only one person who can accurately answer this question and that is the programmer writing the threaded application. The runtime library used to have a comment mentioning 26 threads as being the maximum concurrency that one might

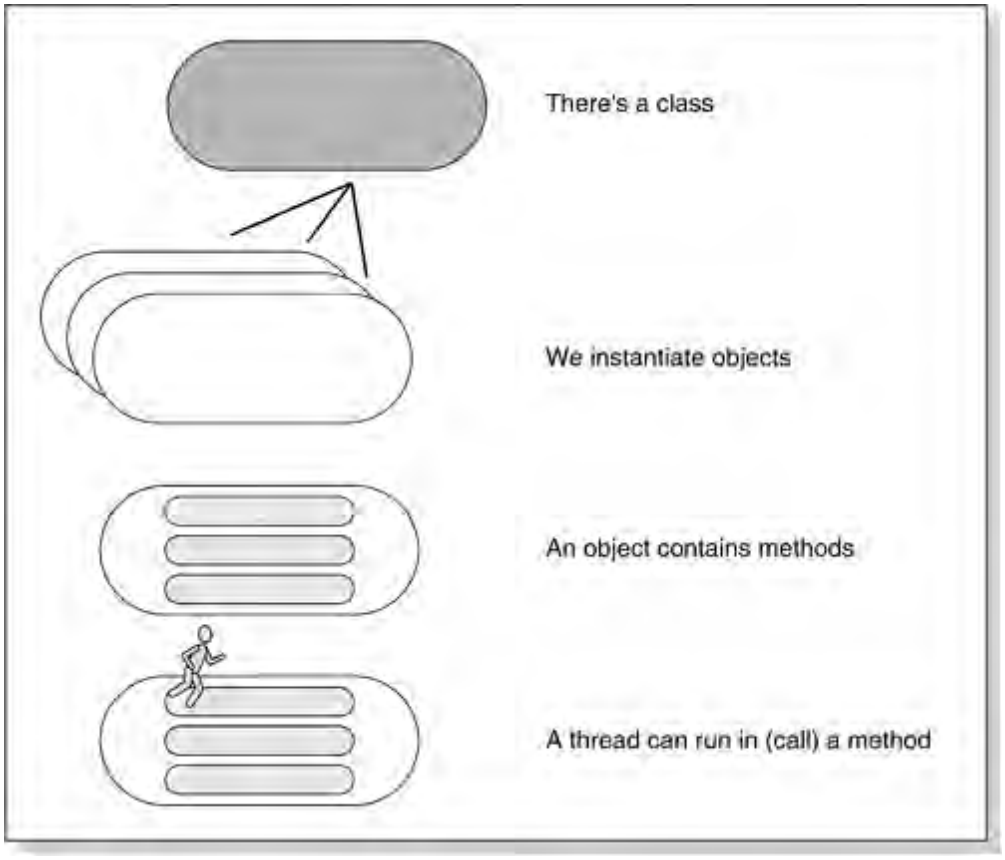
Four Kinds of Threads Programming

Coordination between different threads is known as synchronization. Programs that use threads can be divided into the following four levels of difficulty, depending on the kind of synchronization needed between the different threads.

1. Unrelated threads
2. Related but unsynchronized threads
3. Mutually exclusive threads
4. Communicating mutually exclusive threads

We will deal with the first two here and the second two in the next chapter. Figure 13-3 is the key for all illustrations dealing with this topic.

Figure 13-3. Key to threads diagrams



Unrelated threads

Some Light Relief—The Motion Sensor Solution

A while back I moved into a new office building at Sun Microsystems. This was good in that it gave me an excuse to discard all the flotsam and jetsam I had not yet unpacked from the previous such move. You should move your office every three years for that reason alone. Not me, you. But my move was bad in that there were a few things about the new office that didn't suit me.

Number one on the list was the motion sensor connected to the lights. All modern U.S. office buildings have power-saving features in the lighting, heating, and ventilation. These building services are usually installed in a false floor at the top of a building, but that's another storey[4] . Improving the energy efficiency of buildings is part of the U.S. government's Energy Star program. Energy Star is the reason that the photocopier is always off when you go to use it. There are Energy Star power management guidelines that apply to desktop computers, too. At that time, I was responsible for Sun's desktop power management software in the Solaris kernel, so I know how essential it is to allow users to retain control over power-saving features. Apparently, our building designers did not appreciate this.

[4] Another storey! Geddit?

The light in my new office was wired up to a motion sensor, and it would automatically switch itself off if you didn't move around enough. This was something of a nuisance as there are long periods in the day when the only thing moving in my office are my fingertips flying over the keyboard. Periodically, my office would be plunged into darkness, a sharp transition which spoils my concentration. And concentration is very important to programmers.

After that unpleasant business with the glitter booby trap[5] I installed in the VP's office, it would have been futile to ask the facilities guys to replace the motion sensor with a regular switch. So I considered other alternatives. A gerbil on an exercise wheel? They're high maintenance, noisy, smelly, and they keep erratic hours. One of my colleagues suggested installing folks from the marketing department in my office and having them flap their arms intermittently, thus getting some productive use out of them. I did consider it, but they have many of the same disadvantages as the gerbil, and are not so easy to train.

[5] Booby-trapped ceiling tile, hinged like a trapdoor and piled with confetti on top. The trap is triggered by a thread attached to the back of a desk drawer. It pulls out the safety latch when the drawer is opened. Treat your boss to one today.

The final answer was one of those kitschy dippy bird things. I position it next to the sensor and dip the beak in the glass of water in the morning. It continues to rock backwards and forwards for a good long time, and everyone is happy. I plan to camouflage the dippy bird with a water lily in a dish, which motivates the housekeeping staff to keep the water level topped up. That makes it zero maintenance from my perspective and the closest we'll get to perpetual motion, and furthermore the lights no longer go out on me.

How do dippy birds work?

Dippy birds work on the same general principle as the steam engine, but with less splendor.

Chapter 14. Advanced Thread Topics

In this chapter we will cover the advanced thread topics. Specifically, we will explain how threads synchronize with each other when they have data that they must pass to each other. That is, they cannot solve the problem merely by staying out of each other's way and ignoring each other.

In Chapter 13, we reviewed the easy parts of thread programming as items 1 and 2 in a list there. This chapter covers items 3 and 4 from the same list. These are the hard parts of thread programming. We are about to plunge into level 3: when threads need to exclude each other from running during certain times.

Mutually Exclusive Threads

Here's where threads start to interact with each other, and that makes life a little more complicated. In particular, we have threads that need to work on the same pieces of the same data structure.

These threads must take steps to stay out of each other's way so they don't each simultaneously modify the same piece of data, leaving an uncertain result. Staying out of each other's way is known as mutual exclusion. You'll understand better why mutual exclusion is necessary if we motivate the discussion with some code. You should download or type the example in and run it.

This code simulates a steam boiler. It defines some values (the current reading of, and the safe limit for, a pressure gauge), and then instantiates ten copies of a thread called "pressure," storing them in an array. The pressure class models an input valve that wants to inject more pressure into the boiler. Each pressure object looks to see if we are within safe boiler limits, and if so, increases the pressure. The main routine concludes by waiting for each thread to finish (this is the join() statement) and then prints the current value of the pressure gauge. Here is the main routine:

```
public class p {

     static int pressureGauge=0;

     static final int safetyLimit = 20;


     public static void main(String[]args) {

          pressure []p1 = new pressure[10];

          for (int i=0; i<10; i++)  {

               p1[i] = new pressure();

               p1[i].start();

          }

          // the 10 threads are now running in parallel

          try{

               for (int i=0;i<10;i++)

                         p1[i].join();  // wait for thread to end

          } catch(Exception e){ }


          System.out.println(
```

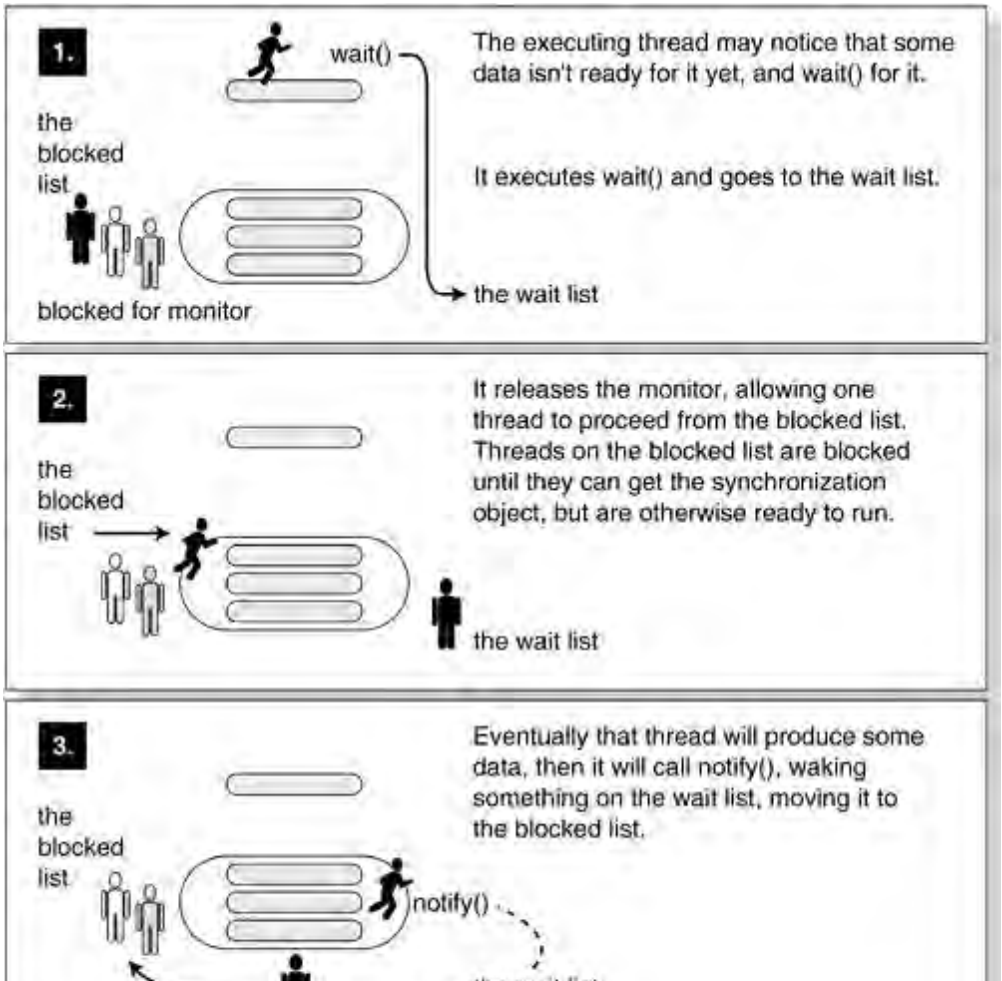
Communicating Mutually Exclusive Threads

Warning: Specialized threads topics ahead

Here's where things become downright complicated until you get familiar with the protocol. This is an advanced section, and you don't need to understand it unless you are tackling sophisticated concurrent programming. You can safely jump over this material to the end of the chapter, returning to study it in depth when you see the words wait/notify in a program.

The hardest kind of programming with threads is when the threads need to pass data back and forth. Imagine that we are in the same situation as the previous section. We have threads that process the same data, so we need to run synchronized. In our new case, however, imagine that it's not enough just to say, "Don't run while I am running." We need the threads to be able to say, "OK, I have some data ready for you," and to suspend themselves if there isn't data ready. There is a convenient parallel programming idiom known as wait/notify that does exactly this. Figure 14-3 shows this in four stages.

Figure 14-3. Communicating mutually exclusive threads



Piped I/O for Threads

A pipe is an easy way to move data between two threads. One thread writes into the pipe, and the other reads from it. This forms a producer/consumer buffer, ready-programmed for you! There are two stream classes that we always use together in a matched consumer/producer pair:

- PipedInputStream – Gets bytes from a pipe (think "hosepipe"; it's just a data structure that squirts bytes at you).
- PipedOutputStream – Puts bytes into a pipe (think "drainpipe"; it's just a data structure that drinks down bytes that you pour into it).

An object of one of these classes is connected to an object of the other class, providing a safe way (without data race conditions) for one thread to send a stream of data to another thread.

As an example of the use of piped streams, the following program reimplements the Producer/Consumer problem, but uses piped streams instead of wait/notify. If you compare, you'll see that this is considerably simpler. There is no visible shared buffer—the pipe stream between the two threads encapsulates it.

The next example shows one thread sending primitive types to another. You can also send Strings using the classes PipedWriter and PipedReader. These four Piped classes are part of the java.io package.

Pipe pitfall

Most operating systems support a pipe feature to communicate between threads or processes. All of these have one common pitfall! The pipe between the threads is implemented by a buffer in shared memory, and this buffer has a limited capacity.

If your input and output happens at different rates, the availability of data will become a bottleneck. The faster thread blocks until the slower thread has either added something to the empty buffer (the output thread is slower), or made some room in an otherwise full buffer by reading something from the buffer (the input thread is slower).

If your system design relies on the two threads taking turns, your program can deadlock and not run satisfactorily. This pipe problem is common to all languages. You can extend these two streams to give the buffer a different size, but that might merely postpone but not solve the pipe problem.

Their use should be clear from the examples here, and it is fully explained in the two chapters on I/O.

Thread Local Storage

Thread local storage is a term for data that is accessible to anything that can access the thread, but that can hold a value that is unique in each thread. A common use for thread local storage is to give a different identifier to each of several newly constructed threads (as in, "if I am thread 5,...") or tells them what they need to work on. ThreadLocal objects are typically private static variables used to associate some state with a thread (e.g., a user ID, session ID, or transaction ID).

Thread local storage was introduced with JDK 1.2 and allows each thread to have its own independently initialized copy of a variable. It's easy enough to give threads their own variables. The bit that's tricky is in getting them initialized with a value that's different for each thread in a thread-safe way. It's tough for the thread to do it because each copy of the same thread is (naturally) executing exactly the same code.

The following describes how you give your threads an int ID. You will create a subclass of ThreadLocal. You extend the ThreadLocal class and add members as needed. The unfamiliar thing in <anglebrackets> is a generic parameter:

```
class MyThreadLocal extends ThreadLocal <Integer>{

    private static int id = 0;


    // this method overrides a method in ThreadLocal

    protected synchronized Integer initialValue() {

        return id++;  // autoboxing at work for you.

    }


    // this method overrides a method in ThreadLocal

    public Integer get() {

        return super.get();

    }


}
```

We'll explain the use of generics in full in the next chapter. For now, it's a way of telling a class "this is the specific type I want you to work on in this subclass". It's a way of getting better type checking than if you just use Object everywhere. Here the <Integer> says we providing an Integer to each thread. We can provide any object as thread local storage.

Package java.util.concurrent

JDK 1.5 introduced several new packages and a couple of dozen new classes to beef up support for threads and thread-based systems. These are centered around package java.util.concurrent. In one sentence, the goal is "to do for threads what java.util.Collection did for data structures". Namely, provide a central, consistent, "bullet-proof" way of expressing best practices, so that programmers no longer need to create them by hand for each new problem.

You can view the concurrent package as a set of design patterns implemented for Java threads (if design patterns are new to you, we mention the basics in the next section). This new package will see the most use in large and sophisticated server systems, and is beyond the scope of this book.

The following is a summary of java.util.concurrency to provide a road map when you are ready to look into it further. The new package contains some utility classes for threads, and some new frameworks.

- There are a small number of new frameworks built around the interface java.util.collections.Executor. It is now easy to use thread pools (which recycle existing threads instead of creating a new thread for each new request). There is support for delayed and periodic task execution. You can have threads that return results, not just fall off the end of the run() method. You can start a long-running thread, query it for completion, and cancel it.
- There are some new Collection data classes that you can use when many threads share access to a common data structure. There is a new Queue class called java.util.concurrent.ConcurrentLinkedQueue, which is suitable for holding the collection for producer-consumer, messaging, parallel tasking, and other threads. There are some other concurrent collections that are thread safe but not bottle-necked by a single lock (a single lock undermines scalability). These containers support multiple readers as well as a tunable number of concurrent writes.
- Improvements in time-related code. There are proper definitions for periods of nanoseconds, microseconds, milliseconds, and seconds. They can easily be converted into one another. There are timeouts for "wait at least this long before giving up".
- Synchronization primitives. There are new classes for common synchronization idioms, including a class that represents a counting semaphore.
- Five new exceptions (e.g. TimeoutException, CancellationException).

The package java.util.concurrent also does some major work on atomicity. That's a performance optimization to support thread safe programs on single variables, without using locks. It's going to take time for all this to sink in, and find its way into the mainstream. It's a very good thing to have sophisticated thread use put on a more formal foundation. The hard work was done by an industry group led by Professor Doug Lea.

An Aside on Design Patterns

There's an area of increasing importance in OOP technology, known as design patterns. A design pattern is a set of steps for doing something, like a recipe is a set of steps for cooking something.

There is a key book on the topic called Design Patterns—Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison Wesley, 1994: ISBN 0-201-63361-2). The book would be a lot more valuable if it were easier to read and gave examples that ordinary programmers could relate to. In spite of its dry, excessively academic style, it's an important text.

As the authors explain, design patterns describe simple, repeatable solutions to specific problems in object-oriented software design. They capture solutions that have been refined over time; hence, they aren't typically the first code that comes to mind unless you know about them. They are code idioms writ large. They are not unusual or amazing, or tied to any one language. Giving names to the common idioms and describing them, helps reuse. Some common idioms/design patterns are shown in Table 14-1.

Table 14-1. Design patterns overview

Design pattern	Purpose and use
Abstract Factory	Supplies an interface to create any of several related objects without specifying their concrete classes. The Factory figures out the precise class that is needed and constructs one of those for you.
Adapter	Converts the interface of a class into another interface that the client can use directly. Adapter lets classes work together that couldn't otherwise. Think "hose to sprinkler interface adapter."
Observer	Defines a many-to-one dependency between objects, so that when the observed object changes state, all the Observers are notified and can act accordingly. Think "monitoring the progress of something coming in over the network."
Singleton	Ensure that only one object is instantiated from some class, and provide a global point of access to it.
Command	Encapsulate a request as an object, rather than a method call. That lets you parameterize servers to handle different requests, and support requests that can be undone.

The recommended book describes a couple of dozen design patterns, and repays further study.

Further Reading

There are several great websites giving additional information on the SCJP tests described in the Light Relief section. Marcus Green has one at www.jchq.net/, and Bill Brogden's is at www.lanw.com/java/javacert/. These sites have examples of the kinds of questions you'll face, FAQs, advice, and suggestions on books that prepare you for the tests.

Exercises

1. Give three examples of when threads might be used to an advantage in a program. Describe a circumstance when it would not be advantageous to use threads.
2. What are the two ways of creating a new thread in Java?
3. Take your favorite sorting algorithm and make it multithreaded. Hint: A recursive partitioning algorithm like quicksort is the best candidate for this. Quicksort simply divides the array to be sorted into two pieces, then moves numbers about until all the numbers in one piece are smaller (or at least no larger) than all the numbers in the other piece. Repeat the algorithm on each of the pieces. When the pieces consist of just one element, the array is sorted.
4. What resources are consumed by each individual thread?
5. Refer back to the text giving the information about the synchronization bug in the thread library. What code change would you recommend to fix that bug and why? Look at the source code for Vector in the current run-time library. Did they follow your recommendation?
6. Rewrite the code example in the previous chapter that shows how to test a range of prime numbers for primes. Give the argument range to the threads by using thread local storage.

Some Light Relief—Are You Certifiable? I Am.

These Light Relief sections are pieces of "infotainment" about Java and the computer industry. This one is heavy on the info, and light on the 'tainment.

Sun Microsystems, the company that originated Java, started offering a Java test and pass certificate to programmers back in 1996. People take these tests for different reasons. Some companies send their employees on Java training courses and buy them this test at the end so they will have a qualification they can take away with them. Other people (like me) took the test at the invitation of the folks at Sun running the program to help them calibrate the results. Some people just like an extra qualification to put on their resume.

The Java Technology Certifications

Java 2 Platform, Standard Edition:

- Sun Certified Java Programmer
- Sun Certified Java Developer

Java 2 Platform, Enterprise Edition:

- Sun Certified Web Component Developer
- Sun Certified Enterprise Architect
- Sun Certified Business Component Developer

Java 2 Platform, Micro Edition:

- Sun Certified Mobile Application Developer

It is not (yet) common to see a job advert that mentions Java certification as a prerequisite for the position, but I think it's safe to assume that if you were a hiring manager trying to choose between two otherwise equal candidates, one of

Chapter 15. Explanation <Generics>

The biggest, most far-reaching new feature to come into Java with the JDK 1.5 release is "type genericity"—the ability to pass types as arguments, just the way we can pass values as arguments. In this chapter we'll take a close look at generic types, see what software problem they solve, and describe how they were retro-fitted into the language and run-time system (not an easy thing).

Explanation <Generics>

Generics introduced some new syntax to Java – writing a typename in angle brackets, like this:

```
LinkedList <Character>
```

That is read as LinkedList of Character. So this chapter title is read as Explanation of Generics. It's a nudge toward getting familiar with the new syntax ;-)

We'll start with a reminder on terminology—if we don't keep the terminology straight here, we're lost! We'll finish with a

Terminology Refresher: Parameters Versus Arguments

Everyone is familiar with the concept of methods having parameters. When you write the method, you write the parameters in a comma-separated list. Here's the declaration of the power() function in Java.lang.Math:

```
static double pow(double a, double b) { /*more*/ }
```

The method returns the value of a raised to the power b. The variables a and b are parameters to the method. Here's the point: The method will calculate that value for any a and b. Parameters let us write that method once, use it on any pairs of numbers, and refer to the different numbers coming into the method by unchanging names in the method body.

Arguments are the converse. At the place where we invoke the method, we write the method name and supply the specific variables or values that we want the method to work on, in this particular case:

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

double result = pow(10.0, PI );
```

The literal 10.0 and the constant Math.PI (the closest approximation to PI that a double can hold) are the arguments for this call. Other calls may have different arguments.

```
result = pow(x, y);
```

The compiler plants code to do a parameter = argument assignment for each parameter on every call. This takes place right before the call, in a method prolog that you don't see.

In a single sentence, the concept of generics is "We want to have a parameter that represents a type, not a value. We'll instantiate objects-of-the-generic-class using different types (Integer, Timestamp, Double, Thread) as the type

The Problem that Generic Code Addresses

Generics, also known as templates or parametric polymorphism, seem to be a popular feature in modern programming languages. Ada has them, so does C++, and so does Microsoft's copy of Java, C#. The overall design for Java generics was published in 1998, long before C# was launched. C# is again following Java's lead and adopting generics, although with a different design. The popularity might lead you to think that generics solve a big and important problem.

If you look at some of the written material justifying generics in Java, however, a different picture emerges. Some texts suggest that a purpose of generics is to reduce the amount of casting needed, particularly when using the Java data structure classes known as the collection classes. From JDK 1.2 (where collections were introduced) up to and including JDK 1.4, collections held Objects. Variables of any class type could be put into any collection, because all class types are compatible with the type used in a collection, Object. When you retrieved something from a collection, you had to figure out what type it really was, then cast it back to that.

The Java Collection classes

JDK 1.2 introduced some really robust data structure container classes. There are about 20 classes and a dozen interfaces kept in the package java.util.

These classes are termed the collection classes, and they do some amazing things. They maintain standard data structures, such as lists, sets, and maps (pairing of a key and value) and several subtypes of these. You can create a data structure that holds your own objects merely by instantiating one of these types.

The collection classes were updated to work with generic types in the JDK 1.5 release. Generics can work with all software libraries, not just the collection classes, but the collection classes are the main beneficiaries within the Java API. The chapter on collection classes comes right after this chapter and gives practical examples of generic use.

Usually, your collections just hold one kind of type, e.g. a given collection of yours holds String objects. A different collection holds Timestamps. A typical collection does not hold a mixed bag of String, Integer, Date, Thread, JFrame, and Timestamp objects. So figuring out what's in the collection isn't a problem. And casting something back to the right class isn't a big deal either:

```
String result = (String) myDict.get(i);  // without generics
```

What Generic Code Looks Like

There are three parts to generics:

1. Declare the class which will have generic type parameters
2. Declare/instantiate an object of that class, passing the actual type arguments to it.
3. Invoke methods on the object of the instantiated generic class.

Using the instantiated generic class is no different to using a non-generic class, so we won't spend a lot of time looking at that. The class declaration has to come before any instantiations, so let's look at declarations first.

A use of generics outside the Collection classes

The class java.lang.Class has been updated to take a generic parameter. Recall that the run-time system creates a separate Class object for each class in your program, when it loads the class. Class allows programs to get information about the classes it's using. Here's how Class was used before generics:

```
Class t = Timestamp.class;   // old style variable declaration

Timestamp ts = (Timestamp) t.newInstance();// old style, cast needed
```

Now when you declare a Class object, you may tell the compiler what type it will hold.

```
Class <Timestamp> t = Timestamp.class;  // declare with generic param

Timestamp ts = t.newInstance();    // with generics, no cast needed
```

The purpose of this change is to tell the compiler to which type you are going to dedicate the Class

Generic Interfaces

Interfaces, as well as classes, can be parameterized with generic types. As you'd expect, it looks exactly the same as when a class is parameterized. We'll walk through a complete "before and after" example here, because we'll use it in the next section as part of a more complicated generic parameter declaration.

This example uses the standard java.lang.Comparable interface type that defines a compareTo() method used for comparing two objects. The method returns an int which is negative, zero, or positive to indicate this is smaller, equal, or bigger than the argument.

java.lang.Comparable without generics

The Comparable interface looks like this in JDK 1.4:

```
public interface Comparable {
    int compareTo(Object obj)
}
```

A class that implements the interface would be declared like this in JDK 1.4:

```
public class Timestamp2 implements java.lang.Comparable {
    int hrs;
    int mins;
    int secs;

    public int compareTo(Object t) {
        Timestamp2 t2 = (Timestamp2) t;
        return (this.hrs - t2.hrs);
    }
```

Bounds—Requiring a Type Parameter to Implement an Interface or Extend a Parent Class

You'll get more use out of generic types if there's a way to call additional methods on them, more than just the handful of methods implemented in java.lang.Object. You can call a specific method, such as compareTo(), on a generic type within the body of a generic class, if you can be sure that the type has such a method. There are two ways to ensure that some class has a given method:

- Make the class a child of some parent that has the method, or
- Make the class implement an interface that has the method.

In general terms, we want a way to tell a generic that one of the parameter types must implement some interface or extend some parent. We express this with a Java generics feature called bounds. Bounds are limitations or restrictions put on type parameters.

Bounds say "The type that you use as an argument to the generic must extend this class" or "... must implement this interface". The purpose of bounds is to inform the compiler of methods that are guaranteed to be present in the type argument. Then we can call them, and it is guaranteed not to fail at run-time with a "method not found" exception.

An ordinary generic parameter is written as:

```
class SomeGenericClass <X>
```

We want a way to add this kind of condition to X:

```
<X has-to-extend-or-implement Comparable>
```

Then we can call Comparable's methods on X, like this:

Some Light Relief—On Computable Numbers with an Application to the Entscheidungsproblem

Generics in Java got me thinking about methods and subroutines in general. Computer science is a young enough field that it might be possible to pinpoint where subroutines were first used, and maybe even who came up with the idea. Like the answer to "What was the first computer?" the origin of subroutines is more a question about definitions than about history. Depending on how you define it, the first computer was:

- The ENIAC (Electronic Numerator, Integrator, Analyzer and Calculator/Computer [accounts differ]). Built by John Mauchley and J. Presper Eckert at the Moore School in the University of Pennsylvania, it was dedicated in February 1946. As originally built, it was not a stored program computer. ENIAC led to a follow-on project, EDVAC (Electronic Discrete Variable Automatic Calculator/Computer), and eventually a spin-off company that built Univac and morphed into the company of the same name (today known as Unisys).
- The SSEC (Selective Sequence Electronic Calculator) in January 1948, built by Wallace Eckert of IBM (no relation to John Eckert). Programs were read from paper tape or plugboards, but the system had the ability to cache in memory, and in theory operate like a stored program computer some of the time.
- The Manchester Mark 1 in June 1948, built by Fred Williams, Tom Kilburn and a team at Manchester University, England. This is the first machine that everyone recognizes as a computer, because it is the first system that stored its programs in the same memory used for data. The Manchester Mark 1 is unrelated to the Harvard Mark 1 built by IBM in 1943. The Harvard Mark 1 was a big calculator built out of relays, and driven by programs on paper tape (like a player piano).
- The EDSAC (Electronic Delay Storage Automatic Computer) in May 1949 by Maurice Wilkes's team at Cambridge University in England. This was the first operational, full-scale stored-program computer. It was closely based on the EDVAC design.

There are other candidates for the title "first computer", too. I would characterize the evolution as, before World War II—paper designs, during World War II—electronic calculators, and right after World War II—the first true computers. The Manchester Mark I is the one I am putting my money on as the very first true computer. They rebuilt the Mark I to commemorate its golden anniversary in 1998, and there's a plaque on the wall of the building where it was originally constructed. Last time I was in Manchester, England I took a photo of it.

One of the first programmers on the Manchester Mark I was legendary computer pioneer Alan Turing. Even before he joined the Mark I team in 1948, he sent them a routine to do long division. When I get time, I want to write a simulator in Java for the Mark 1— imagine the thrill of coding that retraces the footsteps of a great pioneer! Turing wrote the first ever Programmer's Handbook, some of which you can find online at http://www.computer50.org/kgill/mark1/progman.html.

Alan Turing had worked on systems that encapsulated all three phases of early computer evolution: paper designs, electronic calculators, and true computers. His 1937 paper, which I generically re-used for the title of this light relief (ho ho), On computable numbers, with an application to the Entscheidungsproblem appeared in summer 1936 in the

Part 3: Server-side Java

Chapter 16. Collections

We come now to a most important part of Java: the java.util and related packages that provide off-the-shelf data structures for your programs.

Computer science has a couple of dozen standard data structures (such as the linked list, the binary tree, the stack, etc). Java provides many of these standard data structures directly as library classes, and makes it easy for you to implement additional ones in a consistent way. These data structures hold collections of objects, called elements, and the library is known as the Java Collection Framework.

The Java Collection Framework has been updated to use generics starting in JDK 1.5, and you need to read Chapter 15 to understand this chapter. Collections are expected to be homogeneous: an instance of a collection class should only hold elements of one type. The code change is that you use generics to tell the compiler what that type is. The compiler will then detect attempts to add an object of the wrong type into a collection.

Collection API

We mentioned there are a couple of dozen basic data structures in computer science. Java provides the most important of these (including all three mentioned earlier) in a set of library classes. There is a common interface, java.util.Collection, giving the method signatures for insertion, deletion, and other common operations. Whatever java.util data structures you use, you can access them in a uniform way.

There are three main data structure classes that implement java.util.Collection:

- List. Lists can contain duplicate elements. Lists are kept in the order that elements are added. A list might contain an element for each tire sold in a tire store on one day. In this example, if someone buys four "Road Hugger" tires, four identical elements will get added to the list.
- Set. Sets cannot contain duplicate elements. Sets have their own idea of order, which is not the order that elements are added. A Set might contain "all customer objects who have not bought from us in the last six months".
- Queue. 

Queues were added in JDK 1.5. They are intended for holding elements prior to processing. Queues came into Java to help Threads. They typically deliver elements in a FIFO (first in, first out) order, but don't have to. You might use a queue to hold service requests while waiting for a server thread to become free.

On top of these three basic structures, there are many additional flavors or variations:

- Lists that are kept in an array (allowing fast random access)
- Lists in which each element points to adjacent elements rather than being stored next to them (enabling lists of arbitrary size)
- Sets specialized for fast access to enum objects
- Sets that support "copy on write" operations for sharing mostly read-only data between threads,
- Blocking queues of various types that wait for space before completing an add().

There are also a few simple data structures from JDK 1.1, updated to implement Collection: Vector and Stack.

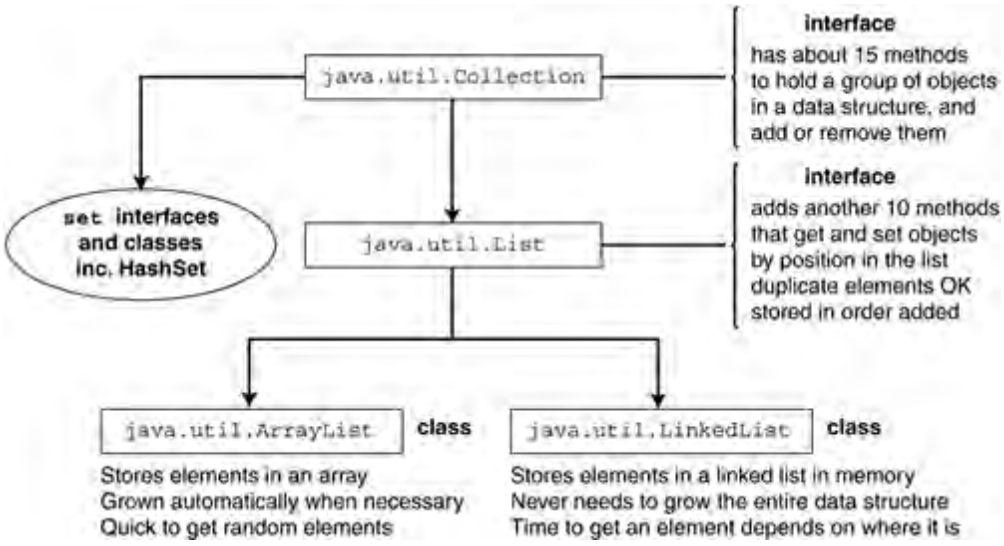
There is a fourth basic data structure in java.util. It doesn't implement the Collection interface because it's about pairs of

List, LinkedList, and ArrayList

One of the concrete classes that implements Collection is java.util.LinkedList. This class is just a plain old "Data Structures 101" forward-and-backward-linked list. You provide separate objects that you want stored, and the library class does the work of setting up and copying references to make a list. You start by instantiating an empty LinkedList, then anything you add is put on the list as a new element.

The class LinkedList implements List which implements Collection. The exact parent-child relationship of interfaces is shown in Figure 16-1.

Figure 16-1. Interface relationship



[View full size image]

The interface java.util.List adds about another ten methods to the elementary data access methods specified in java.util.Collection. Slightly simplified, the methods are:

