JAVA GARAGE

By Eben Hewitt

Publisher: Prentice Hall PTR
Pub Date: August 12, 2004
ISBN: 0-321-24623-3
Pages: 480

Enter your *Java Garage*... where you do your work, not somebody else's. It's where you experiment, escape, tinker, and ultimately succeed.

Java Garage is not your typical Java book. If you're tired of monotonous "feature walks" and dull tutorials, put down those other Java books and pick up *Java Garage*. Java guru Eben Hewitt takes a fresh look at this popular programming language, providing the insight and guidance to turn the regular programmer into a master. The style is straightforward, thought-provoking and occasionally irreverent.

You'll learn the best ways to program with everything that matters: J2SE 5.0 classes, inheritance, interfaces, type conversions, event handling, exceptions, file I/O, multithreading, inner classes, Swing, JARs, etc. Hewitt provides real working code and instructions for making usable applications that you can exploit and incorporate into your own personal projects with ease. Need answers quickly? The book also includes FAQs for speedy reference and a glossary on steroids that gives you the context, not just the definition.

With *Java Garage*, you'll learn the best way to create and finish projects with finesse. Think 'zine. Think blog. But, please, do not think of any other Java book you have ever seen.

- Table of Contents
- Index

JAVA GARAGE

By Eben Hewitt

Publisher: Prentice Hall PTR
Pub Date: August 12, 2004
ISBN: 0-321-24623-3
Pages: 480

PREV < Day Day Up > NEXT

Copyright

Library of Congress Cataloging-in-Publication Data

A CIP catalog record for this book can be obtained from the Library of Congress

Acquisitions Editor: John Neidhart

Editorial Assistant: Raquel Kaplan

Marketing Manager: Stephane Nakib

Marketing Specialist: Jen Lundberg

Publicity: Kerry Guiliano

Managing Editor: Gina Kanouse

Project Editor: Michael Thurston

Cover Design: Anthony Gemmellaro

Interior Design: Wanda Espana

Manufacturing Buyer: Dan Uhrig

Dedication

dedication.bas

10 PRINT "For Alison, of course"

20 GOTO 10

RUN

ACKNOWLEDGMENTS

I am grateful for the help of many people during the making of this book.

First, I must thank my editor, John Neidhart (Iron Chef Garage), for being a standup guy through the many labyrinthine conversations that helped shape this new series. I appreciate your high-availability, ease of maintenance, and extensibility. Here's to more sushi at Bond Street.

I am grateful not only to John, but to the many good and hard-working people at Prentice Hall for the opportunity to work on this series. Thank you to production editor Michael Thurston, and Kelli Brooks who worked very hard to root out the errors in this book and bring it all together. Special thanks to Anthony Gemmellaro and the entire production team for the stunningly cool artwork and interior. I am also grateful to the good people of the Salt River Prima-Maricopa Indian Community; may you live long and prosper.

The efforts of my technical reviewers were tremendous. Thank you to Pawel Zurek, a terrific software developer with an eagle eye. I appreciate very much the careful and insightful comments of J. Benton: it's really fun to talk with you about searching, and planning, and algorithms (all important things in life). Thank you also to Vic Miller for your terrific work in technically reviewing this book. You caught a number of errors; thanks for helping me develop with pleasure.

I am a lucky sonofagun to be married to Alison Brown, my favorite thing in this world. She has also been very helpful in shaping this series. Thank you, Alison, for your smashing ideas, perseverance, and hard work—and for helping keep me monotonously on-message.

Eben Hewitt
July 2004
Scottsdale, Arizona

ABOUT THE AUTHOR

Eben Hewitt

Eben is a Sun Certified Java Programmer, Sun Certified Web Component Developer, Sun Certified Java Developer, and Senior Java Programmer/Analyst for Discount Tire Company in Scottsdale, AZ. He is the Garage Series Editor, and the author of four other programming books, including the acclaimed *Java for ColdFusion Developers*, and the forthcoming *More Java Garage*. He has been an invited speaker on Java at regional user groups.

Eben has a Master's Degree in Literary Theory, and had his first full-length, original play produced in New York City in 1996. Questions haunt him in this Champagne-light and Orwellian time: Who will win, Struts or Faces? What if they trade Johnson to the Yankees? At long last, have we left no sense of decency?

He and his family, Alison, Zoë, Mister Apache Tomcat, Doodle, and Noodle, burn to ashes every summer.

Chapter 1. WHAT IS THE JAVA GARAGE

[View full size image]

```
login as: eben
Sent username "eben"
eben@www.thegarage.net's password:
[eben@dev eben]$ what is the garage?...
```

Programming on the Internet, in Internet time, is a collaboration. There are a million sites out there to find out how to write a little snippet of code. Go to Google and search for C# Custom Cursor, and up come 7,000 web sites to tell you how to change the appearance of the mouse pointer in C# for .NET. So you don't need me, do you smart guy?

It works once you know what you're doing. It'd be a rough ride learning that way.

You came here. You wanted something. You felt it your whole life. That tech books could be something more.

The garage is a place to go to be alone. To be with your friends. It's an accessory. It's a style, a way of life, it's a friend, it's a companion, it's an IRC server, it's what a book would be if a book could be a wiki with exactly what you need and nothing you don't. And no banner ads.

Says the Oracle: "How could our jobs be so fantastic, so filled with art and innovation and dripping with collaboration and speed and the thrill of making it work—and yet our books be so dull and dusty and cardboard and boring and in a format that's essentially unchanged for more than 35 years?"

You came here to learn what you need to know about Java until everything turns green.

So let the garage be green.

Our work and our play are so interweaved they are indistinguishable. This is how we think now. There is no such thing as the weekend. Luckily for us, our work is fun. It is an absolute delight. IT is like Wonka's factory for smart people. Can our books be that? Force the boring writers to listen to Yoko Ono nonstop until they relent in their boringness.

And the garage god said, "Make It So."

In the garage you do your projects. On your time.

So our garage has a toolkit, which means you got a load of code to work with, and it's well documented so you understand why it works the way it does. Cuz that helps you remember it next time. And there are FAQs to look up quick deals that you need to use in your daily job. And a glossary on steriods that doesn't just point to a pointer of a pointer of a pointer, but tells you the whole definition, and gives you a context.

Working in IT happens with windows open and processes running and the ftp here and the zone file there and the Photoshop and your IDE all open and going and getting it done. Show me the Java programmer who doesn't need to know anything about html and networks and naming and sql and xml and administration and uml. A book about Java isn't written for any human; it's written to be Da Word. I reject the word. That's not how I work, or how anyone I've ever met in IT works. We are situated in a context with a lot of technologies that work together. Perhaps our books can be like this?

And thus the Architect wrote it.

Garages are all different. I live in the hot and hollow, barenaked desert of the American southwest. Sometimes scorpions find their way into my garage. And rattlers sometimes, and thick brown spiders. Another garage will be different.

Because it is different, each garage has a blog, where the writer goes to rant. This keeps everybody honestpuck. It makes sure there is a human voice. Selling something isn't allowed. We're just going to talk for a while. And at the end of the conversation, you'll see green.

Cuz in a garage, you bang it around until it works.

So each garage has real, working code, and instructions like recipes for making usable, neatokeen applications that you can incorporate into your own projects. Cuz that's the point. Too bad how lotsa programmer book guys eject from the cockpit at the very last second, leaving the real work as "an exercise for the reader." In the garage, we don't get much of that kind of "exercise." Maybe just lifting a beer once or twice. When Willy Wonka is your role model, you should be able to eat everything.