```
public interface java.util.List<E> extends java.util.Collection <E> {

    public boolean addAll(int index, Collection);

    public Object get(int index);

    public Object set(int index, E element);

    public void add(int index, E element);

    public Object remove(int index);

    public int indexOf(Object o);
```

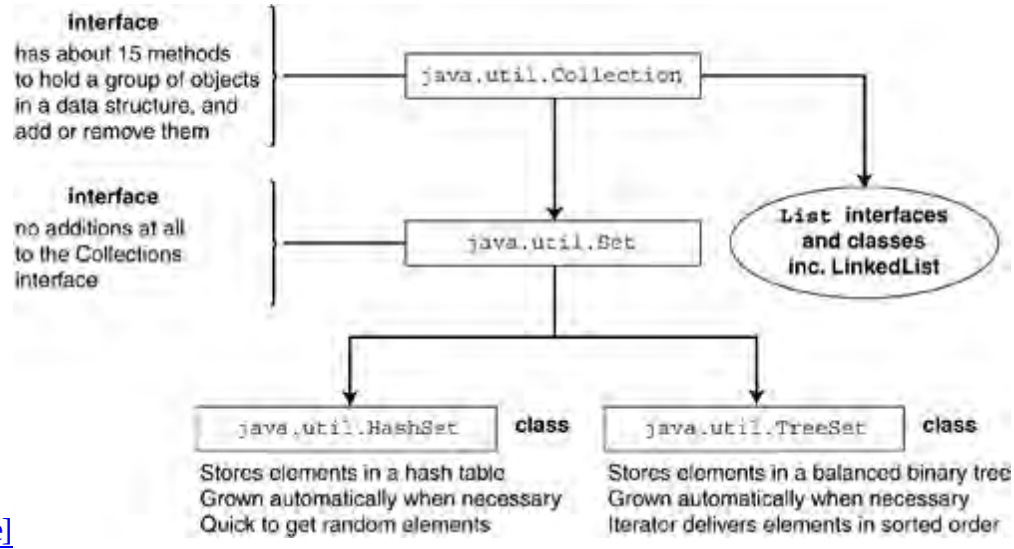
Set, HashSet, and SortedSet

One of the concrete classes that implements Collection is java.util.HashSet. We briefly visited HashSet at the beginning of this section on Collections, and now it is time to expand on it a little. This class is just a plain old "Data Structures 101" hash table.

You provide each object that will be stored as data in the hash table, and the library class does the work of maintaining the table and putting the elements in the right places. You start by instantiating an empty HashSet, then anything you add is placed in the table. If an object is already in the table, it won't be added again.

The class HashSet implements Set which implements Collection. The exact parent-child relationship of interfaces is shown in Figure 16-2.

Figure 16-2. HashSet relationship



[View full size image]

The interface java.util.Set does not add any methods at all to the elementary data access methods specified in java.util.Collection. It is written as a separate interface to help document and represent the design of the Collections Framework. It shows the symmetry between classes that are Sets, and classes that are Lists. As a reminder, the Collection interface, and thus the Set interface, looks like this.

```
public interface java.util.Set<E> extends Collection<E> {

     // basic operations

     public int size();
```

The Collections Helper Class

As we mentioned above, there is a "helper" class that contains about 50 static methods and nested classes that operate on or return a Collection. Here's the API for that class Collections, simplified to omit generic parameters.

We'll first show the class, then look at generic parameters in more detail.

```
public class Collections extends Object {

    public static final Set EMPTY_SET;

    public static final List EMPTY_LIST;

    public static final Map EMPTY_MAP;


    public static void sort(List);

    public static void sort(List, Comparator);

    public static int binarySearch(List, Object);

    public static int iteratorBinarySearch(List, Object);

    public static int binarySearch(List, Object, Comparator);

    public static int iteratorBinarySearch(List, Object, Comparator);

    public static void reverse(List);

    public static void shuffle(List);

    public static void shuffle(List, Random);

    public static void swap(List, int, int);

    public static void fill(List, Object);

    public static void copy(List, List);

    public static void rotate(List, int);

    public static boolean replaceAll(List, Object, Object);

    public static int indexOfSubList(List, List);

    public static int lastIndexOfSubList(List, List);

    public static List nCopies(int, Object);

    public static List list(Enumeration);
```

Wildcard Parameters and Generic Methods

The final piece of generics relates to the interaction between generic type parameters and methods. Most of the generic features we have seen so far are intended for API and library programmers. The main thing regular programmers need to remember is that:

```
public class HashSet <E> { /*more code*/
```

means that (to use this class) you will declare and instantiate a HashSet to work with a specific type:

```
HashSet<Timestamp> ht = new HashSet<Timestamp>();
```

The generic features presented in this section are different. These features are intended for use by regular programmers. You'll need these features in your own code, when you use somebody else's generic class.

There are two main generic-related features we are going to cover. Both of them concern methods and generics. The two features are:

- Expressing various bounds on a generic that is used as a method parameter. Here's an example method signature with a generic parameter:

  ```
  static void shuffle(List<Double> list)
  ```

  We need a way to get rid of that too-detailed <Double> and express "this method can work with a List of <AnyType> or <AnySubclassOf T> or <AnySuperclassOf T>".

- Parameterizing a method, (rather than a class or interface) with a generic parameter. From our experience with generic classes, we would expect to write a generic method something like this:

Wildcarding a Generic Parameter

Say we want to write a method that will shuffle the elements in a Collection. We can shuffle a list regardless of what type the elements are. We don't want to have to write one shuffle method that works on List<Integer> and another for List<String> and another for List<Byte>. We need to give shuffle() a parameter that is a "List of anything" and have it move elements around. The usual way to express "List of anything" is "List of Object", which in the new world of generics is written "List <Object>"

However, there's a big problem with passing around a nice, type-safe, homogeneous "List <String>" and letting someone untrustworthy, like our shuffle() method, get their hands on it as a "List <Object>" A parameter that points to a "List <Object>" can easily be used to add any Objects to the list: Strings, Integers, Timestamps, or anything else. The whole generics initiative is about limiting collections to only hold one type of object, and endowing the compiler with the information to check this. Unhappily List<Object> would blow our type-safe homogeneous collection model out of the water.

There's a piece of new syntax introduced here to express "parameterized type, but I don't care what it was parameterized with, they all match". It's called a wildcard, and it's written like this:

```
LinkedList <?>
```

You pronounce it "LinkedList of unknown", and it's intended to make you think of regular expression symbols where a "?" matches any single character. LinkedList<?> is the superclass of every LinkedList<T>. Collection<?> is "any class that implements Collection". Wildcards can be used with any parameterized type. This is the actual declaration of the shuffle() method in class java.util.Collections

```
static void shuffle(List<?> mylist, Random rnd)
```

Shuffle will thus take an argument that is a List of any type, just as we need.

With <?> come a couple of rules to preserve type safety:

- The compiler will treat parameterized types as incompatible with each other. I.e. List<A> and List<B> are incompatible and can't be assigned to each other, even when the types A and B are parent and child. The

Generic Methods

So far we have seen the wildcard feature used to give the methods you write some minimal knowledge about parameters that are generic types. You may be surprised (appalled, dismayed, horrified, or even delighted) to learn that individual methods can also be generic. In other words, there is a way to parameterize your methods with one or more type parameters.

As with classes, the type parameters are spelled out in angle brackets before the variable parameters. The type parameter comes immediately before the return type for the method, and after any access modifier. Here is the printFirst() method from the previous section rewritten to be a generic method:

```
public <T> void printFirst(LinkedList <T> e) { }
```

The generic parameter, T, comes before the method name to make the job of parsing easier for the compiler. It also separates the generic parameters away from the method signature. Once you have given names to the type parameters, like T here, you can use them in the argument list or the method body. Here we are using T to say that printFirst() takes an argument which is a LinkedList of T, whose name is e.

It may be that the return type uses the parameter type in some way, in which case you'll have two sets of angle brackets together, as in this example from the Java API:

```
static <T> Set<T> emptySet()
```

static	**<T>**	**Set<T>**	**emptySet()**
Modifiers	Generic parameter	Return type	Method signature

The method emptySet() has one type parameter, T, and it returns a value "set of T".

Let's emphasize that this feature is intended for your methods that have parameters that are collection types. If you don't write code to pass around and process collection types, you probably won't need to use generic methods.

Summary of Collection

- JDK 1.2 introduced support for Collections, also known as "container classes." There are two basic interfaces that extend Collection in different directions, Set and List.
- A List is a collection that has an order associated with its elements. That order is the order in which the elements were added to the collection. There does not need to be any logical relationship between elements.
- The List interface is implemented by two concrete classes, LinkedList and ArrayList. LinkedList is a doubly linked list. The time to access elements depends on where they are in the list, but there is no overhead to growing or shrinking the collection. ArrayList is a list stored as an array. It provides quick access to any element, but it is expensive to grow or shrink the array.
- A Set is a collection that never has any duplicate objects. Unlike a mathematical set, the objects may come in some order, but it is not the order in which you add them to the collection.
- The Set interface is implemented by two concrete classes, HashSet and TreeSet. HashSet is a set backed by hash table. It is quick to add and retrieve elements, but the elements are in no particular order. TreeSet is a set backed by a red/black tree. The elements are kept in sorted order, so it costs more to insert them.
- JDK 1.5 introduced support for generic classes, interfaces, and methods. The collection classes are the main user of these.

We saw the helper class for collections, and there is a similar one for arrays. These support some very useful sorting and extraction utilities for collections and arrays. It's possible to flatten any collection into an array, although you'll probably never need to.

All of these features together make up the Collections Framework, and there is one additional piece to it, which we will review in the next section. So far, all these data structures have dealt with individual objects. There is another interface backed by concrete classes that you can use to process pairs of key/values together. This is the Map interface.

Map, HashMap, and TreeMap

Map is a data structure interface that connects keys and values. Each key has at most one associated value. Some examples of key/value pairs would be driver's license number and "Object representing a licensed driver", username and password, ISBN and book title. The first of each of these pairs is a key for uniquely retrieving the second. The most obvious way of storing a Map is as a two-column table, and you can get fancier from there.

The Map interface provides three views onto its collection of keys/values. You can see the map as a collection of keys, or as a collection of values, or as a collection of key/value pairs. Once you have these collections back, you can invoke the standard Collection method to get an iterator and visit all the elements.

The order of a map is defined as the order in which the iterators on a map's collection views return their elements. Some Map implementations, like the TreeMap class, make specific guarantees as to their order; others, like the HashMap, do not.

The Map interface looks like this:

```
public interface java.util.Map <K, V> {

     public java.util.Set<K> keySet();         // get keys

     public java.util.Collection<V> values();  // get values

     public java.util.Set<Map.Entry<K,V>> entrySet();// get mappings


     public boolean containsKey(Object);

     public boolean containsValue(Object);


     public V get(Object key);

     public V put(K key, V value);

     public void putAll(Map<? extends K,? extends V> t);


     public V remove(Object key);

     public boolean isEmpty();

     public int size();

     public void clear();
```

Exercises

1. (Collections) Here is the outline of a class that provides access to its data structure through an iterator. Write the iterator.

```
  class storage {

 private Object[] data = new Object[256];

 // don't allow access to anything not yet stored
 private int nextEmptySlot = 0;
 private int i=0;

 public Iterator iterator() {
    // returns a class that iterates over the data array
    return new Iterator() {
    // insert the body of the inner class here
    // 3 methods: remove(), hasNext(), next()
  };
 }
}
```

2. (Collections) Take the storage class from the previous question and add all the code necessary to make it implement the Collection interface. Make the type of the object stored a generic parameter. Limit the number of objects in the collection, and hence the size of the array to 128 elements. Throw an UnsupportedOperationException if someone tries to add element number 129.
3. (Collections) Take the storage class from the previous question and remove any limit on the number of data items held in the collection. If you try to add an element when there is no more room in the array, then allocate a new, larger array, and use System.arraycopy() to fill it with the contents of the old array. Hint: Add the extra code in a subclass that overrides the methods of the parent class. That way, you simply inherit all the things that

Some Light Relief—Early Names for Java

When Shakespeare wrote "A rose by any other name would smell as sweet" (Romeo and Juliet, Act II, Sc 1), he hadn't seen the effects of modern marketing, consumer branding, and product positioning. Names are incredibly important to the positioning of a product. First impressions count for way too much, and (you'll be horrified to learn) people do indeed judge books by their covers.

Names sometimes arise from the most fortuitous of circumstances. Linus Torvalds (himself named after two-time Nobel laureate Linus Pauling) planned to call Linux "Freax" because it was free and freaky. But his friend Ari Lemmke gave him an FTP directory called /pub/OS/Linux, and the name stuck. While still at college, Red Hat founder Marc Ewing was given his grandfather's Cornell lacrosse team cap (with red and white stripes). Marc lost track of it, and searched with increasing desperation. The manual for the original Red Hat Linux beta appealed to readers to return Marc's Red Hat if found. We're lucky that distribution isn't called Red-and-White-striped Lacrosse Cap Linux.

Would Java have been a runaway success if it was called "Oak"? It was called Oak for most of its early development years starting in 1991. James Gosling, the master programmer, Sun VP, and Sun Fellow behind the language, named it after the view from his office window. James's office was at 2180 Sand Hill Road, Palo Alto, California, on the fourth floor of the old Bank of America building. From the window he could see a large oak tree. When he needed a name for the language he was developing, what could be more handy than to look out of the window waiting for inspiration to strike?

The name Oak was short, natural, familiar, but it didn't have that edge, that buzz, that sizzle successful products need. But worse than that, years later, as Sun was planning to go public with Oak, the name failed a trademark search! There was a video card manufacturer called Oak Technology who held the rights. As a stop-gap, Oak was bogusly renamed to O.A.K. for Object Application Kernel. This was done so the source code didn't need to be changed, but a real name change was needed.

Oak was recognizably Java at this point. Patrick Naughton and Jonathan Payne had used it to create the Webrunner prototype web browser. The pair then got downloadable applets working in a weekend. By now it was January 1995, and the 1.0a2 semi-public alpha release was planned for March 1995. So a brainstorming session was called for all Oak team members, to come up with ideas for a new name. Table 16-1 is the shortlist taken into the meeting, and Java isn't even on it!

Table 16-1. Proposed names before Java was suggested

Candidate name	Comments from the Oak team
Calypso	Trademarked already?
Caravelle	Style of ship used for exploration (the Nina, Pinta and Santa Maria were caravelles).
C+-	

Chapter 17. Simple Input Output

SCSI-wuzzy was a bus.

SCSI-wuzzy caused no fuss.

SCSI-wuzzy wasn't very SCSI, was he?

Designing input/output (henceforth "I/O") libraries is a lot harder than it might appear. There have been some

Getting to Know Java I/O

We'll start off by giving you an idea of what the problems are with Java I/O (we'll get all the gripes off our chest in one go). Then we'll look at the purpose of the java.io and mention the names of some of the many packages that make up the Java I/O library.

Java I/O problems

So what are the problems with Java I/O? It boils down to this: it is a very large library and it took six release with four different implementations to get it mostly right. Excessive use of the wrapper design pattern confuses rather than helps. It relies too much on the C I/O API.

Package java.io has about 75 classes and interfaces. Size alone doesn't make an API poor, but there are too many low-value classes (like SequenceInputStream) and it took until JDK 1.5 to get decent support for interactive I/O. Sun got I/O internationalization wrong the first time and had to add many more classes in JDK 1.1 to properly cope with Unicode. The Java I/O package is not intuitive to use, and there are a number of peculiar design choices often reflecting the use of the underlying standard C I/O library. That's the same mistake the Fortran team committed 40 years earlier. Finally, because this library makes heavy use of wrapper classes, you may need to use two or three classes to do simple I/O. Apart from all that, it's really a fine API.

Purpose of java.io package

Enough of the criticism. Obviously, the purpose of the java.io package is to conduct I/O on data and on objects. You will use the java.io package and others to write your data into disk files, into sockets, into URLs, and to the system console, and to read it back again. There is some support for formatting character data, and for processing zip and jar files. Several other packages are involved in this, and the key ones are listed in Table 17-1. The point of presenting this table is to indicate the vast range of Java I/O features.

Table 17-1. Java packages involved with I/O

Package name	Purpose
java.io	Contains the 75 or so classes and interfaces for I/O.
java.nio	New in JDK 1.4. An API for memory-mapped I/O, non-blocking I/O, and file locking.
java.text	For formatting text, dates, numbers, and messages according to national preferences and conventions. This task can now be achieved more easily by using java.util.Formatter in conjunction with PrintWriter.format or printf.
java.util.regex	New in JDK 1.4. For matching a string against a pattern

Design Philosophy

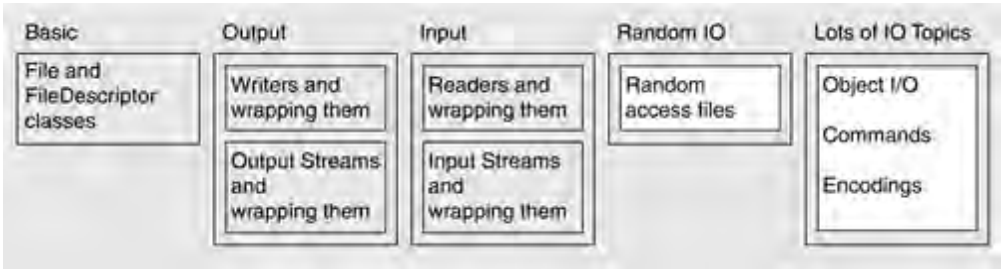
The design philosophy for Java I/O is based on these principles:

- Programs that do I/O should be portable, even though I/O has some non-portable aspects. Platform differences in file names and line terminators must be handled in a way that ensures the code runs everywhere.
- I/O is based on streams. A stream has a physical "device" at one end, like a file or a location in memory. That physical device is managed by some class, and you wrap (layer) additional logical classes on top of that for specific kinds of I/O.
- There are lots of small classes that do one thing, instead of a few big classes that do many things. There is one class that interprets data in binary format, and another class that reads data from a file. If you want to read binary data from a file, you use both of these classes. The constructors make it convenient to use the classes together.

We'll see examples of these principles throughout the chapter. This is a long chapter, but a worthy one. To help you get the most out of it, Figure 17-1 represents the topics that will be covered, and how they are grouped together. As you can see, many of the I/O topics are freestanding and only relate to each other in a general way. If you feel lost at some point in this chapter, refer back to Figure 17-1.

Figure 17-1. Topics in I/O

[View full size image]



The first three boxes in Figure 17-1 are covered in this chapter, and the next two in Chapter 18, "Advanced Input Output." We'll start by looking at the File and FileDescriptor classes that provide a convenient way to represent a filename in Java. Then we'll cover Java support for interactive input (new in JDK 1.5). We'll round out the chapter by reviewing the most widely used input/output classes.

Portability of I/O

The basic portability approach of the Java run-time library is to have the same method do slightly different things appropriate to each platform. The standard end-of-line sequence on Windows is "carriage return, linefeed," while on Unix it is just "linefeed." Any library method that writes an end-of-line sequence, such as System.out.println(), will output

The Class java.io.File

The class java.io.File should really be called "Filename" since most of its methods are concerned with querying and adjusting filename, and pathname information, not the contents of a file. File can't actually do any I/O. Directory, filename and pathname information is often called "metadata," meaning "data about data." Methods of java.io.File allow you to access metadata to:

- Return a File object from a String containing a pathname
- Test whether a file exists, is readable/writable, or is a directory
- Say how many bytes are in the file and when it was last modified
- Delete the file, or create directory paths
- Get various forms of the file pathname.

Here are all the important public members of java.io.File. Method names are in bold for visibility.

Public members of java.io.File

```
public class File implements Serializable, Comparable {

    public static final char separatorChar;

    public static final String separator;

    public static final char pathSeparatorChar;

    public static final String pathSeparator;


// constructors:

    public File(String path);

    public File(String directory,String file);

    public File(File directory,String file);
```

Keyboard I/O

For the first nine years of its life, Java did not have any routines to do console input. The official spin from Sun was that you were supposed to use a GUI for interactive I/O. If you wanted to get input from the command line, you had to roll your own support using several classes. It was ugly, and it needlessly gave a bad impression of Java to generations of student programmers. Most teachers gave their students a class to do console I/O, and only taught the API support much later.

The JDK 1.5 release introduces a simple text scanner that can be used to read input. There is also a new method that provides a feature essentially the same as the printf() method used in C. We'll show examples of both of these over the next several pages.

System.in, out, and err

On all Unix operating systems and on Windows, three file descriptors are automatically opened by the shell and given to every process when it starts. File descriptor '0' is used for the standard input of the process. File descriptor '1' is used for the standard output of the process, and file descriptor '2' is used for the standard error of the process. The convention is so common because it is a part of the C language API.

These three standard connections are known as "standard in," "standard out," and "standard err" or error. Normally, the standard input gets input from the keyboard, while standard output and standard error write data to the terminal from which the process was started. Every Java program contains two predefined PrintStreams, known as "out" and "err." They are kept in Java.lang.System, and represent the command line output and error output, respectively. There is also an input stream called System.in that is the command line input. This is also referred to as console I/O or terminal I/O.

You can redirect the standard error, in, or out streams to a file or another stream (such as a socket) using these static methods:

```
System.setErr(PrintStream err);

System.setIn(InputStream in);

System.setOut(PrintStream out);
```

Stdin and stdout are used for interactive I/O. Stderr is intended for error messages only. That way, if the output of a program is redirected somewhere, the error messages still appear on the console.

Keyboard input

Output

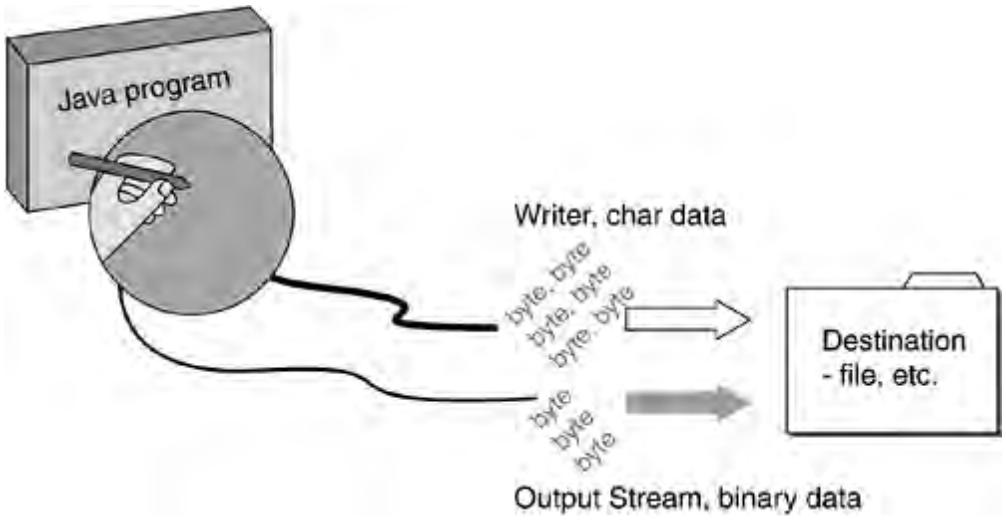
At last we're at the point where we'll talk about one of the major confusions in Java I/O, use of char (double byte) versus single byte I/O.

Readers versus OutputStreams

Originally, Java only had stream classes, and the streams only operated on bytes of data. However, characters in Java are two bytes wide, and byte-oriented I/O did not properly cope with internationalization.

So a wider type of stream was introduced in JDK 1.1 specifically for character-based I/O. Reader classes are able to get Unicode character input two bytes at a time. Writer classes are able to do Unicode character output two bytes at a time, as shown in Figure 17-2. Input and output streams operate on data one byte at a time.

Figure 17-2. Your program outputs data into a stream or writer



Java I/O is overly complicated to the newcomer because it relies on layering or wrapping classes. You need two or three classes to do anything. You determine exactly what classes to use by reviewing three attributes of your output:

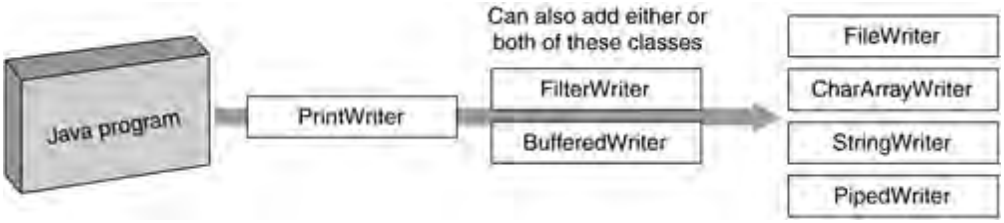
- Whether you need 2-byte characters
- What you will be writing to (file, socket etc.)
- What format you want to write (binary or printable)

Here are the steps for choosing the classes in detail:

Wrapping Additional Output Classes

As Figure 17-3 suggests, the java.io package has more classes that you can wrap around a Writer or OutputStream. The additional Writer wrapping classes can do things like filter the text as it goes by, or buffer it for performance. The additional OutputStream wrapping classes can bridge to a Writer class, filter, buffer, encrypt, and/or compress that data! When you wrap classes, only write from the outermost one. Otherwise, your I/O may get scrambled due to internal buffering.

Figure 17-3. Wrapping more writer classes



[View full size image]

All these additional wrappers go between what we have termed the top layer classes and the bottom layer classes. It makes sense—the bottom layer (like FileWriter) is the ultimate destination and nothing can come below that. The top-layer class has the methods that pour the bits into the stack, so nothing can come above that. Additional wrappers go between the two end points.

We'll present two examples here. First, we'll show an example of using a FilterWriter to modify text as it gets written. We'll also show how a BufferedWriter is used in this example. The second example will use OutputStream 8-bit wrappers to create a zip file archive. If you haven't seen this before, you may be surprised at the small amount of code that achieves these great results.

Additional writer wrappers

FilterWriter is an abstract class for you to extend and override. It provides the opportunity to look at and modify 16-bit characters as they are output. You could do the same thing by extending many of the other writer classes, but using this class makes your purpose explicit.

Here is an example program that post-processes the stream written into it and changes all "1"s to "2"s. A Filter can do other things like count lines, correct spelling mistakes, calculate checksums, or write an encrypted or compressed stream.

A Filter to replace chars

In this example, the Filter overrides just two of the write() methods, but you may need to override any or all of the FilterWriter methods depending on which of them may be called by your code.

Input

The classes to do input are mostly the flip side of the output classes we have already seen. Java programs access external data by instantiating a stream on the data source. Each place from which an input stream can flow has a class dedicated to getting that kind of input. Input is read from a stream of data representing the file, pipe, socket, memory array, or whatever. If you want to read 16-bit characters, you use a Reader class. If you want to read binary bytes or ASCII, you use an input stream.

Inputting double-byte characters

As usual, first decide between binary and character I/O, then choose your class based on where the data is coming from. For reading double-byte character data, you will use one of the Reader classes shown in Table 17-6. Note the symmetry with the Writer classes.

Table 17-6. Choose the Reader class based on where the Input comes from

Get input from	java.io class	Constructors

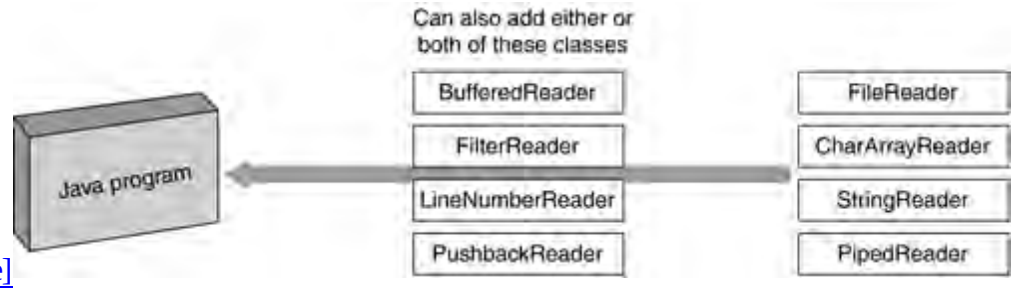
A file.	FileReader	`FileReader(java.lang.String) throws java.io.FileNotFoundException;`
		`FileReader(java.io.File) throws java.io.FileNotFoundException;`
		`FileReader(java.io.FileDescriptor);`

A char array in your program. You read from the array passed to the constructor.	CharArrayReader	`CharArrayReader(char[]);` `CharArrayReader(char[],int from,int to);`
A String in your program. You read from the String passed to the constructor.	StringReader	StringReader(String s)
A pipe that is written by a PipedWriter in another thread.	PipedReader	`PipedReader()` `PipedReader(PipedWriter source)`

Reader Wrappers

There is the usual variety of wrapper classes that can wrap a Reader, shown in the Figure 17-4.

Figure 17-4. Wrapping the Reader classes



[View full size image]

Classes that wrap readers

The classes that wrap a Reader are:

- BufferedReader. This class can provide a performance boost, and also has a readLine() method. The BufferedReader needs to wrap the class that actually accesses the data (e.g., the FileReader or whatever). Other classes may be layered on top of the BufferedReader, too.
- FilterReader. You subclass FilterReader, and your overriding methods allow you to see and modify individual characters as they come in—before the rest of your program sees them.
- LineNumberReader. This class keeps track of the line number count on this stream. You can find out the input line you are currently on by calling getLineNumber(). This class doesn't really offer enough value to justify its existence. It was written to support the first Java compiler and included in the API for no better reason than "Hey, we already had it written".
- PushbackReader. This class maintains an internal buffer that allows characters to be "pushed back" into the stream after they have been read, allowing the next read to get them again. The default buffer size is one character, but there is a constructor that lets you specify a larger size. You might use this if you were assembling successive characters into a number and you come to a character that can't be part of a number. You will push it back into the input stream so it can be ignored, but kept available for the next read attempt. This is less necessary now that we have the Scanner class.

Inputting ASCII Characters and Binary Values

You choose an input stream when you want to bring bytes into your program. As with Readers, you decide where you want to read from, and choose one of the InputStream classes accordingly (see Table 17-7).

Table 17-7. Choose the InputStream class based on the source of the input

Read binary input from	java.io Class	Constructors

A file	FileInputStream	`public FileInputStream(java.lang.String) throws java.io.FileNotFoundException;`
		`public FileInputStream(java.io.File) throws java.io.FileNotFoundException;`
		`public FileInputStream(java.io.FileDescriptor);`

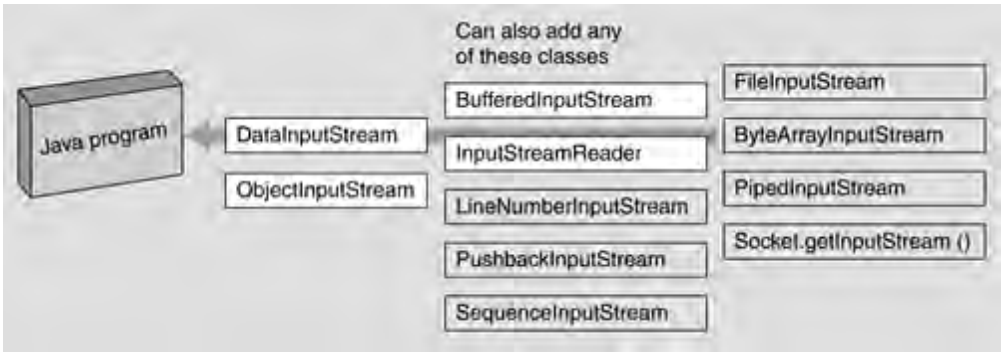
A byte array in your program	ByteArrayInputStream	`public ByteArrayInputStream(byte[]);` `public ByteArrayInputStream(byte[], int from, int len);`
A pipe to be read by a PipedOutput-Stream in another thread	PipedInputStream	`public PipedInputStream(java.io.PipedOutputStream) throws java.io.IOException;` `public PipedInputStream();`
A StringBuffer object	StringBufferInputStream	This class has been deprecated; don't

Input Stream Wrappers

We have seen several times how a basic I/O class can be wrapped or "decorated" by another I/O class of the same parent class. So it should be no surprise that an InputStream can have a BufferedInputStream and/or a subclass of FilterInputStream interposed between the FileInputStream (or other data source) and the DataInputStream.

There is quite a variety of InputStreams that can decorate the basic access classes. Figure 17-5 shows some, but by no means all, of the most popular classes.

Figure 17-5. Classes that wrap InputStreams



[View full size image]

You can wrap any or all of the following output streams onto your original InputStream:

- BufferedInputStream. This class must directly wrap the input source (e.g., the FileInputStream) to get the most performance benefit. You want the buffering to start as early as possible. Wrap any other classes around the buffered input stream.
- Your subclass of FilterInputStream. You will extend the class and override some or all of the read methods to filter the data on the way in.
- LineNumberInputStream. This class keeps track of the number of newlines it has seen in the input stream.
- PushbackInputStream. This class allows an arbitrary amount of data to be "pushed back" or returned to the input stream where it is available for re-reading. You might do this when you are trying to assemble a number out of digits in the input stream and you read past the end of the number.
- SequenceInputStream. This class provides the effect of gluing several input streams together, one after the other, so that as one stream is exhausted you seamlessly start reading from the next. You might use this when your data is spread across several data files with a similar format.
-

Further Reading

Table 17-9 shows some online resources for more information on other I/O packages.

Table 17-9. Online resources for other I/O packages

Package	Online resources
Image I/O	java.sun.com/products/java-media/jai/whatis.html
	API docs package javax.imageio
Speech	Programmer's guide at java.sun.com/products/java-media/speech/forDevelopers/jsapi-guide/Preface.html
Logging	Overview at java.sun.com/j2se/1.5/docs/guide/util/logging/overview.html
	API docs package java.util.logging
Communication ports (serial port)	See the home page at java.sun.com/products/javacomm/
	The home page has a pointer to a user guide.
Printing	What Java calls "printing" is actually "painting GUI objects onto paper". Be careful not to mistakenly read information on the older print APIs.
	java.sun.com/printing/
	API User Guide: java.sun.com/j2se/1.5.0/docs/guide/jps/spec/JPSTOC.fm.html
	API docs package javax.print. The API docs contain a small printing example.

Exercises

1. Measure the difference between buffered and non-buffered I/O operating with 10K 1-byte writes and one 10KB write, repeated 10,000 times in a loop. Draw a graph to illustrate your results. How do the results change with a buffer size of 128KB, 256KB, 512KB?
2. Modify the program that does a hex dump of a file so that it also outputs any printable bytes in a set of columns to the right of the hex dump on each line. Print the character if it has a printable form, and print a "." if it does not. This ensures that lines are the same length and columns line up.
3. Write a Java program whose output at run-time is an exact duplicate of the program's source code. Now that we have printf, the shortest Java program to do this is well under a page of code.
4. Write a program that prints a table of printable ISO 8859-1 characters and their bit patterns.
5. Rewrite the hex dumper utility to use one or more Filter classes. The first filter can turn binary bytes into the equivalent printable hex characters. The second filter can insert the addresses and newlines at appropriate points.
6. Rewrite the deCSS utility (see Some Light Relief—The Illegal Prime Number!) in Java. For extra credit, look up the algorithms on the web to actually carry out the decryption of an encoded DVD stream, and write Java code to do that. Does it run quickly enough to decode and play in real time? Explain why or why not.

Some Light Relief—The Illegal Prime Number!

By now everyone is familiar with DVDs—originally an acronym for "Digital Video Disc," later changed to "Digital Versatile Disk" for pointless marketing reasons. DVDs are similar to CD-ROMs in many ways, with a crucial difference that commercial DVDs can hold about 8 GBytes, or more than ten times as much data as a CD. The tracks and the bits in the tracks are packed closer together on a DVD, which is why DVD players can read CDs but not vice-versa. If you use a suitable compression technology, you can actually squeeze up to 133 minutes of high-resolution video with several soundtracks and subtitles onto a DVD. The compression is essential, and the movie industry uses the MPEG-2 algorithm that was designed for this purpose, and which provides 40-1 compression. The more efficient MPEG-4 (DivX) compression, which provides another fivefold reduction, is also being introduced.

However, since the movie industry doesn't want to be Napstered (have their content ripped off and broadcast for free on the Internet), they encrypt the MPEG-2 files using an algorithm called the Content Scrambling System, or CSS. If you do a directory listing of a DVD, you'll see some large .VOB files. These are Video OBjects, a fancy name for content-scrambled .MPG2 files. Every maker of DVD players on the planet is supposed to license the decryption algorithm from the DVD Copy Control Association (DVD-CCA) for a fee, and they impose several restrictions on the player. DCC-CCA is believed to be a subsidiary of Matsushita, the company mainly responsible for the development of DVD and CSS. Some of its restrictions take away rights that consumers have long enjoyed under copyright law. They seem more geared towards controlling what consumers can do, rather than dealing with problems of rip-offs and piracy.

So what are the restrictions that licensed DVD players have to impose? CSS encryption allows the DVD industry to force region restrictions into all DVD players. There are six geographic regions (North America, Europe, etc.) and in 1999 they added a seventh for DVDs intended for airplanes. A player in region one will refuse to play disks labelled as belonging to any other region. Region restrictions allow the movie industry to sell the same DVD at different prices in different markets. It prevents any DVDs you buy on business trips outside your region from being played on your home system. The CSS encryption also prevents you from fast-forwarding past the copyright warning or advertisements or any other content the producer wants you to see. You can sell preview commercials for a much higher price if people cannot skip past them. Some people speculate that CSS is also paving the way for more restrictions such as DVDs with a limited lifetime or limited number of viewings. The movie industry blows a lot of smoke about CSS preventing large-scale piracy, but CSS does nothing whatever to prevent pirates from copying DVDs. Its only purpose is enforcing use limitations that take away the legal rights of consumers.

For a long time, there was no software to play DVDs available for Linux. If you had a shelf full of DVDs that you had bought, you could play them all on your TV or Windows box, but because of the CSS restrictions, not on your Linux or Solaris system. The CSS restrictions were the equivalent of a book publisher enforcing a restriction that you could read a book under incandescent lighting but not under fluorescent lighting or daylight. No one in the Linux community had the means to pay the "CSS tax" to the DVD-CCA. Then, in October 1999, anonymous German hackers reverse-engineered CSS. The source code to decrypt DVDs was published on the web by a 15-year-old boy from Norway. The program was called "deCSS" because it reverses CSS, turning the encrypted files into ordinary MPEG-2 files.

There then followed an extraordinary game of "whack-a-mole" as the DVD-CCA and the Motion Picture Association of America (MPAA) tried to chase the source code around the web and sue it out of existence. That game continues today. The 15-year-old boy was hauled off by the foolish Norwegian police who also seized his PC and his cell phone. The cell phone was a lucky guess on the part of the cops, because he did actually have a back-up copy of the source stored in it. Cell phones these days are effectively quite powerful computers, and a quarter of a billion cell phones (2004 figure) contain a JVM. After lengthy legal proceedings, the kid was eventually found not guilty.

Chapter 18. Advanced Input Output

I-O!

I-O!

It's off to work we go!

I-O, I/O, I/O.

This chapter follows on from the basic I/O chapter and provides information on more advanced I/O techniques in about a half-dozen sections, mostly independent. The section on the new I/O package (new in JDK 1.4 that is, with the name java.nio) is needed to understand the three other JDK 1.4 features of pattern matching, file locking, and character sets. You can read other sections in any order, or indeed skip the entire chapter now and return when you need information

Random Access File

Until now we have seen how input and output is done in a serial mode. However, sometimes we want to be able to move around inside a file and write to different locations, or read from different locations, without having to scan all the data. Traveling in a file stream in such a manner is called "random access".

Java has a random access file class that can do these operations. This is not an indexed sequential file or a direct access file of the kind supported by data processing languages. Those two techniques require the creation and maintenance of special indexes to address the file. The class java.io.RandomAccessFile does not use indices, but lets you access a file as though it were an array of bytes. There is an index or pointer that says where the next read or write will take place. You set the position of the file pointer with the method seek(), giving an argument that is the absolute offset into the file. When you set the file pointer, the next access will take place at the offset indicated. Because of the way operating systems store file contents, moving around in a file does not necessarily involve inefficient reading and rereading.

class java.io.RandomAccessFile

```
public class RandomAccessFile implements java.io.DataOutput, java.io.DataInput {

    public RandomAccessFile(java.lang.String,java.lang.String) throws

         java.io.FileNotFoundException;

    public RandomAccessFile(java.io.File,java.lang.String) throws java.io

➥.FileNotFoundException;

    public final java.nio.channels.FileChannel getChannel();  // used for locking file

    public final java.io.FileDescriptor getFD() throws java.io.IOException;

    public native int read() throws java.io.IOException;

    public int read(byte[], int, int) throws java.io.IOException;

    public int read(byte[]) throws java.io.IOException;

    public final void readFully(byte[]) throws java.io.IOException;

    public final void readFully(byte[], int, int) throws java.io.IOException;

    public int skipBytes(int) throws java.io.IOException;

    public native void write(int) throws java.io.IOException;

    public void write(byte[]) throws java.io.IOException;

    public void write(byte[], int, int) throws java.io.IOException;
```

Running Commands and Getting Output From Them

This section explains how to execute a program from Java and read the output of that program back into your application. Just as a reminder, the use of OS commands in your code destroys portability.

If you can live with that limitation, there are four general steps to executing a program from your Java program.

1. Get the object representing the current run-time environment. A static method in class java.lang.Runtime does this.
2. Call the exec method in the run-time object, with your command as an argument string. Give the full path name to the executable, and make sure the executable really exists on the system. The call to exec() returns a Process object.
3. Connect the output of the Process (which will be coming to you as an input stream) to an input stream reader in your program.
4. You can either read individual characters from the input stream, or layer a BufferedReader on it and read a line at a time as in the code below.

Many books use the example of getting a list of files in a directory, even though there is already a Java method to do that in class File. The following code in class java.io.File will return an array of Strings, one string for each file name in the directory:

```
File f = new File(".");

String myFiles[] = f.list();
```

The file with the special name of "." is used on many operating systems to refer to the current working directory. We will use a different example here. We will execute a command in our Java program to make a file read-only. On Linux or other Unix, the command used would be "/bin/chmod a-w", on Windows we use the "attrib +R" command as shown in the following code.

Executing the attrib command from Java

Formatted String Output

Support for formatted output was introduced with JDK 1.1, in package java.text. There is a class called java.text.DecimalFormat for formatting numbers, and a class DateFormat for formatting dates and times. Other classes provide support for collation (sorting order), and formatting program messages to users.

Most people now will prefer to use the standard formatting used in C and printf, available from JDK 1.5 in class java.util.Formatter. Please refer to the Chapter 17 for a description of these features.

Writing Objects to Disk

We've already seen how to write out strings, ints, doubles, etc., in both printable and binary forms. It may surprise you to learn that you can also write out, and later read back in, entire objects. When you serialize (write out) a single object to a data stream, it automatically saves the object, namely, all its instance data. If any of these non-static fields reference other objects, those objects are serialized too. That way, when you later deserialize (restore) the object, you get back the object and all its member fields pointing to all the things they pointed to before—everything needed to reconstitute the original object.

For example, if you serialize one element of a doubly linked list, everything it references, and everything the references reference, and so on, will be saved. For an element of a doubly linked list, that means the elements on each side of it, and the elements on each side of those (one of which is the original element—that doesn't get written out twice), and so on until the entire list has been written to disk.

The point of including everything that your object connects to, and all the things they connect to and so on, is to ensure that (when you read them back in) you can use those objects with the same state and contents that they had when you originally wrote them out. Doesn't this swell up the size of what you're writing until it's as big as your entire program? Objects contain references to their member fields, but members tend not to back reference the object of which they are a field. That means the links mostly go one way, and so serialized objects in practice remain a manageable size.

It's quite powerful to be able to do I/O on an entire graph of objects with one simple method call. If an object can be written to a stream, it can also be sent through a socket, compressed, encrypted, read out of a socket on another host, backed up onto a file, and later read back in again and reconstituted.

To make an object serializable, all you need do is make its class implement the Serializable interface. The interface java.io.Serializable doesn't have any methods or fields. It is an example of the Marker Interface design pattern. The purpose of requiring a class to implement an empty interface is to identify to other programmers and to the run-time library that it can be serialized. Here is a class that can be serialized:

```
package java.util;

public class Date implements java.io.Serializable { ...
```

Here is how you can serialize a Date object, and save it in a file:

```
// first create the file
```

New I/O Package

JDK 1.4 introduced a package called java.nio. "Nio" stands for "new I/O" (so the JDK 1.5 I/O improvements must be "even newer than new I/O"). The java.nio package and subpackages support four important features not previously well-provisioned in Java:

- A non-blocking I/O facility for writing scalable servers
- A file interface that supports locks and memory mapping
- A pattern-matching facility based on Perl-style regular expressions
- Character-set encoders and decoders

Instead of building these on top of streams or file descriptors, these features are implemented using two new concepts: buffers and channels.

The right way to understand a buffer is to think of it as a big array in memory that holds data. Just like an array, a buffer can only hold things that are all the same type. So you can have a byte buffer, a char buffer, a short, double, float, and int buffer. If your file contains a mixture of floats and ints, you can pull them out of a Byte Buffer. Byte Buffer has methods to get and put all the primitive types except boolean.

The idea behind the Buffer class is to have a region of memory which can be accessed from both native code and Java at the same time, and that region has some special I/O characteristics in the native OS. Buffer is a kind of "native" array—native code can access it directly, and Java can use method calls to get its hands on the contents.

Because a buffer is essentially an area of memory, it can do things relating to memory, like clear its contents, support read/write or read-only operations, give you a range of elements, and tell you how many data elements it contains.

The second new concept in java.nio is the channel. A channel is a connection between a buffer and something that can give or receive data, such as a file or a socket. Because a channel connects to an underlying physical device, it can do things relating to an I/O device like support read/writes or provide file locks. There are channel classes specialized for files, for sockets, for pipes, and so on. You may think of a channel as an alternative to a stream. It has fewer fancy features (no elaborate wrapper classes), but it may have higher performance.

When to use channel I/O

Memory Mapped I/O

Let's return to the topic of memory-mapped I/O. We stated that file channels/buffers are an alternative to reads/writes on streams. Memory-mapped I/O is an alternative to both, implemented as a refinement to channels/buffers. The whole point of mapped I/O is faster I/O. When transferring large amounts of data, mapped I/O can be faster because it uses virtual memory to make the file contents appear in your address space. It takes some initial setup and puts more work on the virtual memory subsystem, but mapped I/O avoids the extra copying from buffers in kernel memory into buffers in your process.

When you read with a stream, the OS reads from the disk into a buffer owned by the device driver and then moves the contents from kernel space to your buffer in user space. Memory mapping only needs a couple of bits twiddled in the VM system to say "that disk page is now part of this process address space." So why doesn't everyone use mapped I/O all the time? Kernel whackers do, and the rest of the world is still hearing about the feature. Also, it's not part of the ANSI C API, which is one of the most widely used I/O APIs.

It came from Multics

Mapped memory is also known as shared memory. As well as offering performance improvements for larger files, it can be used for bulk data transfer between cooperating processes that all map in the same file. These processes don't even have to be on the same system, as long as the same file is visible to each. Mapped files were first used in Multics, the 1960s operating system that was wildly over budget and schedule, but which was the stepfather of Unix (and thus the ancestor of Linux, MaxOS X, and Solaris, too).

When you do a map operation on a FileChannel to map a file into memory, your return value is a mapped byte buffer that is connected to the file. The run-time system is expected to use the operating system features for memory mapping. The result is that when you write in the buffer, that data appears in the file. If you read from the buffer, you get the data that is in the file. Everyone is familiar with the way an operating system can read an executable file and make the instructions appear in the address space of a process. Mapped I/O does essentially the same thing for data files. The signature of FileChannel's map method is:

```
MappedByteBuffer map(int mode, long position, int size)
     throws IOException;
```

The position argument is the offset in the file where you want the mapping to start. This will usually be offset zero, to start at the beginning. The size is the number of bytes that you want from the file. This will usually be myFile.length() to get the whole thing.

The mode argument is one of FileChannel.MapMode.READ_ONLY,

File Locking

As we saw in the threads chapter, sometimes you have two things going on at once in a program, and to keep them straight you may need to stop them from doing the same thing together. This situation can occur in file handling. An example would be a data file that several programs want to update at the same time. Let's say the last record in the file contains the total of all the other records in the file. When a program updates the file, it first writes the new data value, then it reads the current total, adjusts it for the new value, and writes it back. If another program should happen to come along at just the wrong time, both programs may read the old total, then they will both update it, but one update of the total will overwrite the other. Result: two data changes, but only one change to the total, so the file is now inaccurate.

One way to avoid this situation is to use threads and synchronize them on some suitable object. That only works when all the threads trying to update the file are running in one JVM. Many applications cannot accept that limitation, which is where file locking comes in. Using the new FileLock class introduced in JDK 1.4 as part of package java.nio, the programmer can lock part or all of a file for exclusive access.

As a reminder, a method called getChannel() has been added to each of the FileInputStream, FileOutputStream, and RandomAccessFile classes. Invoking the getChannel() method upon an instance of one of those classes will return a file channel connected to the underlying file.

Once you have the FileChannel object for an output file, you call its lock() or tryLock() method. Lock is a blocking call—it won't return until it has the lock, or the channel is closed, or the thread interrupted. The method tryLock() is not a blocking call. It returns at once, whether it got the lock or not. The return value from both these calls is a FileLock object or null. Both of these methods have variants that let you provide arguments to specify that a region of the file (rather than the whole thing) is locked. That allows finer control over how much different processes stay out of each others way, with consequently better performance.

Here is an example program that repeatedly tries to acquire a lock. When it gets the lock, it prints a message saying so, and sleeps for two seconds to simulate doing some work with the file. It then releases the lock. It sleeps a further third of a second and does the whole thing over again.

```
import java.io.*;

import java.nio.channels.*;

public class Lock {

    public static void main(String[] a) throws Exception {

       // Get a Channel for the file

       FileOutputStream fos = new FileOutputStream("data.txt");

       FileChannel fc = fos.getChannel();

       while (true) {
```

Charsets and Endian-ness

A character set, also known as an "encoding," is the set of bit patterns used to represent a set of characters. ASCII is one popular encoding. EBCDIC is a family of character sets with regional variations used on IBM mainframes. Unicode is a third encoding. Before describing the character sets available to Java, we need to explain "big-endian" and "little-endian" storage conventions.

Big-endian storage

Endian-ness refers to the order in memory in which bytes are stored for a multi-byte quantity. The term is a whimsical reference to the fable Gulliver's Travels, in which Jonathan Swift described a war between the Big-Endians and the Little-Endians, whose only difference was in where to crack open a hard-boiled egg. It was popularized in the famous paper, "On Holy Wars and a Plea for Peace" by Danny Cohen, USC/ISI IEN 137, dated April 1, 1980. It's a cool paper, well worth reading, and you can easily find it on the web if you search.

All modern computer architectures are byte-addressable, meaning every byte of main memory has a unique address. If you store a multibyte datum in several successive addresses, should the most significant byte of the datum go at the highest address or the lowest address? Big-endian means that the most significant byte of an integer is stored at the lowest address, and the least significant byte at the highest address (the big end comes first).

In other words, if you have an int value of 0x11223344, the four bytes:

```
0x11, 0x22, 0x33, 0x44
```

will be laid out in memory as follows on a big-endian system:

```
base address +0:  0x11

base address +1:  0x22

base address +2:  0x33

base address +3:  0x44
```

The SPARC, Motorola 68K, and the IBM 390 series are all big-endian architectures. Big-endian has the advantage of telling if a number is positive or negative by just looking at the first byte. It's also the way we read and write numbers in

Exercises

1. (Random Access Files) Create a data file that contains ten ints and a long. The long value holds the total of the ints. Write a program that repeatedly checks the total is correct, chooses one of the ten ints at random, changes it to a random value, and updates the total.
2. (Serialization) Write a program that creates a java.util.Vector object and adds various arrays to it. Serialize the Vector to a file. Read the Vector back in, and write the code to check that it contains the same contents that it had when written out. Modify the serialized file to give one of the arrays different content. (Hint: strings are easy to update in a file.) Check that your program detects the change.
3. (Channels, Buffers) Write a program that has two threads. One thread should engage in I/O using a channel. It should contain a handler for ClosedByInterruptException and AsynchronousCloseException. The other thread should call the interrupt method of the first thread. The first thread should catch the exception, re-open another channel and carry on with I/O. Put the whole thing in a loop and run it overnight. Is it reliable enough that it is still running in the morning? Is your operating system reliable enough to cope with this?
4. (Channels, Buffers) Write a program to measure the difference in performance between I/O through a direct (mapped) buffer and a non-direct buffer. Your program should output a 512 KByte array of ints 1,000 times in a loop to the same random access file. Is the performance of input any different? Account for any differences.
5. (Channels, Buffers) Take the example program that shows memory-mapped I/O for an input file and modify it so that the output is done by mapped I/O too.
6. (Locking) Take the program written for the random access file exercise (question 1 above). Run two copies of the program and demonstrate that the total quickly goes awry.
7. (Locking) Update the program in the previous exercise to protect the file by locking it. Run several copies of the program overnight to show that it works correctly.
8. (Locking) Update the program in the previous exercise so that it locks only the regions of the file that it is going to update: the random int and the total field. Measure the performance of this code and compare it with the code from the previous exercise. Is it faster or slower? Account for any performance differences.
9. (Encodings) Write a program that prints a neat table of EBCDIC and ASCII characters and their associated bit patterns.
10. (Encodings) Write a codeset program to write out data in all the standard encodings and confirm how the bytes are swapped with the different character sets.

Some Light Relief—The Illegal T-shirt!

The story so far: The light relief in the previous chapter described how the Motion Picture Association of America, combined with a shadowy Japanese-controlled organization known as the DVD-CCA, were furiously trying to stuff the toothpaste of DVD decryption back into the tube of secrecy. Their efforts were aided by a bad U.S. law known as the 1998 Digital Millennium Copyright Act.

The DMCA is not the first piece of bad law affecting the Internet. The Communications Decency Act lasted less than a year before the U.S. Supreme Court struck it down as unconstitutional in 1997. The DMCA is a bad law because it tilts the balance between consumers and copyright holders too heavily toward copyright holders. Many kinds of ordinary legal uses of DVDs, such as playing them with your DVD player of choice on the computer of your choice, have become effectively illegal under the DMCA.

Every once in a while, common sense collides with the law. It's not quite the irresistible force meeting the immovable object because the law must always yield to common sense in the long run. But it can be pretty entertaining in the short run. Take a look at the picture of this T-shirt (Figure 18-1).

Figure 18-1. Wear a T-shirt; go to jail!

The T-shirt contains a few lines of C code on the back, and the DVD Copy Control Association (DVD-CCA) is suing the vendor to try to drive this T-shirt off the market!

The dispute centers around those few lines of C code on the back of the shirt. They are just regular lines of C code,

Part 4: Client Java

Chapter 19. Regular Expressions

This chapter provides an introduction to some classes in the java.util and related packages that have utilities for your code to use. We start with the regular expression support that allows you to do pattern matching of Strings. After that, we look at the Date and Calendar-related classes, which have confused and frustrated many people.

The chapter finishes with a brief look at some data structure classes that predate Collections, but are still useful. The rest of this book is devoted to explaining more Java libraries and showing examples of their use. Let's get going with regular expressions and pattern matching.

Regular Expressions And Pattern Matching

This section uses the I/O features described in the previous chapters and describes the regular expression pattern matching feature that was introduced with JDK 1.4.

If you have typed "dir *.java" to see all the Java files in a directory, you have used a regular expression for pattern matching. A regular expression is a String that can contain some special characters to help you match patterns in text. In this case, the asterisk is shorthand for "any characters at all". The name "regular expression" was coined by American mathematician Stephen Kleene who developed the expressions as a notation for describing what he called "the algebra of regular sets". The asterisk is also called a "Kleene star".

JDK 1.4 introduced a package called java.util.regex that supports the use of regular expressions. Using the classes in that package lets you answer questions like "Does this kind of pattern occur anywhere in that String?", and you can split Strings apart and create new Strings with changed contents. These sorts of operations are very useful in the following contexts:

- Web searches (you can use regular expressions in many search engines).
- Email filtering (discard email where the "From:" line matches well-known spammers)
- Text-manipulation tasks. Source code editors usually have a way to search using regular expressions. If you don't know how to do pattern matching in the editor you use to edit programs, you aren't yet reaching your full potential as a programmer. Plus, it's a great way to beguile other programmers who look over your shoulder.

There's lots of good news about regular expressions in Java. First, the language of regular expressions (the way you form regular expressions, the special symbols and their meaning) is very similar to that used by Perl. There are a few obscure things supported by Java that Perl 5 doesn't support, and vice versa. Java is less forgiving about badly formed expressions. But if you already know Perl, there's less to learn about Java pattern matching. If you don't use Perl, your Java regex knowledge will get you jump-started.

Best of all, Java regular expressions are simple. There are only three classes in the package, and one of those is an exception! Well, you can't really judge the complexity of a library by the number of classes it has, but regular expressions are straightforward. Pattern matching is important, and we'll cover it in some detail.

Let's say you have a String somewhere, and you want to look for a pattern in it. A pattern will be something like "at least one letter or digit (and maybe many) followed by a colon followed by a space, followed by at least one letter or digit (and maybe many)". The steps to look for a pattern in a String, s, include:

1. You specify the pattern with a String, p, holding a regular expression representing the pattern.
2.

Calendar Utilities

The JDK 1.0 support for dates was poorly designed. With the benefit of hindsight, it would have been better to throw it out and start over again, but backward compatibility was seen as the more important goal. In JDK 1.1 most of the constructors and methods of java.util.Date class were deprecated, and other classes were provided to offer better support for time zones and internationalization.

The following classes specifically relate to date and time:

- The class Date represents a specific instant in time with millisecond precision. It's really a timestamp, not a date.
- The class Calendar is an abstract class for converting between a Date object and a set of integer fields such as year, month, day, and hour.
- The class GregorianCalendar is the only concrete subclass of Calendar in the jdk. It does the date-to-fields conversions for the calendar system in common use.
- The class DateFormat is an abstract class that lets you convert a Date to a printable String with fields in the way you want (for example, dd/mm/yy or dd.MMM.yyyy).
- The class SimpleDateFormat is the only concrete subclass of DateFormat in the jdk. It takes a format String and either parses a String to produce a date or takes a date and produces a String.
- The class TimeZone is an abstract class that represents a time zone offset and also calculates daylight savings time adjustments.
- The class SimpleTimeZone is the only concrete subclass of TimeZone in the JDK. The class defines an ordinary time zone with a simple daylight savings and daylight savings time period.

Not only was date and time support poorly designed, it was poorly implemented and full of bugs. The good news is that many of the bugs were corrected in 1.1.4 and 1.1.6. In JDK 1.2, the most common problems were corrected. This part of the JDK has been maintained by IBM.

You can instantiate Date with no arguments to represent the current moment.

Other Utility Classes

BitSet

This class maintains a set of bits that are identified by the value of an integer, like an array index. You can have over two billion individual bits in a set (if you have enough virtual memory), each of which can be queried, set, cleared, and so on. The bit set will increase dynamically as needed to accommodate extra bits you add.