Lick the pages. They taste like schnozberrys….

The Garage is where you go on Saturday. You hack it out when you get a second. So none of this long claptrap with 5,000 pages, and I'm breaking my wrist carrying that sheet to work, and all I need is for my wrist to get jiggy with the exchange server come 5 p.m., if you know what I'm sayin'. Short, focused, recipe deals that give you the how and the why. Uh, I guess we won't be doing nothing like, "The All-Time History of Programming blah blah and then dude invented fire and then blah blah and then came Linux." How about instead, "Retrieving User Input from the Console." Then, boom. How to do it. Why it works. How to integrate it. Done.

The garage is a place. I don't know where you're from, but my garage don't really go in a straight line. It's more like I hang out there til the game comes on.

Now. This is our Java Garage.

The keymaker says, "That door will take you home…".

Chapter 2. JAVA BUZZ

D00dle: yt?

Zoomz: y

D00dle: I was thinking about this cuz I was walking by the Taco Tico and I stand there getting tortured by Avril Lavigne and I'm all, All I wants this stoopid burrito and you can't give it a me without I gotta have this claptrap making my ears bleed.

D00dle: stop whining dude

Zoomz: no so I was thinking just like this: if i was in a band called the Yeah-Yeahs all my songs would go "Yeah Yeah Yeah"

Zoomz: dude I would SO buy your album.

D00dle: thanks!

D00dle: you do java rite?

Zoomz: ?

D00dle: what's the story?

D00dle: we got a new cto loves unix esp freebsd

Zoomz: does he hate m$?

D00dle: duh

Zoomz: thats like so late nineties.

Zoomz: is he cute

D00dle: as if. how do you know its a he

Zoomz: does that mean you have to port all your apps

D00dle: yeah18 mos work on the milk lady's project dead.

Zoomz: so you're learning java

D00dle: y

Zoomz: so read a book knob

D00dle: i hate books punk just tell me why thiss guys all hopped up about java

Zoomz: dude I got like 10 words for you:

Simple

OO

Network-ready

Robust

Secure

Architecture Neutral

Portable

Interpreted

Fast

Multithreaded

Dynamic

D00dl3: LOL! that's all marketing dude. i can't believe you would actually write that to me. Do you hate me?

Zoomz: that is just true.

D00dl3: every company says their thing is secure my grandma is secure.

Zoomz: did you ever hear of a virus written in java?

D00dl3: no.

Zoomz: Did you ever hear of a virus exploiting a java application?

D00dl3: no.

Zoomz: well then. would you be in this port trouble today if you had written in java originally?

D00dl3: no. but what about fast, robust, simple, and dynamic? that's where you really sink low bro.

Zoomz: you're right—that's all marketing hoo-hoo. not that it isn't true, but like so is C++ C# VB C Ada Eiffel Lisp Delphi

D00dl3: You don't have to convince me. I can learn it or lose my job to some 14 year old whippersnapper. How did i get to feel so old and i'm not even 25? I need to learn it and i want to learn it because i know it is an important language now so it doesnt matter. So just how do i do that so i can get back to Never Winter Nights?

Zoomz: read java garage.

D00dl3: i hate books.

Zoomz: its not a book: its garage...

Chapter 3. JAVA EDITIONS AND PLATFORMS

Sort out the different Java editions without losing your mind.

Dear Sirs and Madams,

I just want to programming with Java. Yet, I cannot start. How can I do this?

Thank you,

xaolin

do you have the JDK?? go to java.sun.com and download the JDK.

^_^ zilly

Dear Mister Zilly,

Everything I have seen assumes that you already know this. I do not know this. Please: what is JDK?

Xaolin

am not *mister* zilly i am devo d-e-v-o. now. Java Devlopment Kit. That is the compiler, to compile source code into executable code, and a runtime, to run it. And other stuff.

^_^ zilly

Dear Zilly,

I go to the download my thought is: what is the difference between each of different editions?

Which do I get? I prefer to have only the best.

Many courtesies,

Xaolin

Hi Xaolin,

There isnt really a best one, the different editions arent like more powerful or something they dont have superpowers like but that would be cool. Here is the difference between those three J2SE, J2EE and J2ME:

J2SE: this is Java 2 Standard Edition. Download this one. It is the one you want to get started, as it is the most frequently used, and includes a compiler and a runtime so you can write Java applications. Heres what you can do with it:

- Use Swing to create full-blown Graphical User Interface applications.
- Write applets, which are small programs that run inside a Web browser.
- Connect to remote computers and write Client/Server applications with sockets.
- Work with the local file system.
- Parse text with regular expressions.
- Read, write, and transform XML.
- Execute code in a remote Java Virtual Machine using Remote Method Invocation.
- and oh so much more.

The other ones are J2ME and J2EE.

J2ME is Java 2 Micro Edition. This edition is geared toward applications embedded in consumer products such as cell phones, PDAs, set top boxes, smart cards, and car navigation systems. One thing that distinguishes J2ME from the other editions is that it is the only one that is subdivided into profiles and configurations. A profile defines libraries for different platforms and the JVM requirements for supporting a particular micro platform (for example, there is one profile for PDAs and another for wireless devices). A configuration is composed of a virtual machine and a small set of libraries that provide base functionality to devices. Currently there are two configurations: Connected Limited Device Configuration and Connected Device Configuration. A runtime is included, which takes a very small footprint. The smart card runtime in the Card Virtual Machine, for example, consumes only 128K of memory.

J2EE this is Java 2 Enterprise Edition. It adds significant functionality to the Standard Edition. It includes support for the following technologies:

- JavaServer Pages and servlets allow developers to create dynamic server-side Java applications.
- Enterprise Java Beans, which are components that support transactions, security
- JDBC (Java Database Connectivity) allows developers to invoke Structured Query Language operations from Java code to interact with databases.
- JavaMail lets you work with email messages
- Java Message Service (JMS) allows the distributed, asynchronous posting and retrieval of message objects.
- JNDI is the Java Naming and Directory Interface, which lets you perform typical directory operations. But JNDI is not tied to any particular implementation, so it can be used to interact with established services such as LDAP, DNS, and NIS.
- Java Authentication and Authorization Service (JAAS), which is used to determine if users and groups are allowed to enter a system (authentication), and what specific tasks they are allowed to perform once allowed in (authorization).
- JAX-RPC: the Java API for XML-based Remote Procedure Call. This allows you to develop SOAP-based Web Services clients and endpoints.

j2ee apps are typically server-based applications that use a combination of these technologies to get their jobs done. But all you need for a long time is the J2SE.

^_^ zilly

Dear Zilly,

Thank you for this kind response. I must warn you that this j2ee is not acceptable. I must have connect to database for my programming, yet regretfully I am not for using j2ee. What is it to do?

Xaolin

DUDE

you can use the services that j2ee makes available even if you are just doing a standard j2SE app. later, we can talk about JDBC and how to do that. All of them are free. So you dont have to pay to get the language or to write programs in the language which is sweeeeeeeet.

^_^ zilly

Dear Zilly,

I am not pleased to bother you. I confuse again: You refer to Java 2. And yet, these versions are not Java 2, but Java 5.0. Is there somewhere else I am supposed to look?

In Modesty,

Xaolin

Xaolin,

That is a very good question. You are in the right place. There is no such thing as Java 2.0 or 3.0 or 4.0. The most recent version is 1.5 following 1.4 and 1.3, etc. At the 2004 JavaOne defeloper conference, 1.5 was renamed to Java 5.0. This is a little like how Sun came up with Java 2. Java 1.2 contained significant changes to how graphical user interface elements are rendered in desktop applications with the addition of the Swing libraries (or Java Foundation Classes). Three days after they released this version as 1.2, Sun thought that the change was so significant that they should start referring to it as Java 2. But 1.2 already had a foothold in the market, so both names stuck. As a consequence, Java 2 still refers to subsequent releases, such as 1.3, 1.4, and 1.5. So you will see a lot of things referrring to Java 1.5 or Java 5.0—hey, refer to the same version. Ack.

^_^ z

Dear Zilly,

Is there a difference between SDK and JDK? Which should I get? You mentioned JDK in an earlier post.

Thank you and blessings,

Xaolin

Xaolin,

Dont worry about that other little sleight of hand: the two terms refer to the same thing. The Java Development Kit (JDK) was simply renamed the Software Development Kit (SDK) in November of 1999. Maybe it sounds more universal that way.

^_^ zilly

Dear Zilly,

I apologize, but I go to download it and again a problem. There are two links next to every Operating System: SDK and JRE. What is this now? Which one should I get? I thought that Java was cross-platform—yet there are one SDK and JRE for each of these OS: Windows, Linux, Solaris SPARC processor and Solaris x86 processor. Which one do I get? Why isn't Java cross platform now?

Cheerfully but with regrets,

Xaolin

Hey,

First, the cross-platform matter. Java applications that you write are cross-platform. Sun likes to say Write Once, Run Anywhere. That means that you can execute the same Java code on any machine that has a Java runtime available. Which means, for now, Linux, Solaris and Windows.

Your Java applications use the runtime to execute in—but the runtime itself, which is written in native code, is dependent on a particular platforms architecture. Hence the different options.

There are, however, a number of ports of the Java SDK and/or JRE that make Java available on the following systems, though these ports are not produced by Sun:

- Mac OS
- Tru64 Unix
- SCO
- AIX
- HP-UX
- NetWare
- IRIX
- NonStop
- OS/2, OS/390, OS/400
- VxWorks
- NetBSD
- FreeBSD
- Reliant Unix

These different systems are not tested or maintained by Sun. Also check out http://www.blackdown.org/java-linux/ports.html for more information on current port status (Blackdown was the first team to port Java to Linux).

The FreeBSD Unix-based operating system has just recently made an arrangement with Sun to distribute FreeBSD binaries for the JRE and SDK. The FreeBSD Foundation refers to their JRE as Latte Diablo and the SDK is called Caffe Diablo. So you can develop and execute Java programs on FreeBSD but, for now, you have to get it from them. You can do so at: http://www.freebsdfoundation.org/downloads/java.shtml.

The Linux distros include:

- Red Hat7.3 and later
- SuSE 8.0 and later
- TurboLinux 7.0
- SLEC 8

Most Java applications should work fine on Debian, but check the excellent FAQ at http://www.debian.org/doc/manuals/debian-java-faq/ for details.

Bye xaolin i have to go my mom is taking me to my trumpet lesson now :P

^_^ zilly

Chapter 4. COMPILING AND RUNNING JAVA APPLICATIONS

Installing the SDK

First you have to get it. Visit http://java.sun.com/j2se/1.5.0/download.jsp.

Accept the license agreement and download the SDK. Make sure that as you follow the instructions below, you change the file names as necessary for the exact distribution you get.

On Windows

Just run the executable installer.

Open a command prompt. If you can type "java -version" and see something meaningful, you're good to go.

If not, read on.

On Linux

Get the RPM binary for Linux. You have to give the j2sdk* file permission to execute.

```
$ chmod a+x j2sdk-1_5_0-linux-i586-rpm.bin
```

Now execute the file.

```
$ ./j2sdk-1_5_0-linux-i586-rpm.bin
```

Agree to the license.

Change to super user:

```
$su root

Password: ******
```

To install it if a previous version of Java is not already installed, type

```
$rpm -iv j2sdk-1_5_0-fcs-linux-i586-rpm
```

If Java is already installed and you are updating to the new version, type

```
rpm -Uv j2sdk-1_5_0-fcs-linux-i586-rpm
```

Ensure that everything went all right by typing

```
/usr/java/j2sdk1.5.0/bin/java -version"
```

This executes the java program, passing it the "-version" flag. If all has gone well, you will see version information indicating that you are running Java 5.0 or Java 1.5.0.

Now you will need to set your path and classpath.

FRIDGE

You might need to set your path and your classpath before you are able to compile, especially if you aren't writing your code in an IDE. See "Setting the Classpath" in this chapter for details on how to do that.

This section covers how to compile Java source code using the tools distributed with the Sun J2SE 5.0 (a.k.a. 1.5).

Compiling Source Code

To compile Java programs after you have installed the SDK, open a console. In Windows, go to Start > Run and type cmd to get a command prompt.

Note that if you have multiple JDKs on your system, you may need to supply the –source 1.5 flag to the javac command to ensure that your code is compiled correctly.

```
C:\garage\src>javac -source 1.5

    net\javagarage\demo\MyProgram.java
```

This compiles the Java source code into a class file in the same directory.

If all goes well during compilation, you get a new command prompt, and nothing else. You are now ready to run your program using the java command.

Compiling into a Directory Other Than Your Source Directory

It is common practice, and it's a good thing, to keep your source files in one directory and your class files in another. Typically, when you start a project you do something like this:

```
/application root dir

/application root dir/src

/application root dir/classes
```

You create two directories under the root of your application. Put all of your source files under the src directory and compile such that your classes go into the classes directory. Then you can use the JAR archiving tool to package just your classes and deploy those when your application is done.

By default, the javac tool compiles your classes into the same directory that the source file is in. So, you need to pass a command to the compiler to handle this. If you use an IDE such as Eclipse, you can almost always set the compiler output path in a project properties window.

To compile to a directory other than the one your source is in, use the -d flag, followed by the name of the directory you want the .class files to end up in.

```
C:\j15\net\javagarage\demo\swing\

    layouts>javac -source 1.5 -d

C:\j15\classes GridLayoutExample.java
```

The preceding example takes the file GridLayoutExample.java and compiles it into a class called GridLayoutExample.class under the C:\j15\classes directory with its packages intact. In other words, if the GridLayoutExample class is in a package called net.javagarage.swing.demo, you can find the class file in C:\j15\classes\net\javagarage\swing\demo.

Running Programs

After you have compiled your source files into bytecode using the javac command, you are ready to execute them. To do so, you need to follow these directions.

Navigate to the directory that stores your top-level package. For the examples in this book, that is a directory called C:\garage\classes. If you are using an IDE, your classes directory might be different. For example, an Eclipse project might store files at C:\eclipse\workspace\garage\classes. In any case, it is the directory that contains your packages, and where you specify the compiler to output code.

Run the java command, passing it the fully qualified name of your class (including all of the package names).

Make sure to use the . separator for packages—not your system file separator (/ on Linux and \ on Windows). This is a common mistake, because you have to supply the file separator when compiling.

For example, here is how to run the CommonFileTasks.class program:

```
C:\garage\classes>java net.javagarage.demo.MyProgram
```

Make sure that you leave off the .class extension. That can be hard to remember because you have to include the .java extension when compiling.