```
public BitSet();  // constructor

public BitSet(int N);  // constructor for set of size N bits


public void or (BitSet s); // OR's one bit set against another.

public void set (int i); // sets bit number i.

public int size(); // returns the number of bits in the set.
```

Stack

The Stack class maintains a Last-In-First-Out stack of Objects. You can push (store) objects to arbitrary depth. The methods include:

```
public Object push(Object item);   // add to top of stack.


public Object pop();   // get top of stack.
```

In addition, there are three other methods not usually provided for stacks:

```
public boolean empty(); // nothing on the stack?
```

Further Reading

The book Java Regular Expressions: Taming the java.util.regex Engine by Mehran Habibi (published by APress, 2004) is generally regarded as a good study on the topic. If you are interested enough to want to play around with Perl, you can visit the www.perl.com/ website to find tutorials and download a free copy of the software. O'Reilly sponsors the Perl website, and they also support a corresponding "Java in the Enterprise" website at www.onjava.com/. They have some good Java articles and other resources.

See developer.java.sun.com/developer/technicalArticles/releases/1.4regex/ for more information on Java regular expressions.

Exercises

1. (Patterns, easy) Do a web search to find all the pieces that comprise a URL in its full generality. State what these pieces are. Write a pattern-matching program that checks URLs for validity (such as no embedded spaces). Show the output of your program when run on both good and bad URLs.
2. (Patterns, medium) An email message consists of an arbitrary number of headers, then a blank line, then the body of the text. The headers all have a label followed by a colon, a space, and some optional text all on one line. The "From:" and "To:" headers are required, and come first in that order. The pattern would begin something like this:

```
Pattern email = Pattern.compile(
          "^From: .*$"       // "From" line
          + "^To: .*$"       // "To" line
```

   Write the rest of a pattern to match an email message, and use capturing groups to separate the individual parts. Don't forget to make this a multiline pattern.
3. (Patterns, harder) Write a pattern that matches the Java regular expressions presented in this text. This is several days of work. Run your meta-pattern matcher on all the Java regular expression Strings shown in this chapter. Account for any that it does not accept.
4. (Math, medium) Convince yourself that the nextGaussian() numbers form a bell curve by writing a program to actually do it. Hint: You'll find it easier to plot a graph using ASCII text if you generate the values first, save them, and sort them into order before plotting the values. Which of the collection classes is a good choice to help with storing and sorting?

Some Light Relief—Exchanging Apples And Crays

There's an old story that "The people at Cray design their supercomputers with Apple systems, and the Apple designers use Crays!" Apart from this being a terrific example of recurring rotational serendipity (what goes around, comes around) is there any truth to it?

Like many urban legends, this one contains a nugget of truth. The 1991 Annual Report of Cray Research, Inc., contained a short article describing how Apple used a Cray for designing Macintosh cases. The Cray is used to simulate the injection molding of the plastic enclosure cases. The Mac II case was the first Apple system to benefit from the modeling, and the trial was successful. The simulation identified warping problems that were solved by prototyping, thus saving money in tooling and production. Apple also uses its Cray for simulating air flow inside the enclosure to check for hot spots. The Cray magazine also reported that the Apple PowerBook continues to use supercomputer simulations. (Cray Channels, Spring 1996, pp.10-12 "Apple Computer PowerBook computer molding simulation").

The inverse story holds that Seymour Cray himself used a Macintosh to design Crays. The story seems to have originated with an off-the-cuff remark from Seymour Cray, who had a Macintosh at home and used it to store some of his work for the Cray 3. Common sense suggests that the simulation of discrete circuitry (Verilog runs, logic analysis, and so on), which is part of all modern integrated circuit design, is done far more cost-effectively on a large server farm than on a microprocessor. Cray probably has a lot of supercomputer hardware ready for testing as it comes off the production line.

It's conceivable that a Macintosh could be used to draft the layout of blinking lights for the front of a Cray, or choose some nice color combinations, or some other non-CPU intensive work. A Macintosh is a very good system for writing design notes, sending email, and drawing diagrams, all of which are equally essential parts of designing a computer system.

The good folks at Cray Research have confirmed in a Cray Users' Group newsletter that they have a few Macs on the premises.While it's unlikely that they run logic simulations on their Macs, we can indeed chalk it up as only-slightly-varnished truth that "the people at Cray design their supercomputers with Apple systems, and the Apple designers use Crays".

All the action in supercomputers is in clusters these days, not in super-powerful monolithic mainframes. Also bear in mind that Moore's Law means that the desk-side G5 Mac in 2004 (236 MFLOP) is more powerful than the Cray 1 (160 MFLOP) super computer reigning nearly 30 years earlier in 1976.

Chapter 20. GUI Basics and Event Handling

All GUI libraries have four basic areas of functionality:

- Creation of user interface "controls" or components, such as scroll bars, buttons, and labels.
- Support for giving behavior to the controls by connecting GUI events (for example, clicking a button) to code that you write.
- Support for grouping and arranging the controls on the screen.
- Support for accessing window manager facilities, such as specifying which window has the input focus, reading JPEG and other image files, and printing.

Related graphics libraries also provide support for graphics operations, such as drawing an arc, filling a polygon, and clipping a rectangle. There may even be a complete 2-D and 3-D drawing library, similar to the ones in Java.

For the first couple of major releases, Java supported GUI operations with a package called java.awt. The "AWT" stands for "Abstract Window Toolkit." The AWT supported the portability goals of Java. The toolkit gave user programs a common binary window interface on systems with different native window systems. That's a very unusual feature, like having your favorite Macintosh program run on a Windows PC and still do GUI operations. You might wonder how it is done.

You can call methods in the AWT run-time to pop up a native menu, resize a window, get the location of a mouse click, and so on. Java then calls through to the native GUI library. Events coming back from the GUI go into the Java run-time, and are passed on to your code as appropriate. The Java bytecode is the same on each platform, but the

All About Event Handling

First, a few necessary words of explanation about the programming model for window systems. Unlike procedural programs in which things happen sequentially, windowing programs are asynchronous, because actions occur at unpredictable times. You never know which of the onscreen buttons, menus, frames, or other elements the user will touch next. Accordingly, a model known as event-driven programming is used.

In event-driven programming, the logic of your code is inverted. Instead of one flow of control from beginning to end, the run-time system sits in a "window main loop" simply waiting for user input. When the user clicks the mouse, the operating system passes it to the window manager which turns it into an event and passes it on to a handler you supplied earlier. This is known as a callback. Your handler is a callback routine, because the window system calls back to it when the event happens. Your event handler deals with the graphics event and any work associated with it.

If a button says "press here to read the file," your code must arrange for the file to be read when called. Handling a button event means noticing that it occurred and doing the associated action, but other events may involve some drawing on the screen. For example, dragging something with the mouse is just repeatedly drawing it under the mouse coordinates as it moves.

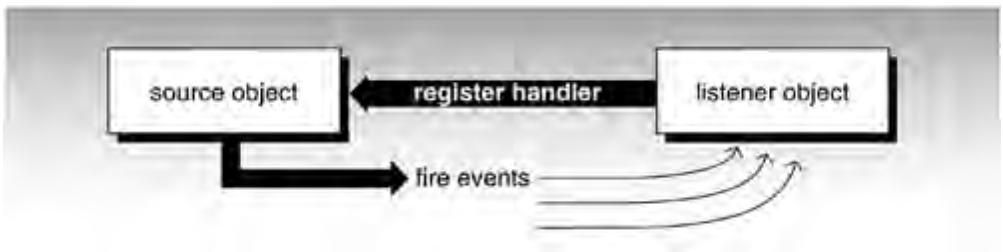
Java event model

The event model is the name for the framework that turns a GUI interaction (mouse click, menu selection, button press, etc.) into a call for your code to process it. The event model can also be used for something unrelated to the GUI, like a timer going off. In other words, the event model is the design for connecting your code to any kind of asynchronous actions, called events, for handling.

The window manager can't directly call your event-handling routines because the run-time library doesn't even see your code until it is asked to run it. Therefore, at run-time the event model has to be told which of your routines handle events.

Java originally used inheritance to tie together your code and the event model. JDK 1.1 introduced a better approach called the delegation-based model. Some of the Java documentation still refers to "1.1-style events," which is the current model. To get any events, your code has to begin by telling the window system, "send those events of yours to these methods of mine". You connect the controls that generate events by registering a callback with your event-handling classes, as Figure 20-1 shows.

Figure 20-1. How events are passed in JDK 1.1



[View full size image]

Tips for Slimming Down Handler Code

Inner classes are intended for event handlers. The inner classes allow you to put the event-handling class and method right next to where you declare the control or register the callback listener. Anonymous classes are a refinement of inner classes, allowing you to combine the definition of the class with the instance allocation. The following example shows the code rewritten using an anonymous class:

```
import javax.swing.*;

import java.awt.event.*;

public class CloseDemo2 {


    public static void main(String[] args) {

          JFrame jframe = new JFrame("Example");

          jframe.setSize(400,100);

          jframe.setVisible(true);


          jframe.addWindowListener( new WindowListener() { // anon. class

               public void windowClosing(WindowEvent e) {System.exit(0);}

               public void windowClosed(WindowEvent e) { }

               public void windowOpened(WindowEvent e) { }

               public void windowIconified(WindowEvent e) { }

               public void windowDeiconified(WindowEvent e) { }

               public void windowActivated(WindowEvent e) { }

               public void windowDeactivated(WindowEvent e) { }

          } ); // end of anonymous class.

    }

}
```

Try to compile and run this code example. Your CloseDemo2.java file generates class files called CloseDemo2.class and CloseDemo2$1.class. The second item represents the anonymous WindowListener inner class.

Summary of Event Handling

We have seen a specific example of handling the event generated by closing a Window. It can be written more compactly if you write it as an inner class or even as an anonymous class. You can junk even more unneeded code if you use an adapter class.

There are several kinds of events for the different controls: a button generates one kind of event, a text field another, and so on. To impose order and to divide them up according to what they do, there are approximately 12 individual Listener interfaces shown in Table 20-1 on page 507. They all work the same way: you write a handler class that implements the interface, and register it with the control. When the control fires an event, the method in the handler object that you registered is called.

The key points about GUI handling include the following:

- Each SomethingListener interface has one or more methods showing the signature of a method that is called when the corresponding SomethingEvent occurs.
- Your handler code implements the SomethingListener interface and therefore has methods with signatures that fulfill those promised in the interface.
- Each control has a method called addSomethingListener(). The addSomethinglistener() method takes a single argument, an object that implements the SomethingListener interface.
- Swing requires that all code that might affect GUI components be executed from the event-dispatching thread. A section in Chapter 21 explains this concept.
- You call addSomethingListener(), using an instance of your handler class as the parameter. This registers your object as the handler for that kind of event for that control.

The SomethingEvent class is a subclass of class AWTEvent and stores all the information about what just happened, where, and when. An object of the SomethingEvent class is passed to the method in the SomethingListener interface. It sounds more complicated than it is. Figure 20-4 shows the design pattern.

Figure 20-4. Design pattern of JDK 1.1 event handling

Exercises

1. Review the javadoc pages for classes and interfaces in package java.awt.event. How many classes and interfaces exist?
2. Write a program that displays a JFrame. Install a key listener and the three kinds of mouse listener on the frame. Print out each event that is received. Are you surprised at the number of mouse motion events?
3. The source code for the event interfaces and their methods can also be reviewed in the directory $JAVAHOME/src/java/awt/event. ($JAVAHOME is the location where you installed the release. On my system it is C:\jdk1.4.)

   Take a look at MouseWheelEvent.java, which shows how support for the new mouse wheels was added in JDK 1.4. What information can a mouse wheel event convey?
4. After doing the previous exercise, design and describe the event that represents a Zap. Zaps are delivered from the new "Wendy Wand" hardware that can be pointed at any component and invoked with a wink and a shake. Zaps have a location on the screen, a Zap-strength field (wimpy, medium, stun, or to-frog), and a Zap-Color.
5. Write a program that displays a JFrame and handles Zap events. Simulate Zap events by instantiating them and posting them to the event queue. Class java.awt.EventQueue has a method postEvent() that will do this for you. Perhaps you could make a mouse click generate a Zap event in its handler.

Some Light Relief—The Mouse That Roared

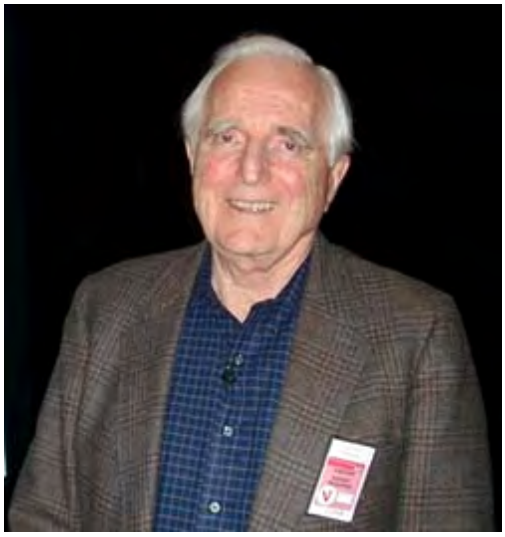
Just about everything computer-related must have been designed for the first time or done for the first time by somebody. At some point there must have been the first editor, the first debugger, the first core dump, the first disk drive.

Sometimes these events are surprisingly recent. Sometimes they are old. The term "core dump", which is a copy of the contents of a process's memory, is pretty old. It dates back to the early 1950s when computer memories really were composed of cores. Main memory was built out of tiny ferrite rings or "cores" threaded on fine wires which could induce or reverse a magnetic polarity in the cores. Memory was literally made up of cores that each held one bit, and the contents of memory was a "core dump". Switching time was slow, but it didn't matter because processors were slow too, with cycle times in the milliseconds.

One of the computer "firsts" was the first mouse pointing device. We even know where this was launched: at the Fall Joint Computer Conference in San Francisco in October 1968. It took another 16 years before memory and graphics software got cheap enough to bring the mouse into everyday use with the 1984 Apple Macintosh.

The pioneering inventor of the mouse was computer scientist Doug Englebart (pictured in Figure 20-5 in 2001), who worked at the Stanford Research Institute in Menlo Park, CA. Doug was interested in graphical displays and ways of improving the human-computer interface. He had the idea for a hand-operated pointing device in 1964, and it took four years to complete it.

Figure 20-5. Mouse inventor Englebart



At that time, there were no computers for personal use. Time-sharing was just beginning, and the only people with graphical displays were radar operators. Doug had to persuade managers at SRI to buy an $80,000 graphics console to support his research project. He could foresee a time when screens would replace teletypes, and computers would be cheap enough to have several at home.

Chapter 21. JFC and the Swing Package

In Chapter 20, we saw that the basic idea behind Java GUI programs is that you perform the following actions:

- Declare controls. You can subclass them to add to the behavior, but this is often unnecessary.
- Implement an interface to get the event handler that responds to control activity.
- Add the controls to a container. Again, subclassing is possible but frequently unnecessary.

Chapter 20 explained how to handle the events that controls generate. This chapter dives into the details of the controls themselves: what they look like, how you use them, and what they do. Chapter 22

Java Foundation Classes

Supporting a Java interface to the underlying native window libraries achieves the goal of making Java GUI programs highly portable, but it comes at the cost of inefficiency. Peer events must be translated into Java events before they can be handled by Java code. Worse, native libraries aren't identical on each platform, and sometimes the differences leak into the Java layer.

Example of how native library behavior leaked into the AWT

Sun never did get file name filtering to work for the AWT "file selection dialog" on Windows systems. This was bug 4031440 on the Java Developer Connection at java.sun.com/jdc.

To support FilenameFilter, the AWT FileDialog needs to issue a callback for each file to display, and you supply a FilenameFilter that can accept or reject the file. But on Win32, the FileDialog control works in a completely different way. It doesn't issue callbacks. Instead, it accepts simple wildcard patterns to match against file names. That's a reasonable alternative to FilenameFilters, but that model isn't supported by the current Java API. As a result, AWT filename filtering never worked on Win32.

This is an example of the difficulties of trying to make a common window library above several different native libraries. Swing solved these difficulties, since there is minimal reliance on native library support. The javax.swing.JFileChooser class lets the user select a file without dependence on the native components.

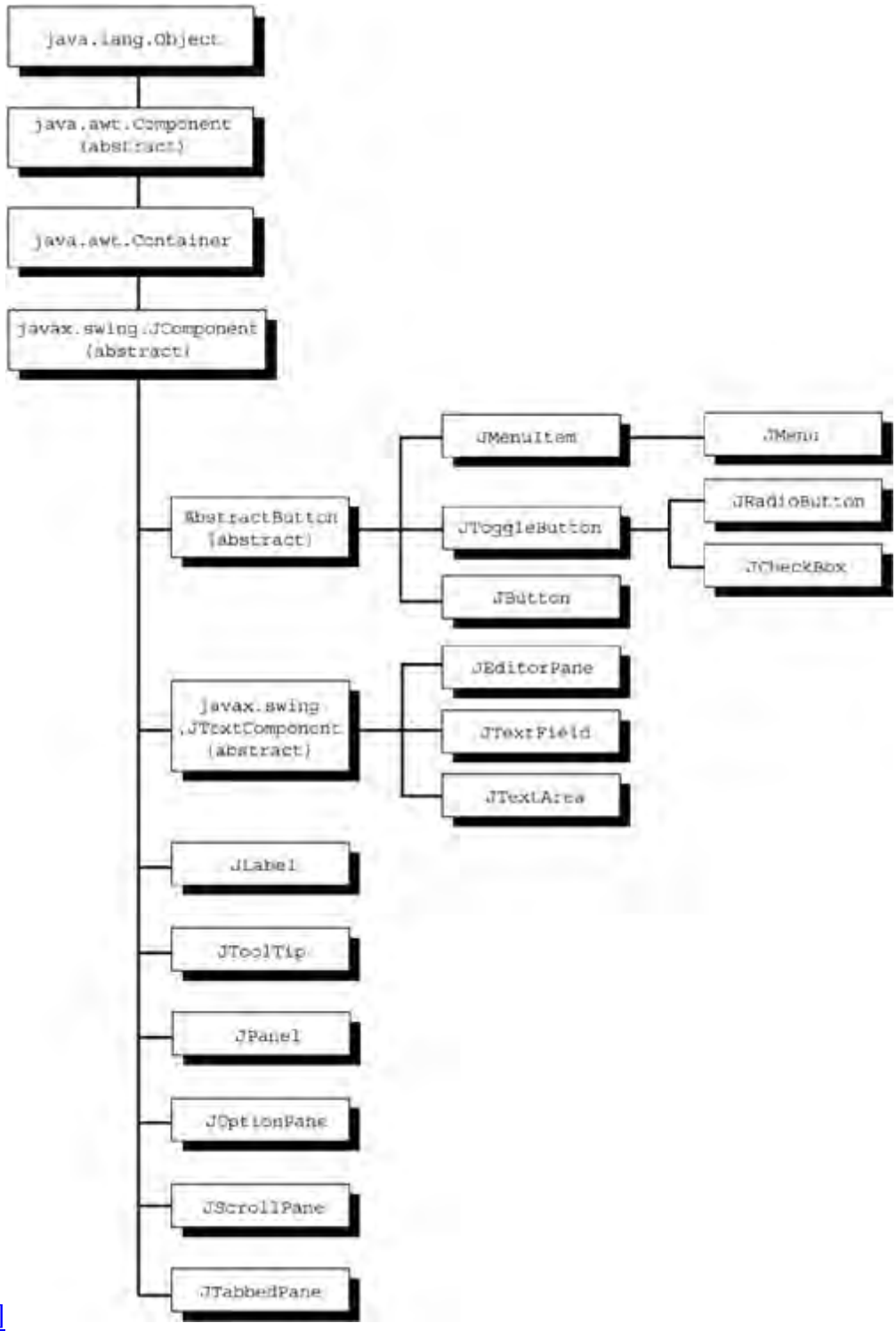
The Java Foundation Classes (JFC) are a set of GUI-related classes created to solve the AWT problem of platform idiosyncrasies. JFC also supports the following items:

- A pluggable look-and-feel, so that when you run the program, you can choose whether you want it to look like a Windows GUI, a Macintosh GUI, or some other style.
- An accessibility API for features like larger text for the visually impaired.
- The Java 2-D drawing API.
- A drag-and-drop library and an "undo last command" library.
-

All About Controls (JComponents)

We now have enough knowledge to start looking at individual controls in detail and to describe the kinds of events they can generate. Most of window programming is learning about the different controls that you can put on the screen and how to drive them. This section describes some individual controls. The controls shown in Figure 21-1 are all subclasses of the general class JComponent that we have already seen.

Figure 21-1. Some JComponent controls (visible GUI objects) of Java



[View full size image]

Swing Threads—A Caution!

The GUI components are maintained on the screen by a thread of their own, separate to any threads that you have running in your code. This GUI thread is called the event-dispatching thread, and it takes care of rendering the GUI components and processing any GUI events that take place. An example of a GUI event would be a mouse click, a selection from a menu, or a keystroke on a text field.

The need for thread safety occurs in the system libraries just as much as it does in your code. To work properly, Swing requires that all code that might affect GUI components be executed from the event-dispatching thread! The reason is performance: the GUI library does not have to do synchronization that would be required if multiple threads were potentially adjusting data.

As you know, events are handled by a callback from the window system to an event-handler object you supply. That means that your event-handler code automatically executes in the event-dispatching thread, as it should. But should you try to create new GUI components in another of your threads, your thread will not be synchronized with run-time library data structures. Therefore, the only place where you should create, modify, set visible, set size, or otherwise adjust GUI components is in your event-handler code. Otherwise, you will bring yourself synchronization problems that are difficult to debug.

The following example is erroneous natural-looking code:

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

    createSomeFrame(); // created in the main thread

    showThatFrame();   // shown by main thread,

    // now that first component is displayed, no more

    // code that affects components may be run from

    // any thread other than the event-dispatching thread

    createSomeOtherGUIComponent(); // WRONG!!!

}
```

There is one exception to the rule of doing GUI work only in the event-dispatching thread, which fortunately allows us to write our programs in the most natural way. You are allowed to construct a GUI in the application's main method or the JApplet's init method, providing there are no GUI components already on the screen from your program and that you do not further adjust it from this thread after the GUI becomes visible. Most people obey these rules by accident, but you should know about them. For practical purposes, this means if you want to create a new GUI component in response to a GUI event, you must do the instantiation in the code that handles the GUI event.

Swing Components

JLabel

What it is:

JLabel is the simplest JComponent. It is a string, image, or both that appears onscreen. The contents can be left-, right-, or center-aligned according to an argument to the constructor. The default is left-aligned. JLabel is a cheap, fast way to get a picture or text on the screen.

How it appears onscreen:

Figure 21-2. How JLabel appears onscreen

The code to create it:

```
//  remember, we are only showing relevant statements from main()
     ImageIcon icon = new ImageIcon("star.gif");
     JLabel jl = new JLabel("You are a star", icon, JLabel.CENTER);

     frame.add( jl );
     frame.pack(); // size the JFrame to fit its contents
```

Note the way we can bring in an image from a GIF or JPEG file by constructing an ImageIcon with a pathname to a file. Labels do not generate any events in and of themselves. It is possible, however, to get and set the text of a label. You might do that in response to an event from a different component. The constructors for JLabel include the following:

More About Swing Components

Components and events named all the significant JComponents, and there were about forty in all (including all the subclasses of subclasses). We've briefly presented some of the more important ones here. That's certainly enough to get you started writing GUI programs. To keep the book to a manageable size, however, and still fit in all the other information, we don't show all of them.

Here are some pointers on how to find out more about the other components when you're ready to. The first resource is the good (if somewhat fluid) online tutorial on Java in general, and Swing in particular, that Sun Microsystems maintains at java.sun.com/docs/books/tutorial/ui/swing/.

The second resource is a book that examines JFC, including the Swing components in depth. These books are frequently intimidating in size. One such book that I like is The JFC Swing Tutorial: A Guide to Constructing GUIs, Second Edition by Walrath, Campione, Huml, and Zakhour (Addison-Wesley, 2004, ISBN 0201914670). It weighs in at 800 pages, so be prepared to put in a few evenings and weekends.

Debugging lightweight components

JComponent supports a method to help the implementation team debug the Swing library, but you can use it, too. You can call the following code:

```
RepaintManager.currentManager(yourContainer.getRootPane()).

            setDoubleBufferingEnabled(false);

anyJComponent.setDebugGraphicsOptions( options );
```

The first statement turns off double buffering for the container your component is in. The "options" parameter on the second statement is an int which is 0 to switch debugging off, or contains any of these flags OR'd together:

- DebugGraphics.FLASH_OPTION // flash the component as it is accessed
- DebugGraphics.BUFFER_OPTION // show the offscreen graphics work
-

Further Reading

The first Java website that you visit should be Sun's at java.sun.com. But the Java website that you visit most frequently should be the Java Lobby at www.javalobby.org.

The Java Lobby is an independent group representing Java developers. Led by software entrepreneur Rick Ross, the Lobby has tens of thousands of members. It is a great place for thoughtful discussion and late-breaking news on Java. You can even get advice on coding. The Java Lobby is a great resource and membership is free.

Exercises

1. Review the javadoc-generated description of the javax.swing.SwingUtilities class. Write a program that demonstrates the use of two of the utilities in that class.
2. Add the JEditorPane that can render HTML to some code that can make a socket inquiry in the HTTP protocol (chapter 25 shows the code for this). Create a basic web browser in less than 150 lines of code. You can do this in one evening, and spend the rest of the year adding refinements to it.
3. Review the class java.awt.Robot. It is intended to generate native system input events for test programs. Instead, use it to develop an automatic player for Minesweeper or other favorite game.

Some Light Relief—The Bible Code

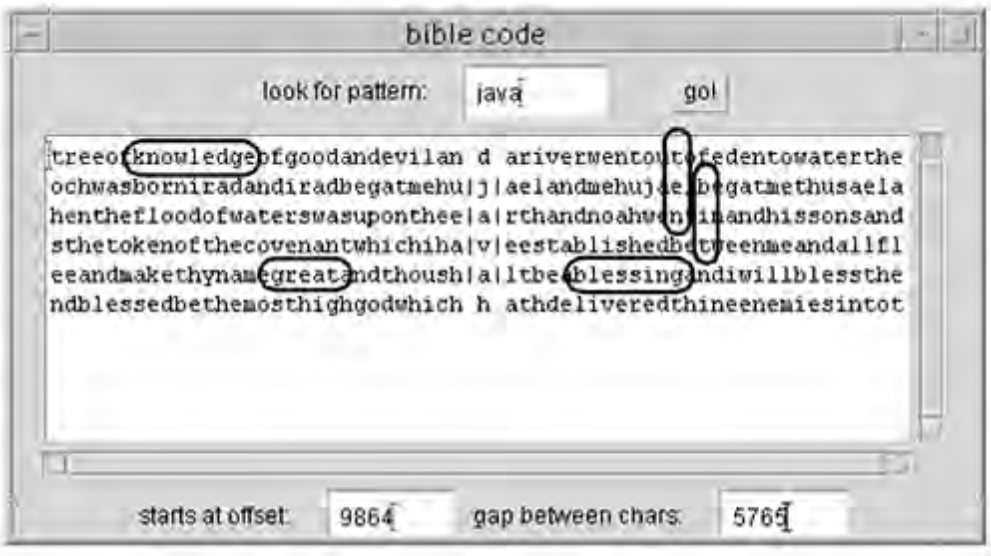
The concept of "Bible codes" was something that became popular in 1997, helped by a mass marketed book on the subject. It's a completely bogus idea that there are hidden strings in the first five books of the Bible, and these hidden strings foretell the future.

The hidden strings, or Bible codes, are allegedly found by looking at individual characters of the Bible, starting at some offset, and taking every Nth letter thereafter to form a phrase. It works much better with a Bible in Hebrew because the classic written form of that language does not have any vowels. Hence, you can construct many possible phrases depending on which vowels you choose to put in and where you choose to end a word. "BLLGTS" can be interpreted as "Boil leg & toes" or "Be a li'l gutsy" or even "Bill Gates."

When you find a Bible "code" you frequently find other related phrases around it. Of course, you can often find clouds in the sky that have shapes that look like animals, and the reason is exactly the same: people tend to see what they want to see. There's a huge amount of sky and clouds to look at, and you can always find something if you look at enough random stuff.

I thought it would be fun to write some Bible code software in Java, so I put it on the book website at afu.com/jj6. There's a program there that you can run to search for arbitrary patterns in the Bible (a copy of the King James Version is also there). See Figure 21-14 for the results when I set it to search for the string "Java", which is a place and language unknown in biblical times.

Figure 21-14. Bible code says "Java a great blessing!"



[View full size image]

As you can see, it has found the word along with other astonishing and highly meaningful phrases ("knowledge of Java, a great blessing, bit, net"). You can run the program for yourself and find other phrases of your choice.

Chapter 22. Containers, Layouts, and AWT Loose Ends

We're now two-thirds of the way through our tour of JFC and Swing. This chapter completes the topic by presenting some containers and an explanation of how you use them to lay out your components neatly on the screen. We also give some information on a couple of topics that are related to the window system in general.

Pluggable Look and Feel

Let's start off the chapter with something that, while not unique to Java, is certainly not widely available. I'm referring to the Pluggable Look and Feel or PLAF as it is usually abbreviated. By default, a Swing program has the Metal look and feel, which is a look and feel designed especially for Java. You can easily change that so a program has the look and feel of Windows, the Macintosh, or of Motif. PLAF means that you, the programmer, can add a few extra lines of code to allow users to select which look and feel they want anytime they run your program, regardless of which operating system is running underneath.

The Pluggable Look and Feel is a win for software portability. Not only do your Java programs run everywhere, but they can even look the same everywhere.

This is a great boon for users who have become comfortable with a program on one particular platform. Now that same knowledge and familiarity can be retained regardless of the execution environment.

As a programmer, you might share my opinion that the look and feel of a window system is not the most important topic in software today. All window systems do roughly the same set of things, and it's no big deal. The fact remains that, for some users, it is a most important topic. Those users buy the software that pays the wages that keep us employed.

Figure 22-1 is an example of the kind of differences you see on different window systems. The three panels show the same controls in three different "look and feel" ways. The top panel is the Metal look and feel. If you choose to have your GUIs display in Metal, your programs will have a consistent look regardless of the operating system or window libraries. The second panel has the Motif look and feel. The lines are so delicate and thin that some of them don't reproduce completely on this screen capture. The bottom panel is the basic Windows look and feel. Pay no attention to the relative sizes of fonts and buttons. That can all be resized to suit the program. The difference is in how much shading appears around a button, how thick the dot is in a selected radio button, whether components are given a 3-D look, and so on.

Figure 22-1. A Pluggable Look and Feel (PLAF)

All About Containers

We come now to the third of the three ideas common to all Java GUI programs: grouping together controls and arranging them neatly by adding them to a container.

We've seen this when we added the JComponents to the content pane of a JFrame, and the JFrame showed up on the screen. The piece that is new is that a container can have different layout policies for where components go on the screen when you add them. A layout policy might be, "Add components from left to right across the container. When you reach the right-hand margin, start a new line of them." Another layout policy might be, "Components can go to the north, south, east, west, or in the center of the component. You have to tell me where you add one." There are a number of classes, called Layout Managers, that implement layout policies like these. We are going to describe them at length in this chapter. Before we do, we'll look at containers a bit more closely.

Controls, containers, component...where will it all end?

Here's the way to tell these three similar-sounding names apart!

Control	This is not a Java term. This is the PC term for what is called a widget in the Unix world. A control is a software element on the screen, such as a button or a scroll bar.
Container	These are screen windows that physically contain groups of controls or other containers. You can move, hide, or show a container and all its contents in one operation. Top-level containers can be displayed on the screen. Non top-level containers have to be in a top-level container to be displayed.
Component	This is a collective name for controls and containers. Since they have some common operations, component is their common parent class. Swing's JComponents are a subclass of component.

The Container class hierarchy

Layout in a Container

Figure 22-5 shows a frame to which we have added several controls. They are positioned automatically as we add them.

Figure 22-5. Arranging controls on the screen

The code for this is on the CD in the directory containing all the other AWT programming material. The problem is the end result doesn't look very professional because nothing is neatly aligned. Solution: layout managers!

Layout Managers are classes that specify how components should be placed in a container. You choose a layout manager for a container with a call similar to the following invoked on the content pane:

```
setLayout( new FlowLayout() );
```

We'll look at six layout managers: the five that are part of AWT and a sixth one that comes with Swing. The first and most basic layout manager is FlowLayout.

FlowLayout

Figure 22-6 uses the same code as previously, but the FlowLayout was used and the JFrame was pulled out wide to the right.

Figure 22-6. In this window, buttons are positioned left to right and centered

Tying up the Loose Ends

At this point, we have dealt with events, components, and containers both in summary and in depth. There are just a few other topics to cover to conclude the chapter.

JDK 1.4 image I/O

Java at last has an API for image I/O, allowing you to read and write files in several popular formats both locally and across the net. It's nicely-designed and extensible so that as new formats appear they can quickly be supported by Java.

Image I/O (introduced in JDK1.4) supports reading GIF, JPEG, and PNG formats. GIF is not supported for writing because it uses LZW compression, the algorithm for which is encumbered by a patent held by Unisys. One patent expired in June 2003, but Unisys has indicated that it thinks it holds some other patents that continue its ownership of GIF.

PNG is "Portable Network Graphics," a newer standard intended to replace GIF, and not encumbered by patents. You can read about the formats (any format, in fact) at www.wotsit.org.

There are five packages in Java Image I/O, but most of them are concerned with the implementation and plugability of new support. You will probably find that the class javax.imageio.ImageIO contains static methods to do all the simple things you need. You can read a JPEG file into an Image with the following two lines:

```
import javax.imageio.*;
```

```
File f = new File("c:\images\myimage.jpg");

BufferedImage bi = ImageIO.read(f);
```

It is equally easy to write out an image in PNG or JPEG format. If you refer back to the previous chapter, we showed code to do screen capture into an object of type java.awt.Image. When you have one of these, check if it is the BufferedImage subclass. You can write BufferedImages to a file with the following lines:

```
BufferedImage bi = (BufferedImage) myImage;
```

Exercises

1. Take the MyFrame program that demonstrates the thread bug and update it to reproduce the race condition more easily. Change the arithmetic that updates variable i into two statements, and put a sleep() or a yield() statement between them. Observe the failure.
2. Update the MyFrame program to remove the race condition by making the reading and writing of the variables mutually exclude each other. Test your code by running it on a version of the program that quickly reproduces the race condition failure.
3. Write a small Java program to capture the screen, display it in a scroll panel inside a frame, and allow the user to trim its size. Write the cropped region to a file on request. Allow the user to choose any output file format that the Image I/O library on that platform supports.
4. Write a program that uses Image I/O to display a thumbnail display of all the GIFs, JPEGs, and PNG files in a directory. Let the user select one, and display that full size in a new window. Allow the user to change the size of it, crop a region, and save it in a new format. That is the beginning of a general-purpose Java image editing application. You can take it as far as your interests run.
5. The following code will grab a screen image.

```
      Robot ro = new Robot();
Toolkit t = Toolkit.getDefaultToolkit();
final Dimension d = t.getScreenSize();
Rectangle re = new Rectangle(d.width, d.height);
final Image image = ro.createScreenCapture( re );
```

   Write code to display it half size, and allow the user to crop the image. Use the class javax.imageio.ImageIO to write the image out as a JPEG file. Refer to the javadoc to see the write() methods.

Some Light Relief—Sky View Cafe: A High Quality Applet

For the end of this chapter, I want to introduce you to the Sky View Cafe program and its author, Kerry Shetline. Sky View Cafe is an astronomy applet, but even if you have no interest in astronomy, it's a terrific showcase for the very professional graphics effects that a careful programmer can achieve.

In case you are interested in astronomy, I'll mention that Sky View Cafe displays many types of astronomical information, and is particularly easy to use. It shows star charts, rise and set times for the Sun, Moon, and planets, Moon phases, orbital paths of the planets in 3-D (I love animating that one!), a perpetual calendar with astronomical events, lunar and solar eclipses, the moons of Jupiter and Saturn, and more. See Figure 22-12 for the main screen.