Make sure that the class you invoke on the command line is the one that contains your public static void main(String...args) method. This is the method that is called when the JVM starts your program, and it will tell you if it cannot be found.

If you see this error,

```
Exception in thread "main"

    java.lang.NoClassDefFoundError:

MyProgram
```

it means that the java runtime program cannot find your bytecode file, MyProgram.class.

The Java runtime will look for your bytecode file in the present working directory. So if your .class file is in C:\garage\classes, change to that directory using the cd <dirname> command on Windows and Linux. This is often not convenient, however. To remedy the problem, you can set the classpath system variable.

Setting the Classpath

If java is still unable to find your program, you might have to change your CLASSPATH variable. The CLASSPATH variable is where the java program looks to find classes to include when it executes.

In your CLASSPATH setting, make sure that . is included, to indicate the present working directory, as well as any custom directory you make to store class files, such as C:\garage\classes.

In Windows 2000, set the CLASSPATH like this:

1. Click Start > Settings > Control Panel > System > Advanced Tab > Environment Variables.
2. Find the one called CLASSPATH (or make it if it doesn't exist) and add the preceding directories to it.
3. Create this variable as a system variable if it does not exist. Also, make sure that the JAVA_HOME location correctly points to the top-level directory of your Java installation. This is probably C:\Program Files\Java\J2SDK1.5.0 on Windows.

Setting the Classpath at Compile Time

You can set the classpath at compile time. This is useful if you have a quick class you want to try out, and it relies on a library or other class that is not currently on your classpath. You do so using the -cp flag. You can also use -classpath if you are exceptionally fond of typing.

Try it like this:

```
javac -cp C:\\garage\classes SomeClass.java
```

Note that you can combine flags as well:

```
javac -source 1.5 -cp

    /user/eben/garage/SomeClass.java
```

Setting the Path

Setting the Path variable allows you to run the Java SDK executable programs, such as java and javac, from any directory on your system. It is a very good idea to set this variable, for convenience. If you do not set the Path variable, you need to specify the location of the javac compiler and java runtime every time you try to compile and run a program. Here's how to set it.

On Windows

In Windows 2000, set the Path like this: click Start > Settings > Control Panel > System > Advanced Tab > Environment Variables. Create a variable called path and set its value to the location of the <J2SDK-install-location>\bin directory. For example, it might be this:

```
C:\Program Files\Java\J2SDK1.5.0\bin
```

The Path can be a series of directories separated by semi-colons (;). Windows looks for programs in the Path directories from left to right.

On Linux

Linux uses a : to separate path variables. You can set the path on Linux by editing your /etc/profile file (su root first) to the location of your Java installation. Your installation may already have a JDK on it (RedHat and many others come with the Java SDK). So just type java at a terminal and see if you get usage information. If you do, you can use it. But this is likely to be a 1.4 JDK for some time.

On Mac OSX

Mac OS version 10.3 comes pre-installed with a Java SDK, which at this writing is version 1.4.1_03. While many of the examples in this book will work with that version, some important syntax has been added, and you probably will want to upgrade. To do so, just go to Finder and getting a terminal window—then you can follow the Linux instructions since Mac OSX is a mix of BSD distros under the hood.

There should be only one bin directory for a Java SDK in the path at a time. Any directory specified subsequent to the first is ignored. So if there is already a PATH variable set (say, because of an IDE or previous version), you can update it to j2sdk1.5.0\bin.

You can verify that this is working correctly by typing the java command at a prompt like this:

```
C:\garage>java -version
```

It should output version information similar to the following:

```
java version "1.5.0"

Java(TM) 2 Runtime Environment, Standard Edition

    (build 1.5.0)

Java HotSpot(TM) Client VM (build 1.5.0-32c,

    mixed mode)
```

You should now be able to compile Java source code and execute Java programs from any directory in your system. Note that if you use an IDE, you might need to make sure that it is pointing to your 1.5 JDK, and not a separate JDK that it ships with, which could be a different version that's incompatible with some examples in this book. For more information on the different tools that come with the SDK, see the SDK tools section at the back of the garage.

PREV < Day Day Up > NEXT

Chapter 5. WHERE TO WRITE CODE

DO OR DIE:

- Get some objectives
- Write them here
- Write about those objectives in the space provided below

This whole part is about where to write code. You can write your code in a plain text editor like vi or emacs or Notepad, or you can use a program that is specially suited to the task.

Integrated Development Environments

To make a Java program, you write Java source code, which gets compiled into class files, which are composed of bytecodes, which are then executed by the Java Runtime Environment when you run your program. That's a number of steps. The more complex your program, the more external details are involved in completing the system. It can be difficult to maintain two or three or four hundred class files for each project you're working on. It gets tiring repeating lengthy classpath arguments when testing your code. Perhaps you are working with other developers and would like to coordinate your efforts by using a common code repository. Or perhaps, being new to the language, you would like to have an environment in which to write code that gives you hints about what you are doing.

All of the above reasons are good reasons to take a little time, and perhaps money, to get a good Integrated Development Environment.

IDEs are very good at organizing projects. But unless you take a non-trivial amount of time learning the IDE itself, you won't find much more benefit to using an IDE than using some text editor and a shell script or Ant build. That being said, there are a number of IDEs that you can get for free or buy.

Friend, here's a list of some good, free Java IDEs:

- **Eclipse**. Available from http://www.eclipse.org. This is the one I use at work. The performance is improving, and this one will be really cool once it is integrated with IBM Rational tools in the form of XDE. There are a lot of plugins now available for Eclipse; you can write EJBs and integrate with JBoss, Tomcat, JUnit, Ant, and many other tools with ease. Its extensibility makes it very attractive. A notable drag about using it is its lean XML support.
- **Sun ONE Studio 5**. Available from http://wwws.sun.com/software/sundev/jde/. Sun bought NetBeans, and this is their Java IDE. This one has a free version and another version that is a couple thousand dollars. A new Sun IDE is stepping up to the plate in the form of Java Studio Creator, which looks and acts a little more like the Microsoft IDE Visual Studio. But for now, that IDE is meant to support JavaServerFaces programming for Web stuff, which we aren't discussing in this book. But it is a groovy tool, and you can read more about it here: http://wwws.sun.com/software/products/jscreator/.
- **Jext**. Available from http://www.jext.org/. This one is really more of a code editor, but it supports a lot of languages, and I like it. This one will soon become part of the Debian Linux distro.
- **Jcreator**. Available from http://www.jcreator.com/. Comes in Lite and Pro Editions.
- **BlueJ**. Available from http://www.bluej.org/. Good, simple tool for learning with visual aids.

Some good Java IDEs that aren't free:

- **IntelliJ**. Available from http://www.intellij.com. This is what many developers used at Macromedia to write the JRun 4 JSP, servlet, and EJB app server. Perhaps with less glamor, I used IntelliJ at my last job. I loved it: it works great, is integrated with Ant and CVS, and provides easy EJBs and plugin facilities. This is not integrated with any deployment environment, which could be good or bad depending on where you run stuff. If you have Apache Web Server and Tomcat and JBoss for EJBs, you're just fine. If you have Borland App Server, you're just fine too. But you don't get tools specific to that environment.
- **Jbuilder**. Available in Personal, Developer, and Enterprise editions from http://www.borland.com/jbuilder/. There are some very nice code generation facilities in JBuilder, but that's not the best way to learn in my view, and the price tag can be steep. Not available for Mac OS or HP-UX. Heavily integrated with Borland's other tools.

You can write Java source code in any plain text editor. You don't need an IDE, but it is pretty silly to think that you can do a project of any matter without one. If you can't decide, get Eclipse or NetBeans. If you want an excuse to go shopping, need an easy way to create desktop apps without writing Swing code, and aren't worried about proprietary JARs in your app, get IntelliJ. You'll probably use what you need to use as dictated by your current or prospective employer.

In this book, and at my work, I use Eclipse. But it doesn't matter. If you want to know exactly what it is that an IDE does that people think they can charge thousands of dollars for, we can talk about that for a second.

What I Hate About IDEs in 50 Words or Less

IDEs are great when you're learning, and great after you've learned a lot. Don't spend your time right now learning some IDE; spend your time learning Java. That might mean typing things like `public static void main(String...args)` a bunch instead of checking a box in a wizard. That's a Good Thing for now.

What IDEs Are Good at

I won't tell you how much easier your life will be if you get this or that IDE. You hear that a lot in tech books, about how such and such feature is going to make your life so much easier. That always makes me nuts. The fact is, until you learn the IDE itself, and especially until you learn Java, your life will get worse. Way worse. That's because it is hard work to learn something new of any complexity. More often than not, you've got a boss breathing down your neck, and you need to get going fast, and computer languages are complicated.

And with an IDE, you've really got two things to learn: getting around in the IDE, and the language you're interested in learning. And it is a bad idea to start learning some IDE before learning the language in question (in this case, Java). It's a bad idea because it is nonportable. You can't take your tremendous knowledge of JBuilder with you to IntelliJ (for instance). And one thing they never do is help you write well-designed, slim code. You need to at least know how to run java, javac, and set up your classpath before using an IDE. If you don't, you'll be forever on crutches.

On the other hand, you're going to use one because that's what all the cool kids do, and it is simply idiotic to try a project of any complexity without one. Most importantly, many IDEs do have terrific features that eventually are useful, including debuggers, code generators, Java-specific project organizers, and so forth. So, to make us feel a little less cynical about ponying up and clicking the download link, let's look at some of the many real benefits we get with most IDEs:

- **Debugging**. This is arguably the best reason to get involved in a relationship with an IDE. It can be very daunting to debug programs while you are learning, and debuggers improve your chances of working through problems quickly. You can specify a point to stop in the middle of the execution of your program, then step line-by-line through the code as it executes, and see all of the current values of all of your variables at each step. That is very helpful to see exactly where code is failing, and why. Using a debugger is not an immediately intuitive thing, but after you get a handle on it, this skill is more or less transferable between IDEs.

- **Code Completion**. This is a fantastic aspect of IDEs that is hard to do without, and is an excellent learning assistant. When you type the name of a package, a small menu pops up for you to choose available classes in that package. Or when you type a variable name, it knows to what class it is assigned, and pops up all of its fields and methods. Also, many IDEs will make suggestions about what you're trying to do. They'll tell you stuff like, "you declared this array but then never referenced it," or do you want to import this or that class. My favorite warning is from IntelliJ: "silly assignment." That's exactly what I thought throughout graduate school.

- **Integration with external tools**. You get an easy, consistent interface for configuring many external tools. While you're learning Java, this isn't too big of a deal. But you do get a way to run programs, and applets, and execute an Ant build.

- **Language Support**. After you've learned the basics of the Java language, you'll likely begin to expand the complexity and reach of your programs. Reach often means heterogeniety. IDEs often support multiple languages, including not only Java but also others you'll need, such as HTML, XML, JavaScript, JSP, and so forth. That comes in handy, and if your IDE is sort of configurable, you can go off the deep end.

- **Project Handling**. Notice that I didn't call this "Project Management" because that sounds misleading: an IDE does not do schedules or budgets or any of that managerial jazz. But it does make a clear way to organize your code, make packages, and deploy a complete application. This also means you can view not only the files in your project, but the structure of your code (its constructors, fields, and methods).

- **Refactoring**. Refactoring is something we'll talk about in more depth later. It means taking existing code and making it better without adding functionality. Refactoring is extraordinarily important to the long life and health of your projects. The IDE version of this often means making it easy to rename or move a package or class. Sometimes it means more. Again, this "something more" is something I'd like to do myself.

- **Code Generation**. There are a lot of repetitive tasks in writing code. The most common examples in object-oriented programming are probably getter and setter methods. Often, developers write classes that contain private variables and public methods to retrieve and update their values; those methods are called accessors and mutators, or more commonly (and perhaps less idiotically), getters and setters. It is wicked tedious to write all those, especially if you decide that your userID String should be an int. Or whatever. Now you have to go change two methods too. IDEs keep this kind of housekeeping in order with little hassle on your part. Be wary of generating huge blocks of meaningful code, however. If you want an IDE to do all your work, save your time and go do VB. I don't mean that. I take it back. I don't want you to go do VB. I think Java is better. I don't really like all that code generation for stuff like updating your database at all. In fact, I hate it. Don't you?

So I'm not going to tell you what to do. That would be beside the point, because probably you're going to use whatever they have at your school or your work. But if you're shopping around, this gives you some ideas. All I want to stress is: write your code by hand for a while—possibly a long while—so that you have real, portable skills.

Chapter 6. PRIMITIVE TYPES

DO OR DIE:

- Dig the primitives.
- Yeah, baby, I'm lovin' the primitives.

This short topic introduces the eight primitive types in Java.

About Java Primitives

Java primitives are not objects. They are simple values. They are used to represent single characters, numbers, and the logical values true and false.

Primitive types in Java have a non-trivial benefit over primitives in languages such as C and C++: the ranges of Java primitive types are not dependent on the underlying system. They do not change from system to system as you port your app, because of the Java Virtual Machine. This keeps you from the dangers of overflow problems you can run into in those languages. When you specify an int in C or C++, you don't know what the actual range is; your program will have to use whatever the range of the primitive is on the target platform. For example, in Java, you don't have to worry that on a 16-bit processor, your int will allow only a 16-bit range, but on a more recent processor, 64 bits. This could cause serious problems for your data, which would not be truncated, but would instead *wrap*.

Java stores its data as different types for different kinds of things. That's why it's called a strongly typed language.

For the numeric number, the lower range is calculated as 2 to the power of the number of bits minus 1. The upper range of the data types is calculated as 2 to the power of the number of bits it can hold minus 1, minus 1. No, that's not a typo. Consider this. How many bits are in a byte? Eight. The Java data type byte is capable of holding 8 bits of data. So now you know the range of possible values. Let's show our work like Mr. Foss made me do endlessly back in the 10th grade. He tortured me, and now you must be tortured too. To determine the range of a byte we do this math: 2 to the power of 8 bits - 1, -1. Which is: 2^8-1 or 2^7-1. And 2^7 is 128, minus 1 is 127. We subtract one in order to accommodate 0, which is counted as positive. So the highest value we can represent with a byte is 127. The low range is one step easier to get: 2^8-1 is 2^7 is 128. So a byte in Java is capable of representing the values -128 to 127.