Figure 22-12. The Sky View Cafe astronomy applet

[View full size image]

That screen shows the half of the world that is in darkness and the half that's in daylight at that moment. It's a funny shape because the world is round and flattened out into an unrolled cylinder on the screen. You can see that the Antarctic is enjoying 24 hours of daylight at the moment.

The first thing to do is to click on the map to tell the program where you are on the planet. That makes the night sky maps accurate for your position. Instead of clicking, you can look up your city name by clicking the Find button on the

Part 5: Enterprise Java

Chapter 23. Relational Databases and SQL

Introduction to Relational Databases

A database is a structured collection of data. It may be anything from a simple list of members of a sports club, to the inventory of a store, or the huge amounts of information in a corporate network. To retrieve and update data stored in a computer database, you need database management software. There are several approaches to database software architecture: relational, hierarchical, and Codasyl network. Here, we'll focus on relational databases, which is the most widely used approach by far.

The relational database concept

Like so many of the best ideas in computer science (e.g., TCP/IP, HTTP, XML, ethernet, sockets, or the JVM), relational databases are based on a simple fundamental concept. The idea behind relational databases is that you can organize your data into tables. Other approaches to databases have tried to keep everything in one big repository. Having individual tables that keep the related pieces of data together simplifies the design and programming. It also adds speed and flexibility to the implementation.

What is a table?

What specifically do we mean by a "table"? We mean the data can be represented in tabular form with columns that all contain values of one type, and where a row holds the related data on one thing (one customer, one account, one order, one product, etc.). The table format is a logical, not a physical, organization. Under the covers, the database management software will typically store the data in indexed files and cache it in memory in tree structures when the program is running. But it will always present the appearance of tables to your programs.

Table 23-1 shows some (fictional) people, their age, favorite bands, and where they live.

Table 23-1. The "People and Music" table

Name	Age	Lives in	Listens to
Robert Bellamy	24	England	Beatles
Robert Bellamy	24	England	Abba
Robert Bellamy	24	England	Oasis
Judith Brown	34	Africa	Muddy Ibe
Judith Brown	34	Africa	Abba
Butch Fad	53	USA	Metallica
Timothy French	24	Africa	Oasis

Primary and Foreign Keys

It's one thing to put data into a database. You also need to be able to retrieve it on demand. The way we get data out of the database is to present it with the right unique identifier, known as a key. For example, in a database of vehicle license records, the VIN or Vehicle Identification Number will be the key to the table of vehicles.

Primary keys

Every table must and will have an attribute (or group of attributes together) that uniquely identifies every record. This attribute or group of attributes is called the primary key to the table. By "uniquely identify every record," we mean that each record has a different value for that attribute or group of attributes. The primary key to our Person table is the "Name" attribute. We can never allow two different people to have the same name in this small database, although that is an unrealistic restriction in real life. That's why banks and other agencies identify you by social security number or an account number, which is guaranteed to be unique.

In our "Listens To" table, we need both attributes (person name and group name) to uniquely identify a row of data. People who like two bands are in there once for each band, so person name is not unique. And since several people can listen to the same band, music group names are duplicated too. But the combination of person name plus music group name is unique. So the primary key to our "Listens To" table is both these attributes.

Foreign keys

As well as primary keys, many tables contain foreign keys. These are attributes in one table that are a primary key in some other table. The "Listens To" table has a foreign key of Name, which is the primary key for Person. The Person table does not contain any foreign keys. Although it has the Name attribute, that is only part of the key for the "Listens To" table, not the whole key. It's called a "foreign" key because it is not a key in this table, but for a table in some other distant place.

When you have a value of a foreign key, for instance, "Judith Brown" in the "Listens To" table, that value must also occur in the table for which it is the primary key. In other words, there must be a "Judith Brown" entry in the Person table. In fact, the purpose of keys is to be able to get to related data in other tables. Keys are how we navigate through the database. When the foreign key existence requirement is met, then the database is said to have referential integrity.

Referential integrity

You keep referential integrity in a database by being careful about the data you remove. If you drop a customer account because of lack of activity, you must also drop all references to that customer in all tables in your database. The onus is on the programmer to keep referential integrity; the database cannot do it for you. If your database lacks referential integrity, you'll get funny results when you try to extract data from more than one table together.

Entity and transaction integrity

There are two other forms of integrity that databases need: entity integrity and database integrity. If you don't have a value for some attribute, perhaps because you are still acquiring data for that table, there is a special value called "null"

Relationships

Let's say more about the relationship between a foreign key and the table where it is a primary key. That relationship can be one-to-one, one-to-many, or many-to-many.

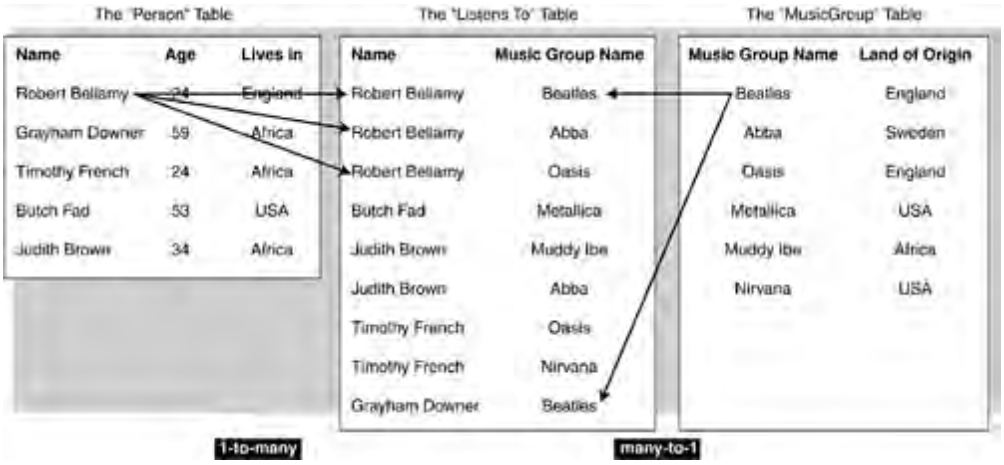
One-to-one relationships

A one-to-one relationship says that the two things are matched exactly.The relationship between ship and captain is one-to-one. Each ship has one captain, and each captain has one ship (ignoring real world details like mutinous ships, captains waiting for a command, etc.).

One-to-many relationships

As you might guess, a one-to-many relationship means that one thing in this table corresponds to potentially many things in that table. Each individual ship has multiple sailors, so the ship/crew member relationship is one-to-many. All the sailors on a given ship will have the same value for the "belongs to the crew of" attribute. The "one" side of a one-to-many relationship will be a primary key, as shown in Figure 23-1. The ship's name would be a primary key in the table of a shipping line's fleet.

Figure 23-1. Breaking up a many-to-many relationship: "Many people listen to many bands"



[View full size image]

Many-to-many relationships

Many-to-many relationships occur when multiple records in one table are somehow related to multiple records in another table. We can see what this means if we introduce a "MusicGroup" table that is a list of bands. We could store any band-specific information in it too, such as the land of origin for each music group. Take, for example, Table 23-4.

Table 23-4. The "MusicGroup" table

Normal Forms

Tables need to follow a set of numbered rules known as "normal form." (It's "normal" in the sense of "normalized" or conforming to a standard). First normal form says that all attributes must be atomic. That means there can be no lists of items in an attribute. You can't have an attribute that is "contents of a shopping cart," because that could contain several items. Atomic means "only one, and it cannot be subdivided any further."

Second normal norm

Second normal form says that it is in first normal form (atomic) and every non-key attribute depends fully on the key. The Person table has non-key attributes of "age" and "lives in" and both of these are completely dependent on the person we are identifying by the "name" attribute. However, if we added a column to the table to hold, say, the land of origin of the band, we would be breaking second normal form. A band's land does not depend on the primary key (the person name).

Third normal norm

A table is in third normal form if it is in second normal form and all non-key columns are mutually independent. In the Person table, the non-key columns are "age" and "lives in." These are mutually independent because they can change without affecting the other. If we were to add a column to store "is a minor" data in Person, the table would no longer be in third normal form. Whether or not someone is a minor depends on their age, which is another attribute in the Person relation.

Add tables to taste

If you find your table designs break a normal form rule, you can always fix it by adding an extra table as we did above to resolve a many-to-many relationship. You may need to move columns from one table to another or to add an identification number to a couple of tables to relate records between them. Raw data can always be put into third normal form. There are additional normal forms beyond this, but third normal form is enough for most purposes. If you make sure all your tables are in third normal form, you will be able to use relational database operations on them and get the right results. A database that has been designed to be in third normal form is said to be "normalized." There's a great memory aid for the form in which you want your tables: the data in a table has to depend on the key, the whole key, and nothing but the key.

Relational Database Glossary

Table 23-5 is a glossary of terms that you can review and refer to as necessary. There are a few terms in here that appear later in the chapter.

Table 23-5. Relational database glossary

Name	Description
attribute	A column in a table, e.g., the "Name" attribute.
cardinality	The number of rows in a relation.
database integrity	You maintain database integrity by grouping statements in a "transaction" that is either executed as a whole, or has no part of it executed.
degree	The number of columns in a table.
domain	The set of permissible values for an attribute.
entity integrity	The requirement that key attributes never contain a null.
first normal form	A table in which no attributes have lists of data items.
foreign key	The attribute or group of attributes in one table that form a primary key for some other table.
join	An operation that combines the data in two or more tables by using foreign keys in the first table to access related data in a subsequent table. This is an "inner join" or "equijoin".
prepared statement	An SQL statement which is cached in native code form to allow faster processing.
primary key	The attribute or group of attributes that together uniquely identify a record in a table.
referential integrity	The requirement that all foreign keys are present in the table where they are a primary key.
relation	A table in a relational database. It corresponds to a file of

Download and Install Mckoi

At this point you should download and install the Mckoi database software, so you can try running the SQL queries, and get the results visually.

To get started, go to the Mckoi website at

```
www.mckoi.com/database
```

Click on the "latest version" link under "Download the software." Download the zip file to your C:\ top level directory. It is about 2MB in size, so it downloads quite quickly. After the mckoi zip file is on your disk, unpack its contents using a command like this:

```
cd c:\

jar -xvf mckoi1.0.2.zip
```

That creates a directory called mckoi1.0.2 containing the database binaries and Java source, some documentation, and sample programs.

Set up the Mckoi jar files

Next, make the Mckoi libraries visible to your Java compiler and JVM. There are three jar files in the directory where you just extracted the release. These jar files are:

file name	contents
mckoidb.jar	The database management software
gnu-regexp-1.1.4.jar	GNU regular expression package
mkjdbc.jar	The JDBC driver software

Basic SQL Primer

At first, every database vendor had its own special database query language. Users eventually got fed up enough to create an industry standard around IBM's SQL. There was the SQL'89 standard, followed by the SQL'92 standard, both created under the umbrella of ANSI (American National Standards Institute). SQL version 3 was published in 1999, and is known as "SQL:1999" or SQL-3. SQL is also a FIPS standard, FIPS PUB 127-2. FIPS is a Federal Information Processing Standard issued with the full weight and authority of the U.S. Government, after approval by the Secretary of Commerce. In practice, SQL is fragmented with many slightly incompatible dialects from database vendors. We keep to the current ANSI standard and do not present any vendor-specific code. You should follow the same practice in your programs, too.

SQL is an abbreviation for "Structured Query Language" and is a programming language in its own right. It's usually pronounced like the word "sequel" or spelled out as individual letters s-q-l. SQL is specialized for its application area, and is not used for general purpose programming. But all of the operations that you are likely to want to do to a database are built-in functions in SQL.

One attractive feature of SQL is that you express what you want to do in English-like text such as this.

```
SELECT name FROM Person

     WHERE lives_in = 'Africa'

     ORDER BY name;
```

By convention, the SQL keywords are written in uppercase. In SQL, you describe the results you want, not the steps to carry out to get them. This style of programming is known as "functional programming" and it contrasts with the "procedural programming" of more familiar languages like Java. Because you don't give the steps to get what you want, database implementors are free to find the most efficient way to get it. The big database companies put a lot of effort into their query optimizers, and this is one of the big advantages of SQL over earlier query languages.

The designers of SQL could have chosen to make programmers express the operations in terms of mathematical formulas or algebra, instead of words. That would make programs harder to read for many people and raise an unnecessary barrier to learning and teaching. Thank heavens they shunned that temptation. Executing the above SQL statement on our Person table yields a result set of:

Grayham Downer

Judith Brown

Creating and Populating Tables

The CREATE statement is used to create a new table, and the INSERT statement is used to add a new record to a table.

The CREATE statement has this general format:

Format of SQL CREATE statement	Additional information
CREATE TABLE tablename(colName dataType <— can repeat this line, separated by commas optionalConstraint);	

Here is an example of the use of the CREATE statement:

Example of SQL CREATE statement	Additional information
CREATE TABLE Person (name VARCHAR(100) PRIMARY KEY,	"PRIMARY KEY" is a constraint
age INTEGER,	
lives_in VARCHAR(100));	

This statement will create the Person table that we saw earlier in the chapter. It will have three columns called "name," "age," and "lives in." The "optionalConstraint" shown in the first example means that you can add or omit a constraint to a column, giving more information about what kind of values are legal there. We have added a constraint to the "name" column, saying that this is the primary key of the table. That has the effect of making sure that records always have a non-null unique value there when the records are inserted into the table. "Not null" and "unique" are also constraints that can be applied individually.

Try typing the SQL statement into the visual query tool and confirm you get these results. There is already a table called Person in this database, so you will get an error message to that effect. Enter the query again, changing the table name to Person2. You need to completely overwrite the old query with the new one, when using the Mckoi GUI tool.

Some datatypes that SQL understands and the corresponding Java types are shown in Table 23-6. The SQL keywords and datatypes can use any letter case. The convention is to write them all in uppercase.

Table 23-6. Some SQL datatypes

Querying and Retrieving Data

The SELECT statement is used to query a database and get back the data that matches your query.

The SELECT statement has this general format:

Format of SQL SELECT statement	**Additional information**
SELECT	
Name1 ,Name2 ,Name3 ...	<— can mention one or more columns, or "*" for all columns
FROM	
tablename1, tableName2, ...	<— can mention one or more tables
WHERE	
conditions	<— the "WHERE" clause is optional and can be omitted
ORDER BY colNames	It returns the data sorted by this field
;	<— the "ORDER BY" clause is optional and can be omitted

We have already seen an example of a basic select from a single table. The power of the statement arises when you select from two or more tables at once. So to find all the people in Africa in our database who listen to the Beatles or the band Fela Kuti, we could use the SQL command shown below. Numbers have been added on the left to help with commenting on the code; these will not appear in actual SQL code.

```
1    SELECT Person.name, Person.lives_in, ListensTo.music_group_name

2    FROM Person, ListensTo

3    WHERE ListensTo.music_group_name IN ( 'Fela Kuti', 'Beatles' )

4    AND Person.name = ListensTo.person_name

5    AND Person.lives_in = 'Africa' ;
```

Going through the statement line by line, we can make the following observations:

Line 1 gives the columns that we want to get back in our answer. Notice that the table name can be used to qualify the column so that there is no ambiguity.

Subquery Selections

Quite frequently you want to submit a further select on the result of a select. There are several ways to do that, one way being to nest a select statement inside another. A nested select statement is called a subquery.

Here is an example of a subquery:

```
SELECT Person.name FROM Person

WHERE

   Person.lives_in IN ('England', 'USA')

AND

   Person.name NOT IN

     ( SELECT ListensTo.person_name FROM ListensTo

       WHERE

          ListensTo.music_group_name = 'Beatles' );
```

The simplest way to understand subqueries is to look at them piece by piece, starting from the innermost nested one. In this case, the nested select statement is:

```
( SELECT ListensTo.name FROM ListensTo

  WHERE

     ListensTo.music_group_name = 'Beatles' );
```

A moment's reading should convince you that this provides a result set of names of people who listen to the Beatles. So substitute that into the entire statement, and we get:

Result Set of a SELECT Query

We've seen informally in previous examples how the results of a SELECT statement are returned to you. The results of a query come back as rows-in-a-table, held in an object called a result set.

Contents of a result set

The result set contains zero or more rows which are retrieved and examined individually using something called a cursor. Just as a GUI cursor marks your position on the screen, a database cursor indicates the row of the result set that you are currently looking at. A cursor is usually implemented as an unsigned integer that holds the offset into the file containing your result set. It has enough knowledge to move forward row by row through the result set.

The cursor

Database management systems typically provide a cursor to the SQL programmer automatically. The programmer can use it to iterate through the result set. JDBC 2 upgrades the features of a cursor available to Java. Now you can move the cursor backward as well as forward, providing the underlying database supports that. You can also move the cursor to an absolute position (e.g., the fifth row in the result set) or to a position relative to where it is now (e.g., go to the immediate previous record).

Getting a sorted result set

We can ask for our result set to come to us sorted by some column or columns. We achieve this by using the "ORDER BY" clause in the query. When you do that, it makes sense to use the cursor to ask for the record before the one we are currently looking at. For example, if you order by "billing price" you can go backward until you reach orders under $10. That way, you can process your most valuable orders first, and stop invoicing when the amount is smaller than the cost of processing.

SELECT pitfalls

Here are some common pitfalls encountered when using the SELECT statement. When you hit an error in your programs in the next chapter, check if it is one of these!

- Not surrounding a literal string in single quotes.
- Spelling the table name or the column name wrongly.
- Only mentioning the tables that you are extracting from in the "from" clause. You need to mention all the tables that you will be looking at in the "where" clause.
- Failing to specify "distinct" and thus getting duplicate values in certain columns.

Updating Values

The UPDATE statement is used to change the values in an existing record.

The UPDATE statement has this general format:

Format of SQL UPDATE statement	**Additional information**
UPDATE tablename SET colName1=value1 ,colName2=value2 ... WHERE colNamei someOperator valuei ... ;	<— can provide a value for all attributes or just some <— can repeat this line, separated by AND or OR

Here is an example of the use of the UPDATE statement:

Example of SQL UPDATE statement	**Additional information**
UPDATE Person SET age = 25, lives_in = 'USA' WHERE name='Robert Bellamy' ;	Robert celebrated his birthday by moving to the USA.

This statement will start to populate (fill in with data) the Person table that we saw earlier in the chapter. The values are inserted into the record in the order in which the attributes are named.

Deleting Records and Tables

The DELETE statement is used to remove records from a table, and the DROP statement is used to completely remove all trace of a table from the database.

The DELETE statement has this general format:

Format of SQL DELETE statement	Additional information
DELETE FROM tablename WHERE colName someOperator value ... <— can repeat this line, separated by ; AND/OR to further refine which records get deleted	

If you forget the "where" clause, all records in the table are deleted! A table with no records still exists in the database, and can be queried, updated, etc. To get rid of all traces of a table (not a common operation in most databases), use the DROP statement.

The DROP statement has this general format.

Format of SQL DROP statement	**Additional information**
DROP TABLE tablename ;	

There is frequently more than one way to write an SQL query. Some of the ways will do less work than other ways. Nowadays it is the database's responsibility to reorder queries for the best performance.

SQL Prepared Statements and Stored Procedures

Prepared statements and stored procedures are two different ways of organizing your SQL code and getting it to run faster. When you send an SQL statement to your database, there is an SQL interpreter that reads the statement, figures out what it means and which database files are involved, and then issues the lower-level native instructions to carry it out. Depending on what the statement is exactly, it may be quite a lot of work to analyze and interpret it.

Prepared statement

If you find that you are issuing a statement over and over again, the database will be doing a lot of work that can be avoided. The way to do this is with a prepared statement. As the name implies, the prepared statement is constructed and sent to the SQL interpreter. The output of the interpreter (the native code instructions) is then saved. The prepared statement can later be reissued, perhaps with different parameters, and it will run much more quickly because the interpretation step has already been done. Does this remind you of anything? This is exactly how Just-In-Time (JIT) Java compilers speed up execution—by compiling to native code and caching the results.

Stored procedure

A stored procedure is a similar idea to prepared statements but taken one step further. Instead of caching an individual statement, you can save a whole series of statements as a procedure. A stored procedure will typically implement one entire operation on a database, like adding an employee to all the relevant tables (payroll, department, benefits, social club, etc.). It is typical to provide parameters to a stored procedure; for example, giving the details of the employee who is being added to the company.

The vast majority of database systems support stored procedures, but a major sticking point has been the variation in the exact syntax used. JDBC 2 solves this issue by allowing you to write stored procedures in Java. That means your library of stored procedures is now portable to all databases, which is a major step forward!

This concludes our tour of the concepts of SQL and databases, and we now proceed to Chapter 24 to look at how you put it all together in Java.

Exercises

1. Define and give examples of the following database terms: tuple, attribute, relation.
2. What does it mean to normalize a database design? Describe first, second, and third normal forms.
3. Reinforce your knowledge by reviewing the basic SQL course at www.sqlcourse.com. It allows you to formulate and run SQL queries online.
4. Write an SQL statement to display the name and age of everyone in the Person table who is older than 39.
5. Write an SQL statement to display the name of everyone in the Person table who lives in a NATO country and listens to the Beatles. There are 19 member nations of the North Atlantic Treaty Organization, including the USA, UK, Canada, France, Germany, Greece, and Poland. New members join from time to time, so top students will keep the list in a table, rather than a set of literals.
6. What is an SQL subquery, and when would you use one?
7. Write an SQL statement to display the name of everyone in the Person table who lives in a NATO country and does not listen to the Beatles. Be careful to exclude people who listen to the Beatles, and also listen to other bands as well. The simplest way to do this is to use a subquery.
8. Explain, using examples, the difference between a primary key and a foreign key.

Some Light Relief—Reading the Docs

How do you tell if a user has read the software documentation? If users are anything like us programmers, it's a pretty safe assumption that they have not read the documentation. Time is short, and reading manuals is tedious and time-consuming.

I once knew someone who worked on the support desk for a large internal software application. He cut his workload by 85% using one simple technique. Whenever someone reported a problem with the software, he asked them which page of the manual it violated before he would investigate it. Most users preferred to live with any bug rather than spend hours tunneling through the manual, and Perkins' technique saved him a lot of bother right up until the time he got fired.

Another way of encouraging people to read the manual is to have the program ask "Did you read the manual, answer y/n:" The program won't proceed until it gets the right answer. And neither "yes" nor "no" is the right answer. Somewhere in the manual, buried deep in an obscure paragraph, is the information that, to continue, this question expects the answer "Teletubbies." But you'll only know that if you read the manual thoroughly.

Are you a student reading this chapter for an "Advanced Java" class? OK, then! Please demonstrate that you have read this chapter by writing your favorite color at the top right of the front sheet of your homework for this chapter. If blue is your favorite color, write "blue." Write "black" if you like black best, etc. Professors: see how many of your students really do the assigned reading. But be careful—last time I did this, one of the students wrote a little note buried in his next homework setting me a similar challenge, to test if I really read each page of all the homework that was submitted!

Chapter 24. JDBC

In this chapter we'll build on the relational database and SQL knowledge from Chapter 23. We'll show how to use one of the several excellent open source Java-friendly relational databases available. This will let you run a database management system on your own computer and try the features in practice. The bulk of the chapter to describes JDBC, the Java library that supports access to databases. We'll walk through its classes and the way they are used. We will reuse the data from the previous chapter, involving a database holding the music preferences for a group of people. Finally, we'll show code to create and update a database, and give you the information needed to write more

Introduction to JDBC

JDBC is made up of about two dozen Java classes in the package java.sql. The classes provide access to relational data stored in a database or other table-oriented form. JDBC works in a similar way to Microsoft's database access library (known as ODBC), but redesigned, simplified, and based on Java, not C. ODBC imposed a single library that let your Windows code interface to any database. If you are familiar with ODBC, JDBC will be a snap to learn. And even if you are not, it's still pretty straightforward. JDBC works with the largest database servers such as Oracle, DB2 and mySQL, and with the smallest desktop database systems, such as xBase files, FoxPro, and MS Access. JDBC can even access text files and Excel spreadsheets using the ODBC bridge.

Building database systems in Java

An exciting development (new in 2004) is Sun's product Java Studio Creator. It is a development environment that has all the forms, GUIs, and visual tools to make it very easy to put together a Java client/server/database system. You can get more details on Java Studio Creator at http://developers.sun.com/prodtech/javatools/jscreator/index.jsp

There are white papers and tutorials, as well as a free download of the tool itself.

JDBC classes allow the programmer to use modern database features like simultaneous connections to several databases, transaction management, pre-compiled statements with bind variables, calls to stored procedures, and access to metadata in the database dictionary. JDBC supports both static and dynamic SQL (a query or update constructed at run-time). JDBC and SQL greatly simplify deployment issues, because you can now rely on the presence of a set of vendor-independent standard Java interfaces for queries and updates to your relational database.

Installing the Mckoi Database Software

A major goal of this chapter is to give readers the means to actually try some hands-on relational database programming. That's an ambitious goal, because relational databases are industrial-strength and industrial-sized pieces of software. Up until a few years ago, the only choice in a database management system was which of the commercial vendors would you buy from. More recently, the explosion of interest in open source software has led to a much larger number of choices, some of which require no financial outlay. Table 24-1 and Table 24-2 show some popular commercial and non-commercial products and their characteristics.

Table 24-1. Some commercial databases

Company	Product	Product attributes	Website	Java support
Oracle	Oracle 9i family	Supports even very large datasets, and also effective for small businesses. Available for Solaris and Windows.	www.oracle.com	Full support
IBM/Informix	DB2, Informix	Large capacity, multi-platform database.	www.ibm.com	Full support
Sybase	Adaptive Server IQ	Large capacity, multi-platform database.	www.sybase.com	Full support
Microsoft	SQL server	Runs only on the NT line, limited by the capacity of the underlying PC.	www.microsoft.com	No vendor support
IBM/Informix	Cloudscape	A commercially supported pure Java database that is included with Java 2 Enterprise Edition. It is the reference implementation of an embedded Java database.	www.informix.com/cloudscape	Full support

Table 24-2. Some non-commercial databases

Organization	Software	Product attributes	Website	Java support

Running the Example Code

The next step is to try running one of the example database programs that accompany the release. Go to the demo directory with this command:

```
cd c:\mckoi1.0.2\demo\simple
```

Then run the sample database application that comes with the release. Use this command (assuming you have put the mckoi jar file in the jre\lib\ext directory):

```
java  SimpleApplicationDemo
```

If all is well, you will see some sample output similar to this, assuring you that the database libraries have been properly installed.

```
Rows in 'Person' table: 12

Average age of people:  30.0833333333

All people that live in Africa:

  Grayham Downer

  Judith Brown

  Timothy French

   ...
```

If you do not see output like this, you will need to debug the problem based on the output you do see. After you have the database example running, go to Connecting to the Database on page 615,to see how your Java application code establishes a connection with a database prior to sending across various SQL commands. We will finish up this section by saying a few words about the evolution of the JDBC.

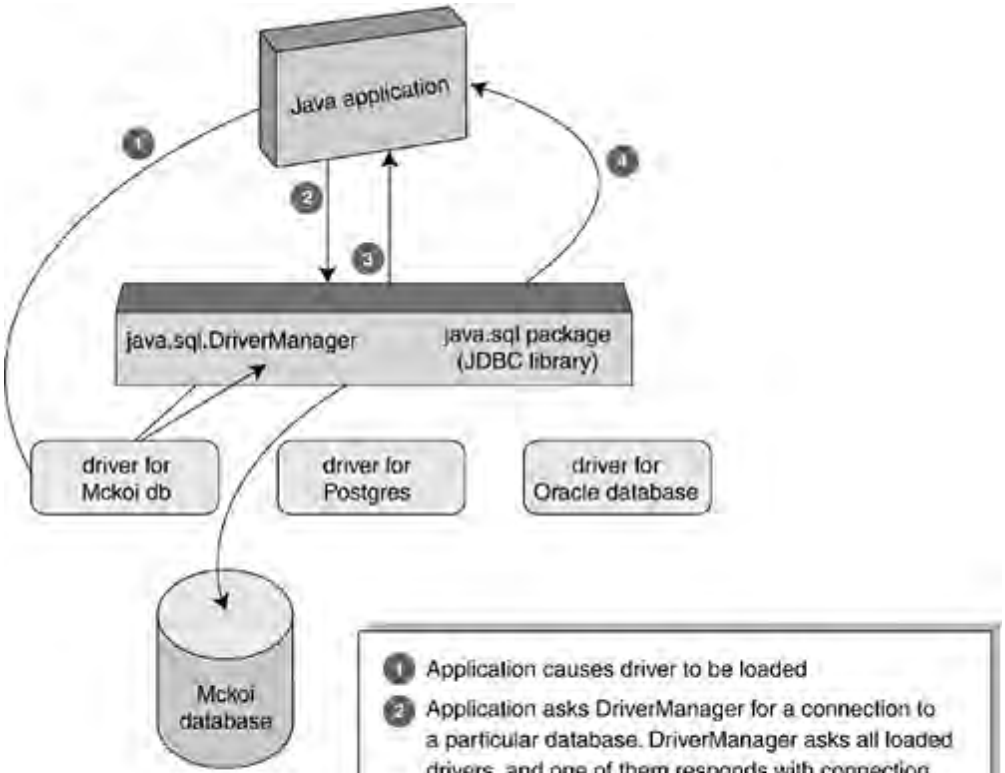
Connecting to the Database

A database works in the classic client/server way. There is one database and many clients talk to it. (Larger enterprises may have multiple databases, but these can be considered independently for our purposes.) The clients are typically remote systems communicating over TCP/IP networks. They may talk directly to the database (called a "2-tier" system) or to a business logic server that talks to the database (known as a "3-tier" system).

How does a client or business logic program open a dialog with a database manager? JDBC uses a piece of software called a database driver. The database driver is specific to each vendor, and it is a library-level example of the Adaptor design pattern. It knows how to connect to its database, send requests over TCP/IP, and how to listen for replies from the database and pass them on to your code. Just as an operating system device driver hides the peculiarities of an I/O device from the kernel and presents a standard interface for system calls, each JDBC driver hides the vagaries of its particular database and presents a standard interface to Java programs that use JDBC.

Putting it another way, the purpose of a JDBC database driver is to know the low-level protocol for talking with its database at one end, and with JDBC classes and methods at the other end. It acts like a human language interpreter, moving information from one end and putting it in a standard form that is comprehensible to the other end (see Figure 24-1). You typically get a JDBC database driver from the database vendor. There are several different kinds of database drivers, depending on whether it is written in Java or native code, or whether it talks directly to the database or through another data access protocol such as Microsoft's ODBC. None of that matters much to the applications programmer. As long as you have a working JDBC driver, you don't care how it works. Essentially, all commercial and non-commercial databases now have excellent support for access from Java programs. There are good third-party libraries that can be used to access the Microsoft database products.

Figure 24-1. How JDBC establishes a connection between your code and a database



Executing SQL Statements

Now we are at the point where we can start issuing SQL commands to our database and getting back results. We do this through a Statement object that we get from the connection object described in the previous section. Table 24-4 shows several methods in Connection.

Table 24-4. Some methods of java.sql.Connection

Method	Purpose
Statement createStatement()	Returns a statement object that is used to send SQL to the database.
PreparedStatement prepareStatement(String sql)	Returns an object that can be used for sending parameterized SQL statements.
CallableStatement prepareCall(String sql)	Returns an object that can be used for calling stored procedures.
DataBaseMetaData getMetaData()	Gets an object that supplies database configuration information.
boolean isClosed()	Reports whether the database is currently open or not.
void setReadonly(boolean yn)	Restores/removes read-only mode, allowing certain database optimizations.
void commit()	Makes all changes permanent since the previous commit/rollback.
void rollback()	Undoes and discards all changes done since the previous commit/rollback.
void setAutoCommit(boolean yn)	Restores/removes auto-commit mode, which does an automatic commit after each statement.
void close()	Closes the connection and releases the JDBC resources for it.

You will invoke these methods on the java.sql.Connection object that you get back from the JDBC driver manager, as shown in an upcoming example. You use a connection to create a Statement object. The statement object has methods that let you send SQL to the database. Thankfully, statements are blissfully simple. You send SQL queries as Strings. In other words, the JDBC designers did not try to force-fit object-oriented programming onto SQL, perhaps by creating a Select class. Here's how you send a select query to the database:

Result Sets

As we saw in the previous chapter, the SELECT statement extracts data from a database. Here's an example which should be prefaced with the warning that columns are numbered starting with 1, not zero. That is an SQL convention that really had to be respected by Java. If we run this Java code fragment,

```
ResultSet result;

result = statement.executeQuery( " SELECT Person.name, Person.age "

                                 + "FROM Person "

                                 + "WHERE Person.age = 24 " );

while (result.next()) {

    String p = result.getString(1);

    int a = result.getInt(2);


    System.out.println( p + " is " + a + " years");

}
```

we'll get output like this:

```
Robert Bellamy is 24 years

Timothy French is 24 years

Elizabeth Kramer is 24 years
```

Relating that output to the code fragment shows how the ResultSet object can hold multiple values. I like to think of ResultSet as being similar to a 2D array. Instead of incrementing the most significant index variable, you call the result method next(). Each time you call next(), you are moved on to the next record in the result set. You need to call next() before you can see the first result record, and it returns false when there are no more result records, so it is convenient for controlling a while loop. That does make it different from an Iterator, however, so be alert to that difference. As a reminder, the Iterator next() method returns the next object, not a true/false value. A true/false value can be returned for a result set next() because there is another group of methods for actually getting the data. Read on to find out what!

Batching SQL Statements and Transactions

Performance has always been one of the top concerns of database vendors, and they often go to some lengths to find ways to speed up queries. One of the bottlenecks is the time taken to package up a query, ship it over TCP/IP, and get it into the database where the SQL interpreter can start working on it. In other words, the network latency has a cost.

To reduce the overhead of network latency, many vendors support a way to batch several SQL statements together and send them to the database as a group. You can batch together any statements that have an int return type, which basically means "any SQL statements except for select." You can see why. You are sending over a group of SQL statements to be executed together, but there is no mechanism defined for getting back the result set for each select. It is not that hard to invent such a mechanism (e.g., executing a batch returns an array of ResultSet), but this has not been done.

To bundle a group of SQL statements in a batch, you create a Statement object as usual:

```
Statement myStmt = conn.createStatement();
```

Then, instead of issuing an execute call for the statement, you instead do a series of addBatch(), like this:

```
myStmt.addBatch( myNonSelectSQL0 );
myStmt.addBatch( myNonSelectSQL1 );
myStmt.addBatch( myNonSelectSQL2 );
```

Finally, when you are ready to send the whole batch to the database, invoke the executeBatch() method:

```
int [] res = myStmt.executeBatch();
```

Batching SQL statements is so easy, there's no reason to avoid it. That will cause all the statements to be sent to the

Prepared Statements and Stored Procedures

Another way to boost performance is to precompile the SQL statement using what is termed a "prepared statement." That technique and the related one of "stored procedures" are described in this section.

Prepared statements

A SQL statement is precompiled and stored in a PreparedStatement object. This object can then be used to efficiently execute this statement repeatedly, often changing some of the argument values at run-time. You get a PreparedStatement using a method of your Connection object. It's easiest to see with a code example:

```
PreparedStatement pstmt = conn.prepareStatement(
    "UPDATE EMPLOYEES SET SALARY = ? WHERE ID = ?");

pstmt.setBigDecimal(1, 150000.00);
pstmt.setInt(2, linden4303);
pstmt.executeUpdate();
 // other code goes here
pstmt.setBigDecimal(1, 85000.00);
pstmt.setInt(2, jenkins2705);
pstmt.executeUpdate();
```

That code will set employee linden4303's salary to $150,000, and employee jenkins2705's salary to $85,000. The question marks in the SQL query represent data values that will be filled in before the statement is executed. It works like arguments to a procedure, with one difference: any of the question mark fields that you don't change will retain whatever value you have previously set them to, so you only need to set fields that change.

PreparedStatement has its own versions of the methods executeQuery(), executeUpdate(), and execute(). In particular, PreparedStatement objects do not take an SQL string as a parameter because they already contain the precompiled SQL statement you previously created.

Stored procedures

Let's move on to take a look at stored procedures. These are a group of SQL statements bundled together as one unit that can be called from your program. That's where the "procedure" part of the name comes from. The "stored" part of

Complete Example

This section shows the complete program to create, update, and select from a database using JDBC. A longer version of this code comes with the Mckoi database and can be found in directory c:\mckoi\demo\simple. The code has been split into two programs there for convenience, one to create the tables, and one to query them.

```
/**
 * Demonstrates how to use JDBC.
 */

import java.sql.*;

public class Example {

  public static void main(String[] args) {

    // Register the Mckoi JDBC Driver
    try {
      Class.forName("com.mckoi.JDBCDriver");
    }
    catch (Exception e) {
      System.out.println("Can't load JDBC Driver. " +
                         "Make sure classpath is correct");
      return;
    }

    // This URL specifies we are creating a local database. The
    // config file for the database is found at './ExampleDB.conf'
    // The 'create=true' argument means we want to create the database.
    // If the database already exists, it can not be created.
```

Database and Result Set Metadata

"Meta-anything" is a higher or second-order version of the anything. Metadata is data about data. The classic example of metadata is file and directory information on your disk drive. You don't directly put it there, but you need it to keep track of your real data, and it is maintained by the system on your behalf. Databases have a large amount of metadata describing their particular capabilities and configuration.

The database metadata is going to be different for each database, and JDBC lets you get hold of it through the java.sql.DatabaseMetaData interface. You get an instance of the Metadata class by invoking a method of Connection. There you will find 100 or so fields and methods that you can use to find out specific details on the database. For example, it can tell you if the database supports transactions, and if so, to what level.

You use the database metadata when you know your code is going to run against several different databases. By looking at the metadata, your code can discover the individual features of a database, and perhaps take advantage of performance-related options. Often, but not always, there is a slower more standard way to achieve an effect, and you may prefer to write your database application code that uses that, instead of querying the database about its advanced features. Using database metadata is an advanced technique, beyond the scope of this book. The javadoc documentation is extensive if you want to pursue this topic further.

Result sets also have metadata. An object of type java.sql.ResultSetMetaData can get information about the columns in a ResultSet object. Here is an example. The following code fragment creates a ResultSet and gets the corresponding ResultSetMetaData object from it. The code then uses that object to find out two pieces of information about the result set. It calls two methods, one to find out how many columns the result has, and one to learn whether the first column in the result set can be used in a WHERE clause (i.e., it is a "searchable" column).

```
ResultSet result = statement.executeQuery(

   "SELECT c1, c2 FROM myTable; "


ResultSetMetaData rsmd = result.getMetaData();

int numberCols = rsmd.getColumnCount();

boolean b = rsmd.isSearchable(1);
```

Further Reading

There are some excellent book-length treatments of relational databases and JDBC in particular. One book I like is the JDBC API Tutorial and Reference (Addison Wesley, Reading: MA, 1999), in successive editions by Graham Hamilton and Rick Cattell, and then by Maydene Fisher, and then by Seth White and Mark Hapner, and finally by Maydene and Seth again. If you buy this book, be sure to get the most up-to-date edition!

In addition, Sun has an online tutorial on JDBC that contains some of the same material in the JDBC API Tutorial book.

java.sun.com/docs/books/tutorial/jdbc/index.html.

Exercises

1. Run the javadoc tool to create the javadoc files for the packages of the Mckoi database, and browse the API. The database comes with the Java source code that implements it. The file is called src.zip. Unzip it, cd to the src directory that it creates, work out what the package names are (they mirror the directory names), and run javadoc on them. Look at some of the source code with an editor, and browse the javadoc-generated API documentation for the same files. How useful is javadoc to you? Why? How far does the code follow the Sun recommended code conventions at java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html ?
2. Write a JDBC program to display the name and age of everyone in the Person table who is older than 39. This question builds on a similar one in Chapter 23 that asked you to write the SQL statement. Now the exercise asks that you put it into a JDBC program and actually run it.
3. Write a JDBC program to display the name of everyone in the Person table who lives in a NATO country and doesn't listen to the Beatles. You can google to find out which nations belong to NATO. Be careful to exclude people who listen to other bands as well as the Beatles. You will need a subquery for this.This question builds on a similar one in the previous chapter that asked you to write the SQL statement. Now the exercise asks that you put it into a JDBC program and actually run it.
4. Modify your program from the previous question to submit an invalid SQL query. How do the database and your program respond?
5. Write the JDBC code to create and populate a table for the CD inventory of an online store. Each CD is either domestic or imported. These details are stored for all CDs: artist, title, price, and quantity in stock. Imported CDs also have these fields: country of origin, genre, non-discount status, language, and lead time for reorder. Write some instance data describing your five favorite CDs (include a couple of imported CDs, too), and populate your database.
6. Update your code from the previous exercise question to allow it to work interactively with the user. The user should be able to type in the title of a CD, and the database should return all the data it holds on that CD.

Heavy Light Relief—In Which "I" Spam Myself

It's official: my email is out of control. Actually, everybody's email is out of control. I have 50 MB in my inbox alone, some of this unanswered email dating back three or more years. There's another few hundred MB in other folders. This is why Google Mail or something like it will be successful. I need a halfway decent tool to search my email and other files of concentrated knowledge and keep them secure, searchable, encrypted and archived. Find and grep just aren't good enough.

Managing files is worth paying for

I don't want Google's (or anyone else's) scripts reading my email, so I am prepared to pay for this service, but no one offers it yet. In the USA, the Stored Communications Act (part of the decades-old wiretap legislation package known as the Electronic Communications Privacy Act) says that any electronic data stored with a third party for more than 180 days can be subpoenaed by law enforcement without notifying the owner of that data. So to discourage government fishing expeditions, the data needs to be stored encrypted on the server and only be decryptable by the client. Some companies, like Hushmail.com offer some of these services. [Note to venture capitalists: I can fix this for $10M, business plan on request. Just don't make your request by email.]

Spam, spam, spam, lovely spam

Spam isn't clogging up my mailbox. That's a separate problem. I get a lot of spam, currently more than 2,500 spam messages each and every day just like everyone else who posted to usenet in the 1990s and still uses the same account. With an average spam size around 10KB, that's more than 2 MB a day flowing in, and being automatically filtered out and junked. The signal-to-noise ratio is 0.004 and dropping by the month. Yet nobody seems able to fix the spam onslaught.

I remember the very first spam email I got. I was slightly surprised to get email from someone I didn't know, who seemed to be suggesting a product for me to buy. Why would they do that, and why would they choose me, out of all the thousands of people on the Internet? This was a couple of years before April 1994 when two deadbeat lawyers from Arizona, Canter and Siegel, spammed all 6000 Usenet newsgroups with their unwanted adverts. They offered to help people enter the Green Card immigration lottery for $145, suppressing the information that people could enter by themselves for free.

After my first email spam, several weeks went by, and then a different one arrived. I kept that too, for its novelty value. I stopped saving them pretty quickly. Currently spam flows in at 100,000 bytes/hour in ever-increasing torrents. So I have three email problems: answering it, searching it, and fending off spam.

The dead-level limit came in April 2004, when I hit all three problems in one email. Specifically, I got three spam emails that were apparently sent by me. I was spamming myself! OK, I wasn't really spamming myself. Spammers are up to all kinds of tricks to get you to read the pitch and/or click on the viral attachment. They forge sender names at random. Friends had complained in the past of getting spam that was forged to look like it came from me, and finally I got three of them myself within a week.

The email header showed all three of these spams came from the same ISP:

Chapter 25. Networking in Java

"If a packet hits a pocket on a socket on a port, and the bus is interrupted and the interrupt's not caught, then the socket packet pocket has an error to report."

— Programmer's traditional nursery rhyme

The biggest barrier to understanding Java networking features is getting familiar with network terms and techniques. If you speak French, it doesn't mean that you can understand an article from a French medical journal.

Similarly, when you learn Java, you also need to have an understanding of the network services and terminology before you can write Internet code. So this chapter starts with the basics of TCP/IP networking, Everything You Need To Know about TCP/IP but Failed to Learn in Kindergarten, followed by a description of Java support, starting with A Client Socket in Java.

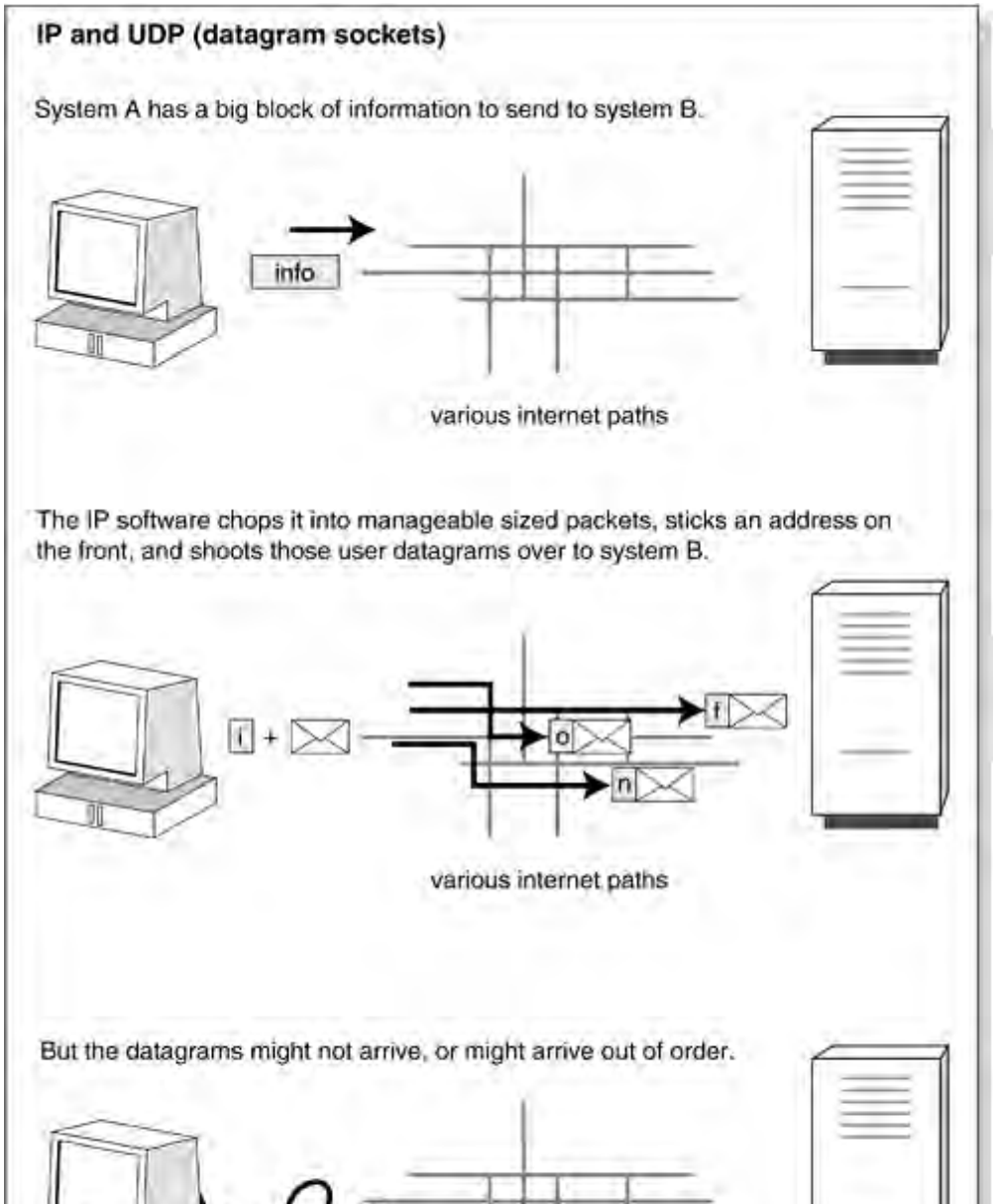
There is a lot of knowledge in this chapter. After the TCP/IP basics, we'll develop some socket examples. We'll see how a client gets services from a remote server using sockets. Then we will look at server sockets to see how incoming connections are accepted. Our first example will merely print HTTP headers. We will add to it little by little until it is a complete HTTP web server.

Everything You Need To Know about TCP/IP but Failed to Learn in Kindergarten

Networking at heart is about shifting bits from point A to point B. We bundle the data bits into a packet, and add some more bits to say where they are to go. That, in a nutshell, is the Internet Protocol or IP. If we want to send more bits than will fit into a single packet, we can divide the bits into groups and send them in several successive packets. The units that we send are called user datagrams or packets. Packet is the more common term these days.

User datagrams can be sent across the Internet using the User Datagram Protocol (UDP), which relies on the Internet Protocol for addressing and routing. UDP is like going to the post office, sticking on a stamp, and dropping off the packet. IP is what the Postal Service does to sort, route and deliver the packet. Two common applications that use the UDP are: SNMP, the Simple Network Management Protocol, and TFTP, the Trivial File Transfer Protocol. See Figure 25-1.

Figure 25-1. IP and UDP (datagram sockets)



A Client Socket in Java

This section shows a simple example of using a socket to communicate with another computer. You should type this code in and try it. If you haven't done much network programming, you'll find it a gleeful experience as you network with systems around the planet, and even in space. The space shuttle has a TCP/IP network connection to Mission Control, but the spoilsports at NASA keep its address secret, so we'll use a different host.

There is an Internet protocol known as Network Time Protocol or NTP. NTP is used to synchronize the clocks of some computers. Without periodic sync'ing, computer clocks tend to drift out of alignment, causing problems for times they need to agree on, like email and file timestamps. NTP is pretty fancy these days, but a simple part of the protocol involves making a socket connection to a NTP server to get the time.

Our example program will open a socket connection to an NTP server and print out the time it gets back. The way a client asks for the time is simply to make a socket connection to port 13 on an NTP server. Port 13 is the Internet standard on all computers for the time of day port. You don't have to identify yourself or write some data indicating what you want. Just making the socket connection is enough to get the server to give you an answer. Java does all the work of assembling the bytes into packets, sending them, and giving you an input stream with the bytes coming back from the server.

Here is a Java program that connects to an NTP server and asks the time:

```
import java.io.*;

import java.net.*;

public class AskTime {


    public static void main(String a[]) throws Exception {

        if (a.length!=1) {

            System.out.println("usage:  java AskTime <systemname> ");

            System.exit(0);

        }


        String machine = a[0];

        final int daytimeport = 13;

        Socket so = new Socket(machine, daytimeport);
```

Sending Email by Java

As our next example, let's write a Java program to send some email. Email is sent by socketed communication with port 25 on a computer system. All we are going to do is open a socket connected to port 25 on some system that is running a mail server and speak "mail protocol" to the sendmail demon at the other end. If we speak the mail protocol correctly, it will listen to what we say, and send the email for us.

The following requires an Internet standard mail (SMTP) program running on the server. If your server has some non-standard proprietary mail program on it, you're out of luck. If your ISP uses a different port for mail (mine uses port 5190) use that instead. You can check which program you have by telnetting to port 25 on the server, and seeing if you get a mail server to talk to you.

Here's how you use telnet to say helo to the SMTP program on port 25 (the bold lines are what you type, the other lines are responses to you):

```
telnet yourisp.com 25

220 yourisp.com SMTP

HELO

250 yourISP.com

QUIT

221 yourisp.com
```

You could feed email to the server by hand, if you memorized the protocol and had the patience. There are two wrinkles to connecting to SMTP servers. First, it became common for spammers to steal time on other people's mailservers to relay their spam. As a result, most mail servers are now selective about who they accept a connection from. You won't be able to get mailers around the world to talk to you, just your ISP mail server. Second, Java now has a mail API with a somewhat higher-level interface, so you don't need to understand individual mail commands. But the point here is to show some give and take over a socket connection. Again, this example shows the client end of the socket connection.

The code to send email is:

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

A Server Socket in Java

This section shows a simple example of creating a server socket to listen for incoming requests. We could write the server side of a simple NTP server, but let's try something a little more ambitious. It should be fairly clear at this point that HTTP is just another of the many protocols that use sockets to run over the Internet.

A web browser is a client program that sends requests through a socket to the HTTP port on a server and displays the data that the server sends back. A basic web browser can be written in a couple of hundred lines of code if you have a GUI component that renders HTML, which Java does.

A web server is a server program that waits for incoming requests on the HTTP port and acts on those to send the contents of local files back to the requestor. It can be implemented in just a few dozen lines of code.

Security of network programs—a cautionary tale!

Be very careful when you start developing networked programs on your computer. Before you try it at work, check if there is a company policy about network use. You can get fired for doing the wrong thing!

The problem is that any server sockets you create may be visible more widely than you intended. If you are running this at home, and you are not using a firewall, your server socket will be visible to the entire net. That's like leaving the front door of your home wide open.

When I was developing the HTTP server in Java for this chapter, I left it running on my PC to test it. Someone's automated port scanner script soon noticed my server, made an unauthorized connection to it, and issued this HTTP command:

```
GET /scripts/..%%35c../winnt/system32/cmd.exe?/c+dir HTTP/1.0
```

This is an attempt to break out of the scripts directory, run a shell, and do a "dir" to see what's on my system. This is the NIMDA worm that pushes its way into the straw house that is Microsoft's IIS. Once in, crackers will try to add their own back door on your computer where you'll never find it. Then they can use your system whenever it's on the net (they love cable modems) for such things as distributed denial of service attacks.

HTTP and Web Browsing: Retrieving HTTP Pages

Here is an example of interacting with an HTTP server to retrieve a web page from a system on the network. This shows how easy it is to post information to HTML forms. Forms are covered in more depth in Chapter 26, and you can peek ahead if you want.

HTML forms allow you to type some information in your browser, which is sent back to the server for processing. The information may be encoded as part of the URL, or sent separately in name/value pairs.

The Yahoo site is a wide-ranging access portal. They offer online stock quotes that you can read in your browser. I happen to know (by looking at the URL field of my browser) that a request for a stock quote for ABCD is translated to a socket connection of:

http://finance.yahoo.com/q?s=abcd

That's equivalent to opening a socket on port 80 of finance.yahoo.com and sending a "get /q?s=abcd." You can make that same request yourself, in either of two ways. You can open a socket connection to port 80, the HTTP port. Or you can open a URL connection, which offers a simpler, higher-level interface. We'll show both of these here.

Here's the stock finder done with sockets:

```
import java.io.*;
import java.net.*;
public class Stock {

    public static void main(String a[]) throws Exception {
        if (a.length!=1) {
            System.out.println("usage:  java Stock <symbol> ");
            System.exit(0);
        }

        String yahoo = "finance.yahoo.com";
        final int httpd = 80;
```

A Multithreaded HTTP Server

There's one improvement that is customary in servers, and we will make it here. For all but the smallest of servers, it is usual to spawn a new thread to handle each request. This has three big advantages:

1. Foremost, it makes the server scalable. The server can accept new requests independent of its speed in handling them. (Of course, you need to run a server that has the mippage to keep up with requests.)
2. By handling each request in a new thread, clients do not have to wait for every request ahead of them to be served.
3. The program source can be better organized, as the server processing is written in a different class.

The following code demonstrates how we would modify our HTTP web server to a new thread for each client request. The first step is to make another child in the OneConnection hierarchy to implement the Runnable interface. Give it a run method that will actually do all the work: get the request, then send the file.

```
import java.io.*;

import java.net.*;

class OneConnection_C extends OneConnection_B

                      implements Runnable {


    OneConnection_C(Socket sock) throws Exception {

        super(sock);

    }


    public void run() {

      try {

        String filename = getRequest();

        sendFile(filename);

      } catch (Exception e) {

        System.out.println("Excpn: " + e);}
```

A Mapped I/O HTTP Server

The final section of this chapter presents the code to use the new mapped I/O facility in a socket server. As a refresher, here is a program that uses channel I/O to duplicate a file:

```
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.net.*;
class DupFile {

    void copyThruChannel(String fname) throws Exception {
        File f = new File(fname);
        FileInputStream fin = new FileInputStream(f);
        int len = (int) f.length();

        FileChannel fc = fin.getChannel();
        System.out.println("allocating buff");
        ByteBuffer myBB = ByteBuffer.allocate(len);
        int bytesRead = fc.read(myBB);
        myBB.flip();

        System.out.println("getting fout channel");
        FileOutputStream fos = new FileOutputStream(fname+".copy");
        FileChannel fco = fos.getChannel();
        int bytesWritten = fco.write(myBB);
        fco.close();
    }

    public static void main(String a[]) throws Exception {
```

Further Reading

TCP/IP Network Administration, by Craig Hunt (O'Reilly & Associates, Sebastopol CA, 2002), ISBN 0596002971.
The modest title hides the fact that this book will be useful to a wider audience than just network administrators. It is a very good practical guide to TCP/IP written as a tutorial introduction.

Internet Core Protocols: The Definitive Guide by Eric A. Hall (O'Reilly, 2000)
It's a cover-to-cover read. The book's only defect is the reliance on a now-unavailable commercial tool. (The tool can be recreated for free using Ethereal, instead).

Unix Network Programming, by W. Richard Stevens (Prentice Hall, NJ, 1990)
The canonical guide to network programming.

Exercises

1. Extend the previous example mail program so that it prompts for user input and generally provides a friendly front end to sending mail.
2. Write a socket server program that simply returns the time on the current system. Write a client that calls the server and sends you mail to report on how far apart the time on the local system is versus the time on the current system.
3. In the previous exercise, the server can only state what time it is at the instant the request reaches it, but that answer will take a certain amount of time to travel back to the client. Devise a strategy to minimize or correct for errors due to transmission time. (Hard—use a heuristic to make a good guess.)
4. Read the API for java.net.URLEncoder and URLDecoder and write a program that encodes a string into the MIME format called x-www-form-urlencoded.
5. Update the multithreaded webserver so that it can also serve JPG and GIF files and correctly identify their type to the browser. You can just use the file extension as an indicator of the contents.

Some Light Relief—500 Mile Limit on Email

This chapter's light relief is a true story about networks and system administration.

Trey Harris is a senior systems engineer, and a vice president of SAGE, the System Administrators Guild (www.sage.org). Trust me, if you have a mission-critical server, you want a guy like Trey to keep it running.

A few years ago, Trey was responsible for email at a university in the Research Triangle area of North Carolina. One day, a peculiar problem report came in from the Dean of the Statistics Department. The dean reported that there was a 500-mile limit on their email. Email to places closer than about 500 miles was usually delivered just fine. But as the Dean said, "500 miles, or a little bit more, is our current email limit."

College deans are creatures with god-like powers on campus. Even when they report something blatantly ridiculous they have to be treated with care. Trey cautiously pointed out that email didn't really work that way. The dean replied that, no, he had all the data that proved it did. When they had first noticed the problem a few days ago, the dean assigned one of the geostatisticians to gather data. After all, this was a Statistics department and that's what they do.

The geostatistician had experimented with a great many email addresses, and had drawn up a map correlating geography and email results. The map showed that the Statistics department could send email to most sites closer than 500 miles, but there was a hard cutoff much beyond that. There were some places within that radius that they couldn't reach or could only reach sporadically, but they could never send email farther than that distance.

Trey knew that whatever this problem was, it was probably caused by someone changing the system configuration. Sure enough, the dean acknowledged that the problem started after a consultant patched the Statistics department server and rebooted it. However, the consultant was certain that he hadn't touched the mail system.

Trey logged into the department's mails server and sent a few test mails. Email to his own local test account went fine. The same thing for email to Raleigh, Atlanta, and Washington. Then he emailed a user in Memphis (600 miles away). It failed. Boston failed. So did Detroit. Trey was seeing exactly the ridiculous problem that the dean reported! At this point, Trey pulled out his address book and an atlas and tried to narrow it down. New York (420 miles) worked, but Providence (580 miles) failed.

One of the first things to check on a mail delivery problem when the system has been patched and rebooted, is the sendmail.cf configuration file. It was fine. Thinking about what to try next, Trey telnetted to the SMTP port to issue a few mail commands by hand. Ah! The first clue! The system responded with the old "Sendmail 5" response that was standard with Solaris at that time, even though Trey had installed the more up-to-date Sendmail 8 throughout the campus.

The pieces quickly came together after that. The consultant had upgraded the version of Solaris, which had wiped out the mail upgrade. The version of Sendmail 5 that Sun shipped as standard could deal with most of the Sendmail 8 sendmail.cf. But the new configuration options that Trey had written with more meaningful names were all ignored and therefore defaulted to zero. One of the settings that became zero was the timeout value to allow while connecting to a remote SMTP server. Some experimentation established that on the Stats department mail server, a zero timeout would

Chapter 26. Servlets and JSP

Overview of Servlets and JSP

The first of these technologies, servlets, provides a way for a browser to cause a program to run on the server. The server calculates something, possibly accesses a database, and sends HTML output back to the browser on the client. In a nutshell, that is the architecture of all web-based B2C (business-to-consumer) e-commerce systems, from Amazon.com to the Zdnet.com online store. These are applications where the end-user interface is provided by a web browser, and the back-end logic runs on a server.

Servlets for creating web pages with the latest data

An ordinary page of HTML is static. Each time the server sends it out, it sends the exact same bytes. The only way it changes is if someone updates the HTML file with an editor. But many kinds of information change dynamically: stock prices, the weather, seats available on a flight, amount of inventory on hand, account balance, contents of an online shopping cart, and so on. Servlets and JSP are a great way to get this dynamic information into a web page. The pages that the user sees are calculated by general-purpose programs that can reference and update databases as part of serving the request.

Servlets are the most popular way for a browser to get a dynamic web page from a program on the server. The old way used an interface called CGI, and your scripts would be written in Perl or Visual Basic or some other language. With servlets, your code is written in Java. There is a size/complexity tradeoff here: small scripts (less than a couple of pages) can be written at the drop of a hat and are well suited to Perl or PHP.

What web servers and web browsers do

- A web browser is just a program that sends requests to the HTTP port (port 80 by default) on a server, and displays the data that the server sends back. A basic web browser can be written in a couple of hundred lines of code (if you have a GUI component that renders HTML, which Java does). It's only when people start adding support for newsreaders, mail, instant messaging, HTML editing, SETI analysis, and so on, that a browser balloons up in size.
- A web server is just a program that waits for incoming requests on the HTTP port, and responds by sending the contents of local HTML and image files back to the requestor.
- A servlet container (such as Apache's Tomcat) is an add-on to a web server. It will run a servlet program in response to a request on port 8080 by default. The output (usually HTML) of the servlet program will be sent back to the requestor.

The larger and more complicated your code, the more you will benefit from using a Java servlet. If you need to access a database, you can use Java's JDBC library. If you need threading or network libraries, Java has them. If your code

Why Use Servlets?

Server-side web programming is well established on the web. It lets you build systems with a client part that can run in a browser on any computer, with any operating system, at any time, from anywhere in the world.

Clients and servers

The HTTP protocol provides a framework for communicating with a server, where all the real work is done. The server-side code may do load-balancing to move incoming requests to the system best equipped to handle them. The server code can access/update your database and process any data using the most up-to-date information. It does this in a safe way.

The client cannot call server routines directly. It can only send over HTTP requests saying what it wants. Opportunities to subvert the server are more restricted than when everything runs on one physical system. You still need to code defensively because people may try to access your page in a way that could make the server spend all its time trying to service their request.

Client/server programming was popular even before the web went mainstream, and bringing the two together was a natural marriage. CGI—the "Common Gateway Interface"—was the first attempt to get dynamic content into web pages. CGI got the job done, but it had big problems with security and performance. CGI implementations often start up an entire new process to run a script. It doesn't have to be done that way, but in practice it usually was. Servlets are loaded into memory once, and stay around ready to handle all future requests. So there can be a big performance advantage to using servlets. It also means that servlets can choose to hold system resources (such as a database connection) between requests. Since opening and closing a database connection is a lengthy operation, this is another win for servlets.

Servlets make it easier to separate the business logic used to generate results from the HTML that displays those results. JSP takes that one step further. Separation of logic from presentation has benefits. It's an enabler for the use of component software such as Java Beans. Component software lets a system designer deploy the same code to handle a transaction whatever the origin is: web-based, online transaction, or batch processing. We're getting into enterprise-level software issues here, but servlets deliver consistency and code reuse.

What servlets replace

Servlet technology replaces the family of server plugins such as the Netscape API, CGI, or the Microsoft ISAPI, none of which are standardized or multiplatform like the servlet API. Servlets today are the most popular choice for building portable interactive web applications. Add-on software to run servlets is available for Apache Web Server, iPlanet Web Server, Microsoft IIS, and others. Servlet containers can also be integrated with web-enabled application servers such as BEA WebLogic Application Server, IBM WebSphere, iPlanet Application Server, and others.

Servlets as web services

Servlets are highly effective in implementing web services. Chapter 28 reviews two public implementations of web services. For now, think web services = "a way for a program here to run a program on a remote server and get the

Releases and Versions

By agreement with Sun Microsystems, the reference implementation for servlets and JSP is maintained by the Apache Software Foundation. The Apache web server is an open source project, and by far the most widely used web server in the world. You can look up more information about the Apache Software Foundation at www.apache.org. Apache uses the project name "Jakarta" for their Java-specific work.