Integer Types

The Java integer types are byte, short, int, and long.

byte

Occupies 8 bits or 1 byte, which is:

-2^7 to 2^7-1 or -128 to 127

Default value of 0

Example: -17, 123

short

Occupies 16 bits or 2 bytes, which is:

-2^{15} to $2^{15}-1$ or -32,768 to 32,767

Default value of 0

Example: 31,098, -9001

int

Occupies 32 bits or 4 bytes, which is:

-2^{31} to $2^{31}-1$ or -2,147,483,648 to 2,147,483,647

Default value of 0

Example: 50, 2147000000, -53000

The int is probably the most commonly used integral type in Java. That's because it is often not worth trying to save memory space by using a smaller type, such as a byte. Here's why: when you perform a comparison operation or an arithmetic operation with a byte, it gets promoted to an int anyway by the runtime, then the operation is performed, and then you have an int. So it's extra work, and a little confusing. But there are many situations where you need specifically one of those types.

long

Occupies 64 bits or 8 bytes, which is:

-2^{63} to $2^{63}-1$ or -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

Default value of 0

Example: 9,223,372,036,854,775,807, 0L

The long takes up a good deal of memory, and so it is typically reserved for use in situations that exceed the capacity of the int, such as representing the total population of the earth, the national deficit in dollars, or the total number of hours I've spent goofing off on the Internet.

You can distinguish between a long and other integral primitive types by writing a literal L after the number value, like this: 87999065L. If you don't do that, the compiler will assume that your number is an int.

Real Numbers

The Java floating point types are float and double. Floating point types are numbers with a decimal place.

float

Occupies 32 bits or 4 bytes, with 6 or 7 significant digits

Default value of 0.0

Range: 6-7 significant decimal digits

Example: 3.1459F

You specify the float by placing an F literal after the float value. If you don't, the compiler will treat your number as a double.

Java floating point numbers follow the IEE754 specification, which includes positive and negative signed numbers, positive and negative zero, and a special value referred to as NaN (Not a Number).

NaN is used to represent the result of an illegal or invalid operation, such as an attempt to divide by zero. The Float and Double wrapper classes both have a method called isNan(), which returns a true or false depending on whether the specified number is Not a Number.

double

Occupies 64 bits or 8 bytes, with 14 or 15 significant digits

Default value of 0.0

Range: 15 significant decimal digits

Example: 67, 2.0E10, 111,222,333,444.567, 67.99D

The double value in the preceding example includes an E. This is engineering notation for an irrational number representing the base of the natural system of logarithms. It has an approximate numerical value of 2.7182818284. This notation was first used in the mid-seventeeth century and has been calculated to 869,894,101 decimal places. You can use E in your double values like this: the number 8.9E5 is another way of writing 8.9 * 10^5.

You can place a literal D after the double value in order to explicitly indicate in your code that you mean for the value to be a double, as distinct from a float. This isn't necessary, however, as double is the compiler's default.

Characters

A single character is represented in Java with the char primitive. The Unicode char occupies 16 bits or 2 bytes, stores a - z, A - Z, 0 - 9, and numerous other characters including those on your keyboard, and characters in languages such as Hebrew, Korean, Arabic, and other doublebyte characters.

char

The char stores 0 to 216-1 or 0 to 65,535

Default value is '\u0000', which is null in Unicode

You can write Unicode char values using a \u and the code number that represents the desired character. Here are some examples of Unicode values.

```
'\u0030' is 0

'\u0039' is 9

'\u0041' is A

'\u005A' is Z

'\u0061' is a

'\u007A' is z

'\u000A' is LF (line feed will cause a compiler error)

'\u000D' is CR (carriage return - will cause

            compiler error)
```

A sequence of characters is represented by a java.lang.String object, not a char. And Java Strings are objects, not primitives. The char primitive is used to represent only a single character. This is different than in C and C++, in which a String is a primitive type. A char primitive may only hold one single character, and is actually a numeric type. So you use Strings to represent things like a name or a book title. You will learn a lot about Strings in Chapter 13. You can look up the codes used to represent Unicode characters at www.unicode.org.

Because the char is an integral type, it can be used in certain situations with mathematical operators. This ability allows us to "increment" a char. The following code uses the char with mathematical operators to print out the English alphabet from a to z, with each character separated by a comma.

Alphabet.java

```
package net.javagarage.demo.primitives;


/**

* Prints the 26 letters of the English

* alphabet.

*/

public class Alphabet {


public static void main(String[] arg){


    char letter = 'a';

    //initialize two ints

    int x = 1, y=26;


    while (x < y){

        System.out.print(letter + ",");

        //use the ++ operator to increment char by 1

        letter++;

        x++;

     }

}

}
```

The ASCII character set corresponds to the first 127 values in the Unicode character set.

Note that in the preceding example, we use a single statement (remember that a statement is some code that is delimited by a semi-colon) to initialize the values of two separate integers. This is legal, but some managers pooh-pooh the practice, citing that it is harder to read.

Logical Representations

The boolean is the only logical primitive type in Java, meaning it is capable of helping you perform logical operations.

boolean

The only type used to represent true and false values in Java is boolean. The word is written out completely, not shortened to bool as in C++, C#, and other languages.

Possible values are true or false.

Default value is false.

Note that in Java, the boolean values are literals: you may not substitute a 0 for false or a 1 for true as in other languages. If you are in the habit of using 0 and 1 in this way, break it quick. As a side note, if you are in the habit of returning -1 from a method to indicate a non-standard state, break that habit too. Java has a robust exception handling facility for such business.

So, yeah. I guess that's all I want to say about primitives right now. Let's get on to more interesting things. Like grouting some tile. Have you ever been forced to clean the teeth on your dad's rusting rakes all day on a Saturday? Me neither. But man that sounds boring.

Chapter 7. OPERATORS

DO OR DIE:

- Tumble over ice
- Add twist
- Serve

Operators are used in Java more or less as you would expect. Let's face it, operators are stoopid boring, mostly obvious, and 90% intuitive. But the section 23A of the Law of Tech Book Writing states that all programming authors must talk about operators. Okay. I'll make you a deal. We'll play nice, and deal with operators, and just get it over with as painlessly as possible.

Java features operators for arithmetic, incrementing and decrementing, logical operations, and bitwise operations. First, let's take care of the unpleasant business of operator precedence.

Operator Precedence

The compiler will follow certain rules when determining how to execute a mathematical statement containing multiple operators.

Lucky for us, they are the standard rules regarding operator precedence:

1. Operators inside parentheses are evaluated first.
2. Multiplication and division (left to right).
3. Addition and subtraction (left to right).

The following code illustrates these concepts.

OpPrecedence.java

```
public class OpPrecedence {

  public static void main(String[] args) {

    short x = 10 + 9 / 3 * 2 - 5 + (4 - 2);

    System.out.println(x);

  }

}
```

That code is a complete class that you can type in, compile, and run. I know we haven't talked about classes yet, but I'm going to try to do that a lot so you can get comfortable typing the code in and running it quickly anyway.

If you carefully read the preceding rules on operator precedence, you probably have fallen asleep by now. In the event that you managed to remain awake, you will know that the result is 13. Here's why:

4 - 2 is 2. Hang on to 2.

9 / 3 is 3

3 * 2 is 6

10 + 6 is 16

16 - 5 is 11

11 + 2 is 13

Straightforward enough.

Arithmetic Operators

These are +, *, -, and / for addition, multiplication, subtraction, and division. That's about it.

There is also the modulus operator, which looks like this: %. If you can't remember modulus, which actually comes in handy with surprising frequency, it is used to return the remainder of performing a division operation.