The servlet library is part of the Enterprise Edition of Java, so must be downloaded separately if you are using the Standard Edition. The servlet design underwent some rapid evolution at first with a number of versions and releases, but it has settled down now. Table 26-1 and Table 26-2 spell out the details, so you can relate everything to other versions you may have heard about. At the time of this writing (2004), the most up-to-date version of the servlet API is version 2.4, and that is used in this chapter. Check the Apache website listed in Table 26-2 for any later version and download and use that in preference. This technology is mature enough that there shouldn't be any drastic changes in future revisions.

Table 26-1. Server/servlet versions

Date shipped	Servlet container version	Compatible with servlet version	Compatible with Java Server Pages version
2003	Tomcat 5	Servlet 2.4	JSP 2.0

Table 26-2. Glossary

Term	Definition
Apache	The Apache Software Foundation is a volunteer organization which has produced the most popular web server in the world, and made it available for free download. Their website is www.apache.org
Ant	A utility that accompanies Tomcat, used when Tomcat has to compile some servlet code automatically. Ant is a Java-based build utility that works out the correct dependency order in which to compile each file. It works cross-platform, unlike non-Java based tools. Ant is a replacement for "make" when used with Java.
Catalina	The servlet container (engine) part of Tomcat version 5.
CGI	The Common Gateway Interface was the first server-side scripting technology. It specifies how the server should execute a script when a particular web page is referenced. CGI scripts can be written in different languages, and Perl is a popular choice.

Installing the Tomcat Software

This section describes how to download and install the Tomcat servlet web server software. Note that there is a FAQ at http://jakarta.apache.org/tomcat/faq. To get started on your first servlet, follow these steps.

1. Go to http://jakarta.apache.org, follow the links to the download page, and download the binary of the most recent version of Tomcat 5. Get the right version for your system. For Windows this is an exe file. It's a 10MB download.

2. Unpack the file you downloaded (if a zip or gz file). Run the file if it is an exe file. You'll need to provide the admin password on Windows. Accepting all the defaults works fine, although it is a Very Good Idea to install into a directory called "\tc5" instead of the default. The default pathname is way too long and contains embedded spaces, so you have to quote it every place you use it.

3. The last step in the installation asks you if you want to start Tomcat. You do. You can also stop and start Tomcat manually on WinXP through Control Panel/Performance and Maintenance/Administrative Tools/Services/Apache Tomcat.

4. After you have started Tomcat, browse http://localhost:8080 If everything went smoothly, you will see a page served back to you by Tomcat that looks like this:

Running the Example Servlets

Tomcat will run as a stand-alone web server, which is very convenient for development. It can also be configured to run as an adjunct to most other web servers to handle only the servlet/JSP requests. That's useful for deployment in real IT environments, and we won't get into it here.

The next step is to try running Tomcat on one of the example servlets that accompany it. First, shut down any other web servers that you have running on your system, so they don't interfere with this example. Then start Tomcat by following the steps in the following box.

Starting and stopping Tomcat

Tomcat is installed as a system service on WinXP. Start Tomcat on WinXP using the Control Panel -> (category view) -> Performance and Maintenance -> Administrative Tools -> Services -> Apache Tomcat.

On Unix, you can use the startup and shutdown scripts in the bin directory.

```
% cd $TOMCAT_HOME

% bin\startup
```

For security reasons, you should not leave a web server running on your system where outsiders may see it. There is a corresponding shutdown script in "bin/shutdown."

Tomcat serves web pages to requests that come through port 8080, so start a browser and give it the URL http://localhost:8080. You can use the name "localhost" or the special IP address "127.0.0.1" that means "this system." Or if your computer has a DNS name, you can also use that in the URL. You should see the page in Figure 26-1 again.

Figure 26-1. Home page of Tomcat on your system

Ports and Protocols

As we saw in Chapter 25, you use input and output streams on sockets just as though you were reading/writing a file. The class java.net.ServerSocket lets your server program accept incoming data by spawning off a thread with a socket that you can read client requests from. The class java.net.Socket lets you send data to a server socket on another computer, or read response data that is coming back into the local host.

Servlets are a higher-level alternative to reading/writing sockets. Servlets can be used to service any request that is made via a socket (such as FTP), not just web page requests via HTTP. A servlet that talks something other than HTTP is called a "generic servlet" and it will extend the class javax.servlet.GenericServlet. The vast majority of servlets are used to serve HTTP requests. These are known as "HTTP servlets," and they extend the class javax.servlet.http.HttpServlet.

The computer science term "protocol" means "an agreement on how to talk to each other." Browsers and servers talk to each other using HTTP, HyperText Transfer Protocol. The browser starts the conversation, and then each end takes its turn to say something. It goes back and forth over the net like a game of tennis. Each HTTP request from a browser is replied to with a response from the server.

A commercial web server is multithreaded and typically deals with many clients (browsers) at any moment, but is either reading a request or sending the response to each. The key concept of servlets is that when you browse a page on the client, it causes the servlet to run on the server.

The servlet does whatever processing was coded, and then (usually) writes some HTML to represent the answer The web server sends that newly generated HTML back to the browser for display. Just as with a regular HTML page, a servlet can be invoked many times. A servlet can cope with several concurrent requests, and it may call another servlet or forward the original request to it for processing.

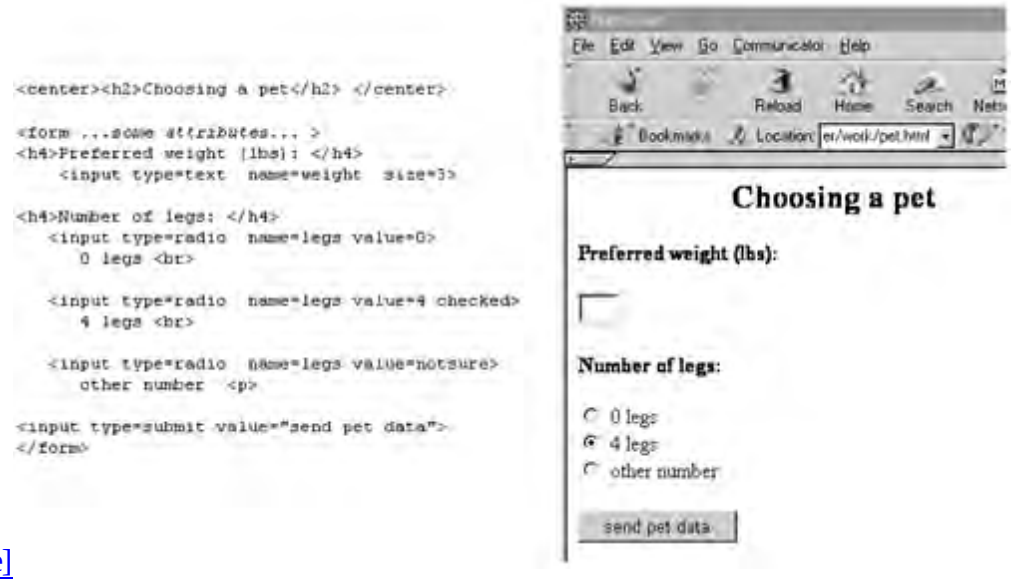
The HTML to Invoke a Servlet

This section describes the HTML that will be displayed on the client and cause a servlet to run on the server. The most common way for a browser to invoke a servlet is via an HTML button that causes the entire form, with all the data the user typed in, to be sent over to the server.

There are about ten different GUI input types, but the most often used are text, radio, checkbox, and submit. For a complete list of all the input types and other attributes, do a web search on "HTML, form, guide." To make it all line up nicely on the screen, everything inside a form is often put in an HTML table.

Figure 26-2 shows an example of an HTML form, some INPUT tags, and the web page they generate. You should create a web page with the HTML shown here, and confirm that you can browse it, enter data, and click the submit button.

Figure 26-2. HTML example form



[View full size image]

How form data is sent to a URL

Now we come to the question of how and where the browser sends the data from the form. The form tag will always have two attributes (omitted in Figure 26-2, for simplicity) that specify how and where the form data goes. These attributes are called "action" and "method." There are also additional possible attributes, to give the whole form a name, and to say how the data should be encoded before it is sent to the server. The default values are fine for these. An example of a complete form tag would be

A Servlet and Its Request/Response

In this section we will present the skeleton of a servlet, and also look at the objects that it uses to learn about a request and send a response. A servlet is just like any other Java program, but one that runs inside a web server engine, termed the "container."

There are some configurations or conventions that tell the servlet container where your servlet is and what URL should invoke it. The servlet container will call your overriding methods when that URL is requested, and pass in parameters that convey all the information in the form sent from the client browser.

You create your servlet by extending one of the javax.servlet classes, and overriding one or more of the methods in it with your own code. The skeleton of an HTTP servlet looks like this:

```
public class MyServlet extends javax.servlet.http.HttpServlet {

    public void init() { /* code */ }

    public void doGet() { /* code */ }

    public void doPost() { /* code */ }

    public void destroy() { /* code */ }

}
```

This code is simplified slightly by leaving off the method parameters and the exceptions they can throw. They are shown a couple of paragraphs later. The init() method is called only once when the class is first loaded. You would use this for one-time initialization, such as opening a connection to a database. The destroy() method is also called only once when the servlet is unloaded. This method is used to free any remaining resources or do final housekeeping on shutting down. If you don't have any special startup or shutdown code, you don't need to override these methods.

The doGet() or doPost() methods are the ones that do the work of the servlet. Obviously, doGet() is called when the HTML form used a get method, while doPost() is invoked by a form with a post method.

There are other less important doSomething() methods, too, corresponding to the other things that an HTML form may do. For example, there is a doDelete() method that can be overridden for the less-common HTTP request DELETE. This request is rarely implemented because you generally don't want to empower users with the ability to delete files on the server. You might allow it on a server on your intranet. You can review other methods in the API docs that are part of the servlet kit.

Servlet Request

The first part of a request will be the HTTP headers, followed by the parameters. Headers are the bookkeeping information supplied automatically by the browser or server, stating things like the locale, and the version of HTTP in use. Parameters are provided by the user and passed in the query string or in the form data. A parameter name is whatever name the HTML form designer gave the parameter in an attribute. It is legal for parameter or headers to have a comma-separated list of values, so you need to be alert to this possibility in your code.

Let's take a look at the classes that implement request and response objects to see what information comes and goes. We'll start with the Java class that represents an HTTP request. Tomcat will create an object that implements this interface to hold all the data in the incoming request. Tomcat will then invoke your servlet, and pass the request object to it as an argument.

These methods are just the highlights of an HTTP servlet request object. There are about 20 getSomething() methods in HttpServletRequest, and another 20 in its parent, ServletRequest, allowing all information in the request to be retrieved. You will invoke these methods (shown in Table 26-5) on the javax.servlet.http.HttpServletRequest parameter, as shown in the example coming up.

Table 26-5. Key Methods of javax.servlet.http.HttpServletResponse

Method	Purpose
getWriter()	Returns a PrintWriter that will get written with the data part of the servlet response.
setHeader(String n, String v)	Adds a response header with the given name and value.
setDateHeader(String s, long d)	Adds a response header with the given name and time value.
setIntHeader(String s, int v)	Adds a response header with the given name and int value.
addCookie(Cookie c)	Adds the specified cookie to the response.
setStatus(int sc)	Set the status code for this response.
setContentType (String s)	Sets the response's MIME content type.
setContentLength (int size)	Sets the Content-Length header of the response.

Notice that some of these methods, such as getHeaders(), return Enumeration objects, rather than the newer Iterator object that was intended to replace Enumeration and was part of JDK 1.2. This is for backwards compatibility with existing servlet code.

Response to a Servlet Request

Tomcat will also pass your servlet one of these response objects as an argument. The object has lots of methods that let you give values to its fields. You send back the actual data by writing to a print writer that you get from the response object.

Table 26-5 shows the most frequently called methods of your response object, the HTTP servlet response.

You will invoke these methods on the javax.servlet.http.HttpServletResponse parameter, as shown in the example coming up. There are about 20 methods in HttpServletResponse and its parent class, ServletResponse, allowing just about any field to be set and returned to the browser. There are about 20 static final variables giving names to each of the status codes. You should review the javadoc descriptions of these classes.

There are about a dozen HTTP headers, but you can ignore them unless you need the special effects that they cause. The content type is the only one you need to set. You can also look a few pages further on in this chapter, in Java Server Pages on page 710, where we write a JSP program to echo the headers received from the browser. That shows you some typical headers.

Writing Your Own Servlet

Here is the code for a servlet that can process the HTML form that we created in the section How form data is sent to a URL on page 694. We're going to send a reply that suggests a suitable pet based on the weight and leg count the user submitted.

Compiling your servlet

Make sure that your CLASSPATH has the servlet jar files in it, as shown in the following command, applicable to Windows:

```
javac -Djava.ext.dirs="\program files\apache software foundation\Tomcat 5.0\common\lib"
➥ PetServlet.java
```

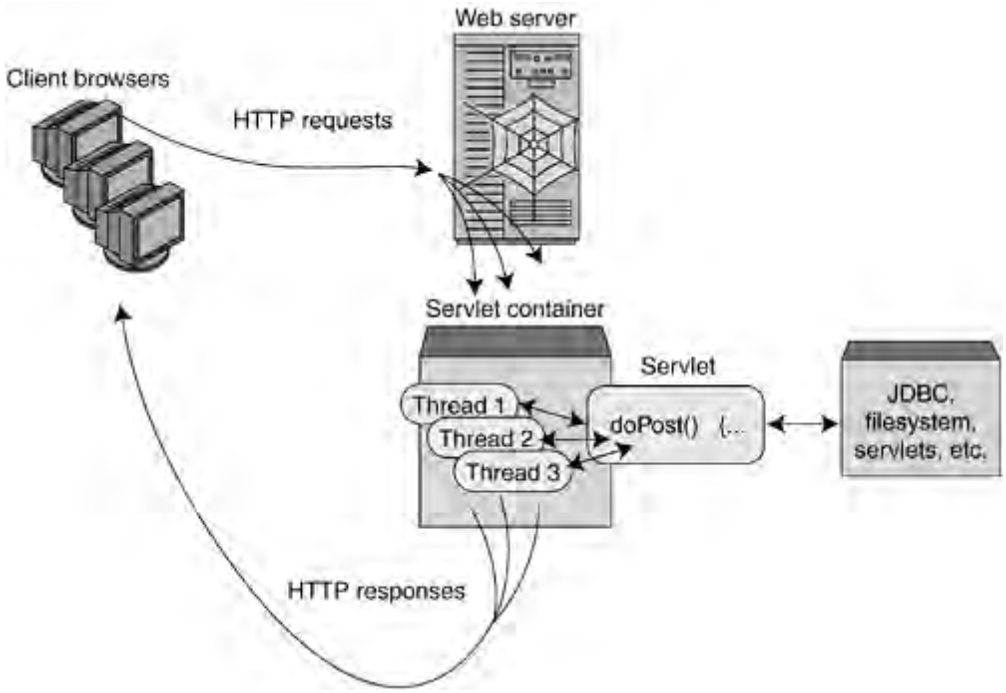
Note: The example command is just one line, but is too wide to print that way in a book. The "-Dproperty=value" option defines a property with that value to the compiler system. The "java.ext.dirs" property is the pathname to a directory of jar files (in this case the Tomcat jar files) that you want to be available to the compiler or run-time. The servlet library is not part of the JDK standard edition. It is part of the Enterprise Edition. It is also bundled with Tomcat.

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.text.*;
import java.util.*;

public class PetServlet extends HttpServlet {

    private String recommendedPet(int weight, int legs) {
        if (legs ==0) return "a goldfish";
        if (legs ==4) {
```

Servlet Operating Cycle and Threading

Servlets typically have the operating cycle, as shown in Figure 26-5.

Figure 26-5. Operating cycle of a servlet



Servlet life cycle

The sequence of events in a servlet's lifetime is:

1. The servlet container starts up, and at some point constructs an instance of the servlet and calls its init() method. The init() method is only called once. Not once per request, not once per session, but once at the beginning of the servlet's lifetime. The init() method is a good place to put code to open a database connection.

2. Unlike, say, a GUI, there is no background thread always running code in the servlet. The servlet instance object just stays ready in memory, waiting for a request. This makes servlets very efficient. Eventually, a request comes to the web server, and the web server passes the request to the servlet container.

3. The servlet container instantiates a new thread to process the request. Note that the container does not instantiate a new servlet object. The newly created thread representing the request calls the doPost() method (or whatever is appropriate for this request) of the existing instance of your servlet. The servlet can access a database, the filesystem, other servlets, etc. It creates the HTTP response, which the container returns to the client. Thread-per-request makes servlets scalable and high performance.

4.

Java Server Pages

We will conclude this tour of server-side Java with a description of Java Server Pages (JSP), and an example JSP program. One way of understanding JSP is to say that JSP is ASP, without the restriction to Windows only. Another way of understanding it is to say that JSP programs are a variant on ordinary servlets, where some of the simple tasks are automated for you. In fact, the container implements JSPs by automatically translating them into the equivalent servlet which is then run in the usual way.

A JSP is a slick way of writing a servlet

An ordinary unchanging web page contains HTML (plus Javascript perhaps). A servlet is a compiled Java program. A JSP program is a hybrid of these two. It lets you mix individual Java statements in with your HTML code. The Java code will be executed on the server when the page is browsed, and it will provide some dynamic content to the page. You might do some calculations, or put something in a loop.

Your JSP Java code fragments automatically have the missing boilerplate code added, to make a complete servlet. This servlet is automatically compiled for you by the JSP container when the page is browsed. As with servlets, JSP code is compiled once and loaded into memory on first use. A developer will typically browse all the JSP pages when deploying a system, so that users don't see the "first time through" compilation time penalty.

JSP syntax

A large part of a servlet is "boilerplate," meaning text that is the same in all servlets. The class declaration, the method signatures, and so on, are needed to make sure your code compiles, but they are the same in every servlet. JSP eliminates all that standard context. It is provided for you automatically. This can dramatically shorten the amount of code you need to write, and also makes it simple enough for non-programmers to produce JSP.

JSP uses special tags to separate the Java from the HTML. The JSP opening tag is "<%" and the closing tag is "%>". The opening tag "<%" might be followed by another character such as "!" or "@" or "="to further specialize its meaning. A very brief example here will show you best. These lines in a JSP file:

```
<b> current time is:

    <%=  new java.util.Date() %>

</b>
```

will produce a line of output like this when you browse the JSP:

current time is: Mon Feb 19 18:37:23 PST 2001

Java Beans in Servlets and JSP

This chapter wouldn't be complete without pointing out the role of Java beans on the server side. Java beans are software components, namely well-specified "modules" that do a specific job and can easily be reused in many applications. Microsoft makes extensive use of software components under the product name ActiveX, often for some visual or GUI feature.

For example, a programmer might write a piece of code that can display a set of numbers as a pie chart. That routine is very suitable for turning into a software component, making it available to any program on the system.

The point of JSP is to use lots of Java beans that cover your business processes. You might have one bean that encompasses everything you can do with a customer record, another for an order, and a third that represents a payment transaction. JSP lets you easily integrate these beans in a web-based display framework. JSP has a special tag that lets web pages on the server easily interact with beans with hardly any "glue" code needed. People call this a "tag library."

Here's an example of the tags that connect a JSP page to a Java bean that manages database access.

```
<%@ page language="java" import="java.sql.*" %>


<jsp:useBean  scope="request"

  class="com.afu.database.DbBean" />


<jsp:setProperty name="db" property="*" />
```

You use the same beans (the same logic) for your non-web-based processing so you have the advantages of consistency, familiarity, and software reuse. The combination of Java beans and JSP is a major use of JSP.

Last Words on JSP, Beans, and Tag Libraries

The main reason for using JSP is that it allows web developers to quickly build web pages that interface to enterprise systems. The JSP tags let HTML designers tie web information into corporate business logic contained in Java objects without having to learn all about Java object-oriented programming.

A separate, smaller programming team can create libraries of software components. Then web designers can use those libraries by writing markup tags that they are familiar with. Use of tag libraries is one of the cornerstones of JSP. JSP thus provides a rapid prototyping framework for building two tiers (the client and the front-end server) of an N-tier distributed system.

Servlets can act as a middleware gateway to existing legacy systems, providing an easy way to web-enable your current systems. Furthermore, since all the code is on the server (not the client), when you want to update your application you just roll the code out to a few servers and your entire user base gets the newest code at once.

Further Reading

If you are interested in a more detailed study of servlets, download and read the Java Servlet 2.4 Specification from the java.sun.com website. Although its title is "Specification," it actually contains some good explanations of the details of servlets.

After you master the basics here, you can go on to get deeper knowledge from either Hall and Brown's Core Servlets and JavaServer Pages, 2d ed, 2003, Prentice Hall PTR, (make sure you get the latest edition) or Servlets and JSP: The J2EE Web Tier by Jayson Falkner, Kevin R. Jones, Addison-Wesley, 2003.

If you want to know what kind of web server a site is running, or you want to see the market share of different web servers, look at www.netcraft.com. Finally, the website www.servlets.com is a great resource for programmers writing servlets and JSP code.

Exercises

1. Modify the petform servlet so that it includes the content length in its response.
2. Write a servlet that sends back to the client (for display) all the parameters and HTTP request headers that it received. Have the servlet get enumerations of all the headers and all the parameters, and echo them back to the client.
3. Write a JSP that handles our pet selection form.
4. Earlier in this chapter, we showed an HTML form that invoked a servlet. It's actually possible to write a servlet or JSP that delivers that form as well as responding to it. When the servlet is invoked by a URL, it should respond with the HTML representing the form for pet selection. When the servlet is invoked by submitting the form, it should make the pet selection. That keeps everything relating to pet selection in one file, possibly easing maintenance. You can tell if a form was submitted by doing a requestParameter() on any of the argument names, like this: String formSent = request.getParameter("legs"). If the string comes back null, there wasn't a form submitted (or at least that argument was not filled in), so the servlet must have been invoked with a URL reference. The service routine should then generate the form. Otherwise, the servlet should send back the HTML with the pet selection. Write a JSP file so that it delivers the pet selection form in this way, and responds to it too.

Some Light Relief—Using Java to Stuff an Online Poll

The email to me was brief. It just read:

```
From billg@Central Mon May  4 11:57:41 PDT
Subject: Hank the Angry Dwarf
To: jokes@Sun.COM

Hey everyone.  If you've got five seconds to spare, go to the following url:
     http://www.pathfinder.com/people/50most/1998/vote/index.html

and vote for:
     Hank the Angry, Drunken Dwarf

This is a huge joke.  We want to try to get Hank way up there on the People Magazine 50
➥ most beautiful people of the year list.  As of 2:00AM, he's already up to number 5!
```

Well, I can recognize a high priority when I see one. I put down the critical bug fix I was working on, went right to the website, and checked what this was all about.

What this was all about

Every year the celebrity gossip magazine People prints a list of "the 50 most beautiful people in the world," and this year they were soliciting votes on their website. People had started the ball rolling with nominations for actors like Kate Winslet and Leonardo DiCaprio, who were in the public eye because of their roles in the Titanic movie.

People magazine gave web surfers the opportunity to write in names of people for whom they wanted to vote. A fan of the Howard Stern radio show nominated "Hank the angry, drunken dwarf" for People's list. When Stern heard about Hank's nomination as one of the most beautiful people in the world, he started plugging the candidacy on the radio. A similar phenomenon took place on the Internet, and many people received email like I did. Another write-in stealth candidate widely favored by netizens was flamboyant, blond-haired, veteran pro-wrestler Ric Flair.

Hank is an occasional guest on Stern's syndicated radio program. Hank is a very short 36-year old dude who lives in Boston with his mother and enjoyed his 15 minutes of fame as a belligerent, if diminutive, devotee of beer, tequila, and

Chapter 27. XML and Java

XML Versus HTML

You'll probably be relieved to hear that the basics of XML can be learned in a few minutes, though it takes a while longer to master the accompanying tools and standards. XML is a set of rules, guidelines, and conventions for describing structured data in a plain text editable file. The abbreviation XML stands for "eXtensible Mark-up Language."

XML is related to the HTML used to write web pages, and has a similar appearance of text with mark-up tags sprinkled through it.

- HTML mark-up tags are things like
 (break to a new line), <table> (start a table), and <li> (make an entry in a list). In HTML the set of mark-up tags are fixed in advance, and the only purpose for most of them is to guide the way something looks on the screen.
- With XML, you define your own tags and attributes (and thus it is "extensible") and you give them meaning, and that meaning goes way beyond minor points like the font size to use when printing something out.

XML advantages over HTML

Don't make the mistake of thinking that XML is merely "HTML on steroids." Although we approach it from HTML to make it easy to explain, XML does much more than HTML does. XML offers the following advantages:

- It is an archival representation of data. Because its format is in plain text and carried around with the data, it can never be lost. That contrasts with binary representations of a file which all too easily become outdated. If this was all it did, it would be enough to justify its existence.
- It provides a way to web-publish files that can be directly processed by computer, rather than merely human-readable text and pictures.
- It is plain text, so it can be read by people without special tools.
- It can easily be transformed into HTML, or PDF, or data structures internal to a program, or any other format yet to be dreamed up, so it is "future-proof."
- It's portable, open, and a standard, which makes it a great fit with Java.

We will see these benefits as we go through this chapter. XML holds the promise of taking web-based systems to the next level by supporting data interchange everywhere. The web made everyone into a publisher of human-readable HTML files. XML lets everyone become a publisher or consumer of computer-readable data files.

Some Rules of XML

XML follows the same kind of hierarchical data structuring rules that apply throughout most programming languages, and therefore XML can represent the same kind of data that we are used to dealing with in our programs. As we'll see later in the chapter, you can always build a tree data structure out of a well-formed XML file and you can always write out a tree into an XML file. When you want to send XML data to someone, the XML file form is handy. When you want to process the data, the in-memory tree-form is handy. The purpose of the Java XML API is to provide classes that make it easy to go from one form to the other, and to grab data on the way.

The XML element

Notice that all XML tags come in matched pairs of a begin tag and an end tag that surround the data they are describing, like this:

```
<someTagName>    some data appears here   </someTagName>
```

The whole thing—start tag, data, and end tag—is called an element.

You can nest elements inside elements, and the end tag for a nested element must come before the end tag of the thing that contains it. Here is an example of some XML that is not valid:

```
<cd>   <title>White Christmas  </cd>  </title>
```

It's not valid because the title field (or "element" to use the proper term) is nested inside the cd element, but there is no end tag for it before we reach the cd end tag. This proper nesting requirement makes it really easy to check if a file has properly nested XML. You can just push start tags onto a stack as they come in. When you reach an end tag, it should match the tag on the top of the stack. If it does, pop the opening tag from the stack. If the tag doesn't match, the file has badly nested XML.

XML attributes

Just as some HTML tags can have several extra arguments or "attributes," so can XML tags. The HTML <img> tag is an example of an HTML tag with several attributes. The <img> tag has attributes that specify the name of an image file, the kind of alignment on the page, and even the width and height in pixels of the image. It might look like this:

The Document Type Definition (DTD)

There is another level of data validation in addition to a document being "well-formed." You also want to be able to check that the document contains only elements that you expect, all the elements that you expect, and that they only appear where expected. For example, we know this is not a valid CD inventory entry:

```
<cd>  <price>22</price>   <qty>3</qty>  </cd>
```

It's not valid because it doesn't have a title or artist field. Although we have three in stock, we can't say what it is three of.

What a DTD does

XML files therefore usually have a Document Type Definition or "DTD" that specifies how the elements can appear. The DTD expresses which tags can appear, in what order, and how they can be nested. The DTD can be part of the same file, or stored separately in another place. A well-formed document that also has a DTD and that conforms to its DTD is called valid.

DTD syntax

The DTD is itself written using something close to XML tags, and there is a proposal underway to align the DTD language more closely to XML. You don't need to be able to read or write a DTD to understand this chapter, but we'll go over the basics anyway. There is a way to specify that some fields are optional and thus might not be present. In other words, it's the usual type of "a foo is any number of bars followed by at least one frotz" grammar that we see throughout programming, with its own set of rules for how you express it. Here's a DTD that specifies our CD inventory XML file.

```
<!ELEMENT inventory (cd)* >

   <!ELEMENT cd (title, artist, price, qty)>

        <!ELEMENT title (#PCDATA)>

        <!ELEMENT artist (#PCDATA)>

        <!ELEMENT price (#PCDATA)>

        <!ELEMENT qty (#PCDATA)>
```

What Is XML Used For?

There seems to be agreement from all sides that XML has a bright future. Microsoft chief executive Steve Ballmer said that he thinks use of XML will be a critically important trend in the industry. Why is this? What motivated XML's design?

The origins of XML

XML was developed in the mid 1990s under the leadership of Sun Microsystems employee Jon Bosak. Jon was looking for ways to use the Internet for more than just information delivery and presentation. He wanted to create a framework that would allow information to be self-describing. That way applications could guarantee that they could access just about any data. That in turn would clear the path to intelligent data-sharing between different organizations. And that in turn would allow more and much better applications to be written and increase the demand for servers to run them on. Well, that last part isn't a goal, but it's certainly a great side effect for anyone in the computer hardware industry.

XML solves data incompatibility

Information access might not sound like a problem in these days of web publishing, but it used to be a significant barrier. The web is still not a good medium for arbitrary binary data or data that is not text, pictures, or audio. A few years ago, every hardware manufacturer had a different implementation of floating-point hardware, and the formats were incompatible between different computers. If you had a tape of floating-point data from an application run on a DEC minicomputer, you had to go through unreasonable effort to process it on another manufacturer's mainframe. IBM promoted its EBCDIC (Extended Binary Coded Decimal Interchange Code) convention over the ASCII (American Standard Code for Information Interchange) codeset standardized in the rest of the Western world. People who wanted to see their printouts in Japanese resorted to a variety of non-standard approaches.

By storing everything in character strings, XML avoids problems of incompatible byte order (big-endian/little-endian) that continue to plague people sharing data in binary formats. By stipulating Unicode or UTF encoding for the strings, XML opens up access to all the locales in the world, just as Java does.

XML makes your data independent of any vendor or implementation or application software. In the 1960s, IBM launched a transaction processing environment called CICS. CICS was an acronym for "Customer Information Control System." When a site used CICS, after a while it usually became completely dependent on it, and had to buy large and continuing amounts of hardware and support from IBM in order to keep functioning. People used to joke that it was the customer that was being controlled, not the information. But it was no joke if you were in that position. Modern software applications cause the same kind of single-vendor lock-in today. XML goes a long way to freeing your data from this hidden burden. But note this key point: just because something is published in XML does not make it openly available. The DTD and semantic meaning of the tags must also be published before anyone can make sense of non-trivial documents.

XML means we can all just get along

So XML makes it possible for otherwise incompatible computer systems to share data in a way that all can read and write. XML markup can also be read by people because it is just ordinary text. So what new things can be done with

XML Versions and Glossary

Table 27-1 contains the latest version numbers relating to XML. This chapter describes the most up-to-date version of everything available at the time of this writing (Summer 2004).

Table 27-1. XML-related version numbers

API	Version number	Description
JAXP	Ver. 1.2	Java API for XML processing. Includes an XSLT framework based on TrAX (Transformation API for XML) plus updates to the parsing API to support DOM Level 2 and SAX version 2.0. The remainder of this chapter has more information on JAXP.
XSLT	Ver. 1	XSLT is a conversion language standardized by W3C that can be used to put XML data in some other form such as HTML, PDF, or a different XML format. For example, you can use XSLT to convert an XML document in a format used by one company to the format used by another company. See www.zvon.org for a tutorial on "eXtensible Stylesheet Language Transformations" (XSLT).
SAXP	Ver. 2.0	Simple API for XML Parsing. This is covered later in this chapter.
DOM	Level 2	Document Object Model, which is another API for XML parsing. This is covered later in this chapter.
JAXM	Ver. 1.1.2	Java Architecture for XML Messaging. A new specification that describes a Java library for XML-based messaging protocols. Objects and arguments (messages) will be turned into XML and sent to other processes and processors as streams of characters.
JAXB	Ver. 1.0	Java Architecture for XML Binding. A convenient new Java library for

JAXP Library Contents

This is a good point to review the packages that make up the Java XML library, their purpose, and their classes. The different package names reflect the different origins of the code. The Java interfaces came from Sun Microsystems, the DOM implementation came from the W3C, and the SAX parser implementation came from yet a third organization.

Package: javax.xml.parsers

- Purpose: Is the Java interface to the XML standard for parsing.
- Contains these classes/interfaces: DocumentBuilderFactory, DocumentBuilder, SAXParserFactory, SAXParser. These get instances of a Parser and undertake a parse on an XML file.

Package: javax.xml.transform

- Purpose: is the Java interface to the XML standard for tree transformation
- Contains: classes to convert an XML tree into an HTML file or XML with a different DTD. Tree transformation is beyond the scope of this text, but you can read more about it by searching for "transform" at java.sun.com.

Package: org.w3c.dom

- Purpose: has the classes that make up a DOM tree in memory
- Contains these classes/interfaces: Node plus its subtypes: Document, DocumentType, Element, Entity, Attr, Text, etc.

Package: org.xml.sax

- Purpose: has the classes that can be used to navigate the data returned in a SAX parse
- Contains: two packages org.xml.sax.ext (extensions) and org.xml.sax.helpers plus these classes/interfaces: Attributes, ContentHandler, EntityResolver, DTDHandler, XMLReader. The helpers package contains the DefaultHandler class which is typically extended by one of your classes to handle a SAX parse, as explained below.

Reading XML With DOM Parsers

XML documents are just text files, so you could read and write them using ordinary file I/O. But you'd miss the benefits of XML if you did that. Valid XML documents have a lot of structure to them, and we want to read them in a way that lets us check their validity, and also preserve the information about what fields they have and how they are laid out.

What we need is a program that reads a flat XML file and generates a tree data structure in memory containing all the information from the file. Ideally, this program should be general enough to build that structure for all possible valid XML files. Processing an XML file is called "parsing" it. Parsing is the computer science term (borrowed from compiler terminology) for reading something that has a fixed grammar, and checking that it corresponds to its grammar. The program is known as an "XML parser." The parser provides a service to application programs. Application programs hand the parser a stream of XML from a document file or URL, the parser does its work and then hands back a tree of Java objects that represents or "models" the document.

An XML parser that works this way is said to be a "Document Object Model" or "DOM" parser. The key aspect is that once the DOM parser starts, it carries on until the end and then hands back a complete tree representing the XML file. The DOM parser is very general and doesn't know anything about your customized XML tags. So how does it give you a tree that represents your XML? Well, the DOM API has some interfaces that allow any kind of data to be held in a tree. The parser has some classes that implement those interfaces, and it instantiates objects of those classes.

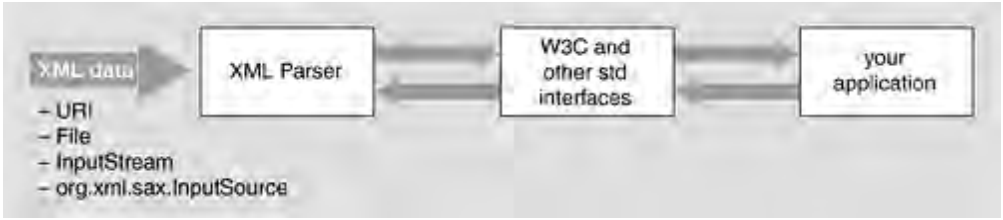
It's all kept pretty flexible, and allows different parsers to be plugged in and out without affecting your application code. Similarly, you get information out of the tree by calling routines specified in the DOM API. The Node interface is the primary datatype for the Document Object Model. It represents a single node in the document tree, and provides methods for navigating to child Node. Most of the other interfaces, like Document, Element, Entity, and Attr, extend Node. In the next section we will review the code for a simple program that uses a DOM parser. DOM parsers can be and are written in any language, but we are only concerned with Java implementations here.

A Program That Uses a DOM Parser

This section walks through a code example that instantiates and uses a DOM parser. The DOM parser is just a utility that takes incoming XML data and creates a data structure for your application program (servlet, or whatever) to do the real work. See Figure 27-3 for the diagram form of this situation.

Figure 27-3. The flow of data from XML to your code

[View full size image]



The code we show in this section is the code that is "your application" in Figure 27-3. The JAXP library has code for the other two boxes. The interface is a little more involved than simply having our class call the parser methods. This unexpected slight complication happens because the Java library implementors wanted to make absolutely sure that the installations never got locked into one particular XML parser. It's always possible to swap the parser that comes with the library for another. To retain that flexibility, we instantiate the parser object in a funny way (the Factory design pattern), which we will explain later.

The program is going to read an XML file, get it parsed, and get back the output which is a tree of Java objects that mirror and represent the XML text file. Then the program will walk the tree, printing out what it finds. We hope it will be identical with what was read. In a real application, the code would do a lot more than merely echo the data; it would process it in some fashion, extracting, comparing, summarizing. However, adding a heavyweight application would complicate the example without any benefit. So our application simply echoes what it gets. The program we are presenting here is a simplified version of an example program called DOMEcho.java that comes with the JAXP library. The general skeleton of the code is as follows.

```
// import statements

public class DOMEcho {

    main(String[] args) {
      // get a Parser from the Factory
      // Parse the file, and get back a Document
```

Reading an XML File—SAX Parsers

DOM level 1 was recommended as a standard by the World Wide Web consortium, W3C, in October 1998. In the years since then, a weakness in the DOM approach has become evident. It works fine for small and medium-sized amounts of data, up to, say, hundreds of megabytes. But DOM parsing doesn't work well for very large amounts of data, in the range of gigabytes, which cannot necessarily fit in memory at once. In addition, it can waste a lot of time to process an entire document when you know that all you need is one small element a little way into the file.

To resolve these problems, a second algorithm for XML parsing was invented. It became known as the "Simple API for XML" or "SAX," and its distinguishing characteristic is that it passes back XML elements to the calling program as it finds them. In other words, a SAX parser starts reading an XML stream, and whenever it notices a tag that starts an element, it tells the calling program. It does the same thing for closing tags too. The way a SAX parser communicates with the invoking program is via callbacks, just like event handlers for GUI programs.

The application program registers itself with the SAX parser, saying in effect "when you see one of these tags start, call this routine of mine." It is up to the application program what it does with the information. It may need to build a data structure, or add up values, or process all elements with one particular value, or whatever. For example, to search for all CDs by The Jam, you would look for all the artist elements where the PCDATA is "The Jam."

SAX parsing is very efficient with machine resources, but it also has a couple of drawbacks. The programmer has to write more code to interface to a SAX parser than to a DOM parser. Also, the programmer has to manually keep track of where he is in the parse in case the application needs this information (and that's a pretty big disadvantage). Finally, you can't "back up" to an earlier part of the document, or rearrange it, anymore than you can back up a serial data stream. You get the data as it flies by, and that's it.

The error handling for JAXP SAX and DOM applications are identical in that they share the same exceptions. The specifications require that validation errors are ignored by default. If you want to throw an exception in the event of a validation error, then you need to write a brief class that implements the org.xml.sax.ErrorHandler interface, and register it with your parser by calling the setErrorHandler() method of either javax.xml.parsers.DocumentBuilder or org.xml.sax.XMLReader. Error handling is the reason why DOM programs import classes from the org.xml.sax and org.xml.sax.helpers packages.

JAXP includes both SAX and DOM parsers. So which should you use in a given program? You will want to choose the parser with an eye on the following characteristics:

- SAX parsers are generally faster and use fewer resources, so they are a good choice for servlets and other transaction oriented requirements.
- SAX parsers require more programming effort to set them up and interact with them.
- SAX parsers are well suited to XML that contains structured data (e.g., serialized objects).
-

A Program That Uses a SAX Parser

This section walks through a code example of a SAX parser. Because we have already covered much of the background, it will seem shorter than the DOM example. Don't be fooled. The general skeleton of the code is as follows.

```
// import statements: see below

public class MySAXEcho extends DefaultSAXHandler {

    main(String[] args) {
      // get a Parser
      // register my callbacks, and parse the file
    }

    // my routines that get called back
    public void startDocument() { /*code...*/}
    public void startElement( /*code...*/
    public void characters ( /*code...*/
    public void endElement(
  /*code...*/
}
```

The first part of the program, the import statements, looks like this:

```
import java.io.*;
import org.xml.sax.*;
import org.xml.sax.helpers.DefaultHandler;
```

The Factory Design Pattern

You can safely skip this section on first reading, as it simply describes how and why you use a design pattern with the JAXP library.

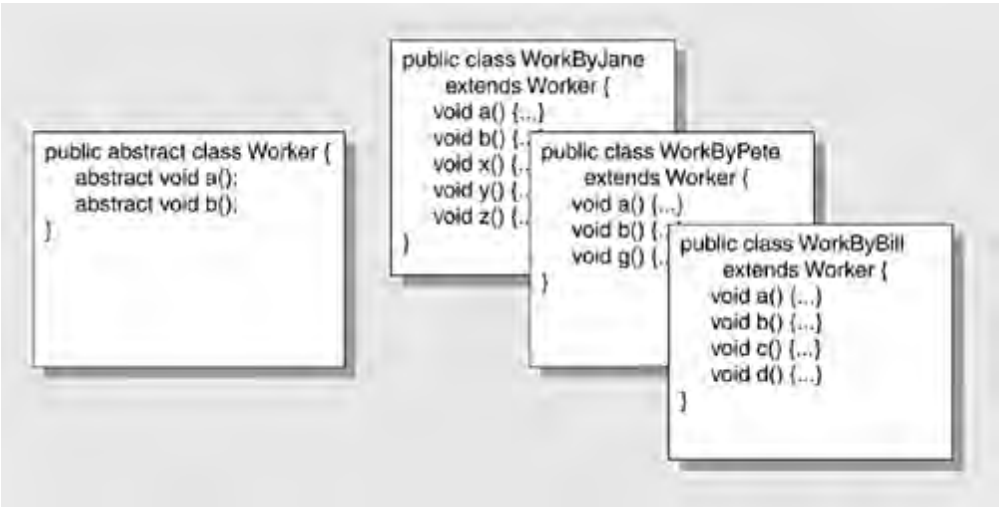
If you implement an XML parser in the most straightforward way, code that uses the parser will need implementation-specific knowledge of the parser (such as its classname). That's very undesirable. The whole XML initiative is intended to free your data from single platform lock-ins, so having your code tied to a particular parser undermines the objective. The Java API for XML Processing (JAXP) takes special steps to insulate the API from the specifics of any individual parser.

This makes the parser "pluggable," meaning you can replace the parsers that come with the library with any other compliant SAX or DOM parser. This is achieved by making sure that you never get a reference to the implementation class directly; you only ever work using an object of the interface type in the JAXP library. It's known as the "Factory" design pattern.

Factories have a simple function: churn out objects. Obviously, a Factory is not needed to make an object. A simple call to a constructor will do it for you. However, the use of Factory allows the library-writer to define an interface for creating an object, but let the Factory decide which exact class to instantiate. The Factory method allows your application classes use a general interface or abstract class, rather than a specific implementation. The interface or abstract class defines the methods that do the work. The implementation fulfills that interface, and is used by the application, but never directly seen by the application.

Figure 27-6 shows an abstract class called "Worker." Worker has exactly two methods: a() and b(). An interface can equally be used, but let's stick with an abstract class for the example. You also have some concrete classes that extend the abstract class (WorkByJane, WorkByPete, etc.). These are the different implementations that are available to you. They might differ in anything: one is fast but uses a lot of memory, another is slow but uses encryption to secure the data, a third might be able to reach remote resources.

Figure 27-6. Worker abstract class, and subclasses with different methods



[View full size image]

Design Pattern Summary

In summary, a classic Factory design pattern looks like this:

1. You have an interface or abstract class, Worker.
2. You have some implementations of that, WorkerBill, WorkerJane, WorkerFred.
3. You have a Factory that has a method, often static, often called getSomething or newSomething." It returns something of type Worker. That method chooses which of the implementors to use. It does a new WorkerBill (let's say) and returns it as the supertype.

The application code now has a concrete class, but typed as the abstract superclass or interface. It cannot use more methods than are in the interface. Voila.

Other Java XML Notes

The Document building code is not guaranteed to be well behaved in threads. You may very well have many XML files to parse, and you may want to use a thread for each. An implementation of the DocumentBuilderFactory class is not guaranteed to be thread safe. To avoid problems, the application can get a new instance of the DocumentBuilderFactory per thread, and they can be configured differently in terms of how much validation they do, whether they ignore comments and so on.

Here's how we use the Factory instance to get back a Parser that has the type of the abstract class javax.xml.parsers.DocumentBuilder:

```
... myDb = myDbf.newDocumentBuilder();
```

Now that we have a DocumentBuilder (which is actually a Pete'sPrettyGoodParser, or equivalent), we can use it in a type-safe, future-proof way to parse an XML file and build the corresponding document, like this:

```
org.w3c.dom.Document doc = myDb.parse( new File("myData.xml"));
```

We did not simply move the dependency from your code into the run-time library. The JAXP run-time library has put the hooks in place to make it possible to switch parser implementations. The full details are the in the Specification document which you will download. However, to summarize, the run-time looks for a property file that contains the class name of any different parser you want to use. If the property is not found, it uses the default. So it all works as desired. Everything is hands-off. You're manipulating the tree by remote control, which admittedly makes this harder to follow.

Factory pattern confusion in Java XML

The folks at Javasoft designed this with a double example of the factory pattern. First, you get a Factory, from that you get a ParserFactory, then you get a parser, then you parse. Even worse, they made the code more confusing by using the same class (DocumentBuilderFactory) for both Factories! The code looks like this:

Further Reading

There is a centralized portal for people developing with XML languages at www.xml.org. The website was formed in 1999 by OASIS, the non-profit Organization for the Advancement of Structured Information Systems, to provide public access to XML information and XML Schemas. You can find everything there from tutorials to case studies.

The website www.w3schools.com/dtd/default.asp has a good DTD Tutorial under the title "Welcome to DTD School." It also has links to many other tutorials of interest to XML developers.

If you're interested in getting more information on XML and the Java XML API, be aware there are a few items we haven't covered. First, there is an additional XML mark-up tag, known as a "processing instruction." This is a piece of XML inherited from SGML, but not really a good fit. (SGML is the mother of all mark-up languages, too large and too complicated to ever get much use in the real world. XML and HTML are simplifications of SGML.) A processing instruction is used to link style sheets (regular HTML CSS style sheets) into documents. Second, the topics of both elements and attributes are deeper than we have room for here. We have not covered the third library in JAXP: the transformation library in javax.xml.transform.

There is an independently developed open source library called JDOM. This is a Java API that does the same job as W3C's DOM. It was created as a more object-oriented and easier to use alternative to DOM, and there is a good possibility it will be adopted into JAXP (or even replace it). It has been adopted as Java Specification Request number 102, if you want to check on its progress. Please see the JDOM website at www.jdom.org for the most up-to-date information.

IBM offers a series of free tutorials on their website. There are tutorials covering parsers, DOM, SAX, and the transformation library. Go to www.ibm.com/developerworks and click or search on XML. You have to register at the site, but it is free and quick.

Exercises

1. Describe the Factory design pattern and state its use.
2. Write a DTD that describes a CD inventory file. Each CD is either domestic or imported. These details are stored for all CDs: artist, title, price, quantity in stock. Imported CDs also have these fields: country of origin, genre, non-discount status, language, and lead time for reorder. Write some XML instance data describing your five favorite CDs (include a couple of imported CDs, too).
3. Validate your XML file from the previous question by running it against the DOMEcho program that comes with the Java XML library. In the output you get, explain what the text nodes with a value of "[WS]" are. Hint: Try varying the number of spaces and blank lines in your instance data, and seeing how that changes the output.
4. Rewrite the DTD describing Shakespearean plays making better use of names, comments, and indenting.
5. It is possible to implement a DOM parser using a SAX parser, and vice versa, although not particularly efficiently. Write a couple of paragraphs of explanation suggesting how both of these cases might be done.
6. Write a servlet that reads an XML file of a CD inventory and sends HTML to the browser, putting the data into a table.
7. Improve the output of the SAXEcho program to make it more presentable and understandable.
8. Write an application that uses a DOM parser to get CD information and outputs the total number of all kinds of CDs that you have in stock, and the total number by each artist. Remember that some artists may have several titles in print at once.

Some Light Relief—View Source on Kevin's Life

The 5K Contest first ran in the year 2000. It's a new annual challenge for web developers and HTML gurus to create the most interesting web page in less than 5120 bytes. That's right, all HTML, scripts, image, style sheets, and any other associated files must collectively total less than 5 kilobytes in size and be entirely self-contained (no server-side processing).

The 5k competition was originally conceived in the fall of 1999 after an argument about the acceptable file size of a template for a project at work. The creator says, "It took a long time to actually get it organized because, back in those days, we all worked hard at our soul-destroying dot.com jobs and didn't have time for fun personal projects."

The 5K size limit is pretty much the only rule, and some of the entries are a bit too Zen for a meat-and-potatoes guy like me, but everyone seems to be having a good time. There is the usual crop of games written in Javascript. You've got your Space Invaders, your Maze solvers, your Game of Life. It's the International Obfuscated C Code Competition (see my text Expert C Programming), updated for the new medium and the new millennium. 3D Tetris, post modernism, poetry, art, angst—it's all there, with clever use of Javascript, style sheets, and DHTML. You can even enter an applet if you want.

One nice entry in 2001 is the Timepiece (shown in Figure 27-9). This is an animated clock showing seconds, minutes, hours, date, day-of-week, month, phase of moon, and year. You can choose the time zone.

Figure 27-9. The Timepiece in under 5K bytes.

Chapter 28. Web Services at Google and Amazon

Web services have had a lot of publicity proclaiming them the new "new thing", but they haven't yet achieved the universal use that would justify that publicity. There are several reasons behind that. One factor was the end of the Y2K upgrade spending, followed a few months later by the collapse of dot-communism. Together these led to a multi-year world-wide recession in the computer industry. Companies have been deferring new IT investments where they can.

In other words, the fact that web services aren't yet ubiquitous may have more to do with the economic climate than with the technology. But another factor is that web services often aren't secure. There is no trivial universally accepted security solution. You can prevent people snooping on the bits by encrypting them using SSL (Secure Sockets Layer, as used in https: protocols everywhere). But we also need a way to ensure that only authorized clients call the server. There are several alternatives for this (such as authentication tokens), but industry has yet to converge on one.

Regardless, organizations are now increasing their use of web services. Two of the premier websites on the internet, Google and Amazon, have launched web services Beta programs. These two initiatives are not related to each other, and they have taken slightly different technical approaches.

The goals of the two Beta programs are the same though. Amazon and Google are both predicting that web services are going to grow greatly in significance.

Amazon and Google are planting a flag in the ground, saying that they intend to be part of that growth, and help shape the future of web services. For you as a programmer, the benefit is getting early exposure to a technology that's got a great future.

In this chapter we'll give you a clear picture of what web services are, and the problems they can solve for an organization. We will show how client programs send XML requests to web services. The work that a web service server does to fulfill a request is beyond our scope here. We will look at some XML that is sent back in response, but not how it is generated or transformed. In the second section, we'll look at the Google web services beta, and walk through an example that uses it. The Google package is quite reasonable, and most programmers could figure it out for

Web Services Introduction

Chapter 27 described XML. That mark-up language lets you label pieces of data with a description of what they are. If you give an XML file to another program (including a program outside your organization) it can easily find within the file the pieces in which it is interested. Web services consist of XML files, plus a way of sending them to a server that will execute a program (probably a servlet) based on the request in the XML, and send back the results.

What are web services?

There are a lot of different perspectives on web services, but in principle, the technology is straightforward. Here is the programmer's perspective.

A web service says "Here's my URL you can post XML files to (exactly like posting an HTML form). I'll read the XML to find out what you are asking me to do, then I'll do it, and send the answer back as a web page of more XML". Instead of a browser at one end, there's a program at both ends.

Web services provide a service-oriented architecture that lets IT groups develop and deploy centrally, while supporting any platform anywhere that is connected to the 'net. Web services are likely to supplant past systems that have tried to do this (CORBA, DCOM, IIOP, RPC and, yes, Java's RMI), but which have all had interoperability limitations in talking to each other. For example, RMI only works when there is a Java system at both ends.

The problem that web services solve

If you've never worked in IT you may doubt this, but if you have worked in IT, you'll immediately recognize it as true. A significant IT expense for large organizations is the cost of moving data from one system to another. You store data for system A in a database, then requirements change and some of the data needs to be shared with System B in real time. But systems A and B are hosted on different computers, and use databases from different vendors. Just getting them to agree on the format of a calendar date is a problem. Actually moving the bulk data reliably and repeatedly is an expensive and fragile undertaking.

Some companies try to avoid the costs of data migration by moving everything into a gigantic data repository known as a data warehouse. That has some advantages and some drawbacks of its own. Other companies, like Vitria Inc., have built a substantial business out of providing software channels that reliably feed data from one system to another.

Today's browser-based web architecture has two potential improvements for business data use:

-

Google Web Services

The first step is to download the beta kit (or the actual web services package if this has turned into a product by the time you're reading this). Go to http://www.google.com/apis/ read the license and click to download the zip file.

The license makes clear this is for personal use; you're not allowed to build this into commercial products. You're also restricted to fewer than 1000 queries a day, and not more frequently than one a second. Google may well offer a different license in future. You've got to respect their trademarks and agree this is beta software which might not work. The download is less than one MB, so only takes a few seconds.

Before you can go any further, you need to create a Google account. This is really just a free registration of your email address and a password. Google will send you an email, and when you click on the link in it, you're able to proceed. Your Google account can also be used to post to Usenet through the Google servers, to access Google mail, and a couple of other things.

Google will then send you a license key, also called a client key, by email. This is a 32-character (not -bit) string of mixed case alphabetics and punctuation. This isn't a valid key, but a key will look something like this: hNpM%kKY6+k;j1hxkO3KnwQmso+/UH2g

You have to include your individual client key in all program interactions with the Google web service. That lets Google keep track of who is doing what, and selectively disable the service if necessary.

Contents of the Google Beta Kit

After you have downloaded the beta kit, unzip it. It's well behaved, and will unpack into a directory called "googleapi" that contains:

- A brief program written in Java that demonstrates how to call the web service.
- An HTML document that explains in detail the semantics of the function calls you can make using the Google Web APIs service. If you didn't already know how to filter a Google search by date-published, or only get details from one site, this will give you the inside information.
- A jar file that you link against. This jar file does all the heavy lifting of XML formulation, SOAP communication, and result parsing of the return value. Seriously, this is so easy to use, that you might get the wrong impression that all web services are that easy.
- The WSDL file that describes the Google web services in XML.
- Javadoc for the Google library contained in the jar file.

Amazon Web Services

Many of the steps in using the Amazon web services mirror those of the Google web services. There is a beta kit that you download, you have to register in order to get a key, a Java library is provided along with a sample application, and so on.

The Amazon people have not gone to quite so much trouble to hide the underlying complexities, so the process of getting code running is a bit more involved than with Google. On the other hand, you get a more realistic experience of engaging with web services. There were some bugs which prevented compilation in the beta version of the Amazon kit at the time I downloaded it. However it is possible to resolve all these and get code running.

Start by visiting http://www.amazon.com/gp/browse.html/?node=3434641 which is the download page. If this 404's, then search Google for "Amazon Web Services". When you get to the right Amazon page, notice the links on the left of that page for a FAQ, the license details, and obtaining the client key, which Amazon calls a token. Obtain your token by clicking on the link and filling in your email address and a password. Your browser will load a new page, which shows your 14-character token. This will also be emailed to the address you gave.

Web services and security

Amazon and Google are both using a featherweight security mechanism for their beta services. Sending a client key in clear text over HTTP is bad enough, but if the transport uses an HTTP GET, the key is appended to the URL. Since URLs are tracked and cached, this is like broadcasting your secret information.

When these services go fully commercial, Google and Amazon will certainly choose a more secure alternative, like SSL, Kerberos tickets, or X.509 certificates.

Although it is still in Beta, the license agreement for Amazon web services is a bit longer than Google's, because Amazon is encouraging people to deploy applications around this. Indeed, they even sell one such application, written by an early adopter in the Beta program. That developer could have been you, if you'd tackled this a few months ago!

Contents of the Amazon Beta Kit

After downloading the file kit.zip, unzip it to create a directory called "kit" that contains a directory AmazonWebServices with these contents:

- A readmefirst file describing the following information, giving advice, and telling you how to run the demo.
-

Conclusions

The computer industry has been doing electronic data interchange for years. Web services is a way of standardizing and turning it into Remote Procedure Calls, using ubiquitous web servers and XML. REST is a way of simplifying that, re-using existing HTTP protocols instead of inventing new ones.

The two giants of the Web, Amazon and Google, both concluded that UDDI and SOAP were not suitable for exposure in their Beta web services initiatives. Google layered a completely new, and much simpler custom library on top. Amazon provided thousands of lines of sample code in 35 Java files. But they also supported the REST approach which the overwhelming majority of their Beta users prefer.

Web services have always been led from in front by the evolving technology, not driven from behind by a pressing need. One result has been the profusion of industry groups rolling the frontiers forward, anticipating problems in advance of users encountering them. There are grand visions of automatically locating global services and connecting to them in real-time to consume billable services. But down on the ground, there are IT managers who just want the data from the customer repair system to flow to authorized service dealers so the dealers can order parts, and get reimbursement for warranty repairs. For these people UDDI, SOAP, and maybe WSDL is an unnecessary expense.

Web services support in MacOS X

There's only one desktop operating system in 2004 with built-in support for web services: MacOS X (pronounced "MacOS ten") from Apple.

MacOS X is BSD Unix with a first-class window system on top, and it is full of hidden treasures. It comes with DVD authoring software. It has speech recognition and speech synthesis as a standard feature (turn it on with apple/system preferences/speech). MacOS ships with Java preinstalled, and native support for Java is part of the OS. MacOS comes with the world's best http server, Apache, built right into the OS and easy to run.

You can run Apple's Sherlock program to see web services in use. It accesses commercial web services, and can do things like tell you movie showtimes in your local area. Sherlock offers a custom view into many web services like airline flights, stock prices, online auctions, language translation and more. You can easily write your own code to work with web services on MacOS X. See the apple page at www.apple.com/webobjects/web_services.html

It would be trivial to reimplement the Google web service so that it used a REST approach, and there would no longer be any complexity that would need to be hidden. The Google developers are generally regarded as clueful, so one wonders if the web services team deliberately chose SOAP for a reason like "exploring a new technology".

That question was posed on the Google developer newsgroup, and an official answer came back: "We chose to deliver

Some Light Relief—Googlewhacking

This is a great place to mention "googlewhacking", the game invented by Gary Stock (or someone else, there are several who claim to have coined the term). Googlewhacking is for people who have not just too much time on their hands, but entire clocksful of too much time on their hands. The idea is to discover a set of two words in a Google search that out of the bazillions of pages on the web, return exactly one match. Recent googlewhacks:

- ambidextrous scallywags
- squirreling dervishes
- panfish interrogation
- disenthralled nimrod
- insolvent pachyderms
- hellkite flamingo

There are only three rules:

1. Do not put your words in quotes. Quoting a string tells Google to find the words in that order next to each other, and that's just too easy.
2. Both words must be listed in the online dictionary at dictionary.reference.com
3. Online lists of words (dictionaries, glossaries, etc.) don't count as web page results.

Few of these terms stay googlewhacks for long. People put them in blogs, then the Google webcrawlers index those pages, and poof!

You can tell the world of your tremendous accomplishments by posting true googlewhacks to the website at www.googlewhack.com. There's a FAQ there, written in a delightfully sarcastic and insouciant tone, offering this justification for why three-letter abbreviations are not accepted as googlewhack terms:

The web and the 'net are not qat zek, nor pij sdo... and not all men can say pyx unp, but, yes... you may. But not now, and not for our use.

Appendix A. Downloading Java

Here are the steps to download and install the latest free Java compiler.

1. Go to the website java.sun.com

   Click on the link to "downloads".

   You want the most up-to-date version of the Java development kit. From time to time, Sun Marketing re-badges the JDK, using ever more silly and awkward names. Most recently, it has been called Java 2 Platform, Standard Edition (J2SE) Software Development Kit (SDK). In summer 2004, the most up-to-date version was 1.5 (yes, "Java 2" is at version 1.5).

2. You want the SDK (the compiler, tools, and run-time library). The JRE is just the part of the SDK needed to run Java programs, but not the development tools (the JRE is a smaller download intended for users). Click through the license (it says you can use the tools for free, but you're not to use them to build any nuclear reactors).

3. You reach a page where you choose which platform on which you will be running the compiler. Wherever you choose to compile, your executable programs will run on all platforms.

   Sun distributes compilers for Solaris, Linux, and Windows. Solaris has been a 64-bit operating system since we re-architected the kernel for Solaris 7 in 1998. Almost everyone else is still running 32-bit versions of Linux and Windows so choose one of those, unless you installed 64-bit Linux or Windows.

   If you are running something other than Solaris, Linux, or Windows, go to the home page of the manufacturer and search for their Java download. MacOS X comes with the JDK preinstalled, but new releases of Java may come out after you buy your Apple, so check the Apple website for the latest.

4. For Windows, Sun offers an "offline" installation and an "online" one. The "offline" installation gives you the entire 50MB release in one download. (It's so big because it also installs the NetBeans IDE, the Java browser plug-in, and Java webstart). Use the offline download if you have a reliable broadband service. The result is a file that you execute to do the installation.

   The "online" download is a small (under 1MB) download of a program. When you run it, it does the real download and installation. It's restartable, if the connection is lost. There's a FAQ of common issues/answers at the download web page.

5. When the installation wizard runs, accept all defaults and optional features (demos, source code, etc.). If you choose a different installation directory, write down where! You need that pathname (plus "\bin" added to the end) in step 7. For Windows, the default install is in:

Now it's Hello World

Java programs are made up of code organized into classes. If you are not using an IDE, the classes go into java source files. Important classes should go in a file of the same name as the class. For example, a class called "hello" should go in a file called "hello.java". Here is the source you can put in such a file:

```
public class hello {

    public static void main(String[] args) {

        System.out.println("hello sailor");

    }

}
```

You compile that with the command:

```
javac hello.java
```

That compilation will create a file called hello.class. It contains the executable program binary. You can run it with the command:

```
java hello
```

Note you use the name of the class, not the name of the file. Try it now. Then, on to the big picture of Java!

Appendix B. Powers of Two Table

Refer to Table B-1 for powers of two.

Table B-1. Powers-of-Two from 21 to 264

21	2	217	131,072	233	8,589,934,592	249	562,949,953,421,312
22	4	218	262,144	234	17,179,869,184	250	1,125,899,906,842,624
23	8	219	524,288	235	34,359,738,368	251	2,251,799,813,685,248
24	16	220 megabyte	1,048,576	236	68,719,476,736	252	4,503,599,627,370,496
25	32	221	2,097,152	237	137,438,953,472	253	9,007,199,254,740,992
26	64	222	4,194,304	238	274,877,906,944	254	18,014,398,509,481,984
27	128	223	8,388,608	239	549,755,813,888	255	36,028,797,018,963,968
28	256	224	16,777,216	240 terabyte	1,099,511,627,776	256	72,057,594,037,927,936
29	512	225	33,554,432	241	2,199,023,255,552	257	144,115,188,075,855,872
210 kilobyte	1,024	226	67,108,864	242	4,398,046,511,104	258	288,230,376,151,711,744
211	2,048	227	134,217,728	243	8,796,093,022,208	259	576,460,752,303,423,488
212	4,096	228	268,435,456	244	17,592,186,	260	1,152,921,5

Appendix C. Codesets

Table C-1 shows 256 decimal and hexadecimal codes that represent characters and control characters in different codesets. The codesets shown are:

- ISO 8859 Latin-1 codeset. ISO 8859-1 is identical to the first 256 Unicode characters.
- ANSI_X3.4-1968 ASCII. The ASCII characters are a subset of the 8859-1 code, and codes within the ASCII range are indicated by shading in the leftmost column in the table below.
- IBM's nearly obsolete EBCDIC codeset.

Table C-1. ISO 8859-1, ASCII and EBCDIC codes

ISO 8859-1 and ASCII		**Dec**	**Hex**	**EBCDIC**	
NUL	Null	0	00	Null	NUL
SOH	Start of Heading (CC)	1	01	Start of Heading	SOH
STX	Start of Text (CC)	2	02	Start of Text	STX
ETX	End of Text (CC)	3	03	End of Text	ETX
EOT	End of Transmission (CC)	4	04	Punch Off	PF
ENQ	Enquiry (CC)	5	05	Horizontal Tab	HT
ACK	Acknowledge (CC)	6	06	Lower Case	LC
BEL	Bell	7	07	Delete	DEL
BS	Backspace (FE)	8	08		