For example, 10 % 3 = 1, because 10 divided by 3 is 3 (which we don't care about) with a remainder of 1 (which is what we care about when we use modulus).

There's one more thing: you can use a special shorthand in combination with any of the arithmetic operators (excluding modulus) to modify an existing value.

Say you have a variable of type int called x and its value is 0, and you want to add 1 to it. You can do this the boring old nerdy way, like this:

```
x = x + 1;

//x is now whatever it was before (0 in this case),

//plus 1
```

Or you can do what all the cool kids do, and write it more concisely, like this:

```
x += 1;

//x is now whatever it was before (0 in this case),

//plus 1
```

With this incredible time-and space-saving feature, you can shave hundreds of minutes off of your programming time. (Maybe, I guess, if you programmed for billions and billions of years. And were a pretty slow typist.)

Although the way I have written it here is the most common way you see this feature used (with the addition operator adding 1 to the value), you could go crazy with it, using other operators:

```
int x = 18;

x /= 6;

//x is 3
```

The + operator is called an overloaded operator. That means it does something different in different contexts. We have seen how it works with integer values. It also can be used to concatenate (stick together) Strings. Like so:

```
String s = "pound";

String t = "com" + s; //compound

String x = "before ";

x += t; //value of x is now "before compound"
```

Incrementing and Decrementing Operators

This is the kind of thing you need to do with some frequency, especially within loops. Java makes incrementing and decrementing a variable by 1 easy with these special operators:

```
x = 0;

x++;

//x is 1
```

You can subtract the same way:

```
y = 10;

y--;
```

The preceding operators are called post-increment and post-decrement, because they add or subtract 1 *after* the variable has been evaluated.

FRIDGE

You may only use these operators with variables—not with literals. That is, you can't do this: 10++;. That's illegal! You can't do this either: String x -= "something";. No funny stuff, now.

You can also use these operators for pre-incrementing and pre-decrementing, which adds or subtracts 1 before the variable has been evaluated. Like this:

```
--x;

++y;
```

Here is a class you can compile and run to demonstrate these operators:

OperatorTest.java

```
public class OperatorTest {

public static void main(String[] args) {
  int x = 10;
    //evaluate, then increment
  System.out.println("post decrement: " + x++);
    //prints 10
  System.out.println(x); //x is now 11
    //decrement, then evaluate
  System.out.println("pre decrement : " + —x);
    //prints 10
}
}
```

Relational Operators

There are relational operators used for testing whether a value is greater than or equal to another value. They are shown in Table 7-1.

7-1. The Logical Operators

Operation	Operator	Example
Equal to	==	(x == 1)
Not equal to	!=	(a!= b)
Less than	<	(i < 10)
Greater than	>	(i > j)
Less than or equal to	<=	(j <= 10)
Greater than or equal to	>=	(x >= y)

Conditional Operators

There are operators used to express boolean relationships between expressions (see Table 7-2).

7-2. Boolean Operators

OPERATION	OPERATOR	EXAMPLE

AND	&&	int x = 1;
		int y = 10;
		((x == 1) &&
		(y==10))

OR	||	((y >2) || (true))
NOT	!	(! x)

The boolean operators are used in expressions to reduce to a boolean value of true or false.

The operators short circuit, which means that if it is not necessary for the second expression to be evaluated, it is not evaluated. In the following example, the second expression is never looked at by the runtime:

```
if ( (false) && (true) ) {}
```

Because the first expression is false, we know that both the first expression AND the second expression evaluate to true. So we don't go there.

By the same token, the second expression will not be evaluated in this example.

```
if ( true || false)
```

In a boolean OR operation, only one of the terms needs to be true for the statement to evaluate to true, so because the first one is true, we already know the whole thing must be true.

Ternary Operator

This operator is a shorthand version of the if/else construct that some developers find confusing, and others like. It takes the following form:

```
boolean expression ? action if true : action if false
```

The first part is an expression that evaluates to either true or false. The code after the question mark executes if the expression is true; the code after the colon executes only if the expression is false.

It serves as a replacement shorthand for the following construct:

```
if (boolean expression) {
   action if true;
} else {
   action if false;
}
```

So it looks like this in code:

```
public int returnLesserOfTwo(int x, int y) {
   return (x < y ? x : y);
}
```

If you pass that method an x value of -20 and a y value of -80, it returns -80. I like it. Just don't nest them if you want to eschew obfuscation. Here's what I mean. Three nested ternary operators just isn't pretty.

```
private static boolean badTernaryNest() {

   return

(6%3>1)?(5>10?(2<=3):(3*4<10?4>3:7==7)):false;

}
```

Although you can do that (it returns false if you're interested), please don't. It is almost guaranteed to incur the wrath of anyone who has to read your code.

Binary Numbers and Logical Operators

For all of my griping in this chapter, the bitwise operators are actually very cool. The bitwise operators let you manipulate the individual bits that make up an integer value. You do so using the operators & (AND), | (OR), and ^ (XOR, the EXCLUSIVE OR). Using these operators with an integer value lets you determine the value (0 or 1) for the bit in the specified position.

If you haven't used these operators before, it might sound ridiculous to think of "2 OR 6 = 2" as being a meaningful expression. But it is. For that expression to make sense, we need to examine the bits of a number, which means you need to be able to visualize a base 10 number as a base 2 number. Base 10 numbers are what we use every day. It includes the numbers that we can make using the digits 0-9. Base 2 numbers use only the digits 0 and 1 for representing values.

The bitwise operators take a number in base 10 and look at it as its base 2 counterpart and do something with the bit value at the specified position. Still sounds opaque. Let's circle around again.

How do you represent a base 10 number as a number in base 2? You only have a 0 and a 1 available to you. What you do is generate a number line that starts with 1 and goes out to the left. The numbers represented on the number line are the powers of 2. Here is such a number line that goes out only a few powers of 2:

```
64 32 16 8 4 2 1
```

We get this number line because 2_0 is 1, 2_1 is 2, 2_2 is 4, 2_3 is 8, and so on through 2_6. Now we can use this number line to take values from. We can represent any base 10 number between 0 and 127 using these numbers. We can do that by adding different numbers together.

For example, 3 is a base 10 number that isn't on the number line. We can represent 3 by using only a 2 and a 1, and none of the other numbers on the number line. We can get 22 by using the 16 and the 4 and the 2, and none of the other numbers.

If we want to represent a base 10 number in binary, we take the numbers we need and leave the ones we don't. To take a number, we put a 1 under it; we put a 0 under numbers that we don't need to use.

So let's do that now. Let's represent 3_{10} as 3_2.

64	32	16	8	4	2	1	
0	0	0	0	0	1	1	(1 + 2 = 3)

If we take the 0s and 1s above as our number, we get 11. 3_{10} = 11_2. So our answer is 11.

Let's do 22 now.

64	32	16	8	4	2	1	
0	0	1	0	1	1	0	= 22_{10} (because 0 + 2 + 4 + 0 + 16 = 22)

So we represent 22 base 10 in base 2 as 10110. The 1 means that bit is on (or true) and 0 means that bit is off (false).

Now we can use this as the basis to perform our logical operations. Table 7-3 shows the outcome of boolean expressions.

7-3. Boolean Operator Expression Results

AND	true AND true = true	true AND false = false	false AND false = false

OR	true OR true = true	true OR false = true	false OR false = false
XOR	true XOR true = false	true XOR false = true	false XOR false = false

In that table, if we substitute 1 for every "true" and 0 for every "false," perhaps you can start to see how we could say something like 3 AND 22.

Let's take our base 2 representation of 3 and our base 2 representation of 22 and line them up on top of each other, like this:

64	32	16	8	4	2	1	
0	0	0	0	0	1	1	= 3 (1 + 2)
						AND	
0	0	1	0	1	1	0	= 22 (0 + 2 + 4 + 0 + 16)

Now we just need to draw a vertical line between each bit that makes the 3 and each bit that makes the 22, performing the logical operator AND on each one. Here we go:

> 1: 1 AND 0 = 0
>
> 2: 1 AND 1 = 1
>
> 4: 0 AND 1 = 0
>
> Skip 8 because it isn't used.
>
> 16: 0 AND 1 = 0

So now we've got a 0 (not used) in every place but the 2. The 2 bit has a 1 (true). So, the answer to the expression 3 & 22 is 2.

Let's try another quick one, because it's so fun. What's 13 ^ 6 (13 eXclusiveOR 6)?

64	32	16	8	4	2	1	
0	0	0	1	1	0	1	= 13 (because 1 + 0 + 4 + 8 = 13)
						XOR	
0	0	0	0	1	1	0	= 6 (because 0 + 2 + 4 = 6)

We can do this quickly now. Remember that exclusive or means that each operand must be a different value.

> 1: 1 XOR 0 = 1
>
> 2: 0 XOR 1 = 1
>
> 4: 1 XOR 1 = 0
>
> 8: 1 XOR 0 = 1

The answer is 1 + 2 + 0 + 8 = 11. 13 ^ 6 = 11.

Here is some Java code that prints out these different values just to prove it.

BooleanOps.java

```
public class BooleanOps {

  public static void main(String[] args) {
    System.out.println(3 & 22); //2
    System.out.println(13 ^ 6); //11
    System.out.println(0 & (3-3)); //0
    System.out.println(45 | 16); //61
    System.out.println((2 & 5) | ((7^3) & 5 )); //4
  }
}
```

Now that we can see behind the base 10 and into the bits, we can use the bit shift operators to manipulate the individual bits of a number.

Shift Operators

There are three shift operators, used for changing the bit values in integral types. They aren't used frequently, except in programs that need to keep their memory usage to an absolute minimum. The shift operators are

1. The left shift, represented as <<
2. The signed right shift, represented as >>
3. The unsigned right shift, represented as >>>

The left-hand side of the operators is the value to be shifted, and the right-hand side is the number of places to shift it. The type of each operand has to be an integral type, or a compiler error will occur. The shift happens on the 2's complement integer representation of the value on the left.

When the value to be shifted is an int, only the last 5 digits of the right-hand operand are used in performing the shift. When the value to be shifted is a long, only the last 6 digits of the right-hand operand are used in performing the shift.

The shifts occur at runtime, on a bit-by-bit basis.

Let's show the bits for the number 13, which we learned how to do in the last section.

64	32	16	8	4	2	1	
0	0	0	1	1	0	1	= 13

If you shifted all of the bits that make up that number 1 to the left, what would you have? You'd have this:

64	32	16	8	4	2	1	
0	0	1	1	0	1	0	= 26 (because 2 + 8 + 16=26)

We have the left shift operator to do this work for us, and we can represent it in Java code like this: (13 << 1).

What if we shifted 13 over 1, but to the right?

64	32	16	8	4	2	1	
0	0	0	0	1	1	0	= 6

Wait a moment. What happened to the 1 value that we originally had sitting under the 1 position? It just dropped off the map. It is forgotten about completely, and we have only what's left. If we shift 13 over to the right 10 positions, or even 938 positions guess what we end up with? 0. That's because the regular right shift operator (>>) is signed. Use the unsigned right shift operator to move 13 over 1 place, and you get the same result (6).

So what if we use a negative number to start with? For example, (-13 >>> 2)? Should be a fairly similar, innocuous result, and then we can call it quits. I'm afraid that (-13 >>> 2) = 1073741820.

What in the world will become of us?

Well.

The value of num<<pos is num shifted to the left pos positions.

The value of num>>pos is num shifted to the right pos positions, with the sign extension retained. The result is num/2pos. For non-negative numbers, performing a right shift is equivalent to dividing the left-hand operand by 2 to the power of the right-hand operand.

The value of num>>>pos is num shifted right pos positions with 0 extension. For all positive nums, the result is the same as using the >> operator. The difference between >>> and >> is that >> fills in all of the left bits with zeros. That means that *the sign value can change.* For negative nums, the result is equivalent to num right-shifted by pos plus two left-shifted by the inverted value of the right-hand operand.

There are limited contexts in which you need to use the shift operators. Many of the Java APIs shield us from having to deal with low-level bit manipulation tasks, so we can probably quit talking about them now. Anyway, more talk at this point would just be more of the same.

Chapter 8. CONTROL STATEMENTS

DO OR DIE:

- To be the smooth jazz of Java chapters

Control statements are the things you write to control the flow of your program. They're pretty straightforward for the most part, and very similar to how these constructs are implemented in other languages.

So I am going to go out on a limb here. I am betting that you're going to be familiar with when and why to use stuff like if/else statements and for loops. Maybe we can spare ourselves the formality. If you're not familiar, that's okay. These are basic elements in any programming language, and there are hundreds of examples of their usage throughout the remainder of this book.

If/Else

The if/else statement is used to execute code when a test condition is true.

```
if (x > 10) {
  System.out.println("x is greater than 10");
} else {
  System.out.println("x is less than 10");
}
```

Switch/Case and Breaks

The following code shows how to use a switch/case construct with a char. But any of the following primitive types are legal to test against: char, byte, short, or int.

```
switch (test) {
case 'A' :
    System.out.print("Found X");
    break; //Print 'X' if test = 'X'
  case 'B' :
    System.out.print("Found Y");
    break; //Print 'Y' if test = 'Y'
  default :
    //This code runs if test does not equal 'X' or 'Y'
    System.out.print("Found Z");
}
```

If neither break were there, and if test equaled 'X', the code would print XYZ. Java switch/case constructs "fall through" until they encounter a break statement or the end of the switch.

The break keyword is used to stop processing within a case statement. Note that the keyword default is used to specify the statement to execute in the event that the passed number does not equal any of the explicit case values.

While Loop

The while loop performs its test against the truth value of the expression *before* the loop executes. That means that if the test expression is initially false, the loop will never run.

```
int x = 0;

while (x < 10) {

   System.out.println(" x is now : " + x);

   //increment x by 1

   x++;

}

System.out.println("x is not less than 10 anymore: " + x);
```

Note that we are only able to refer to the variable x outside the loop because it was declared outside the loop. If you declare a variable inside a loop, you won't be able to reference it outside the loop.

Do-While Loop

The do-while loop differs from the while loop in two ways. Most importantly, the code is guaranteed to run at least once, even if the test expression evaluates to false on the first test. That is the point of this loop. The second way it is different is that it is hardly ever used. I mean, hardly ever used at all. But it's easy.

```
int x = 0;

do {

   System.out.println(" x < 10 : " + x);

   x++;

} while (x < 10); // test not performed until after

 // code block is executed at least once

System.out.println("x is not less than 10 anymore: " + x);
```

For Loop

The for loop is perhaps the most popular of all loops. It comes in two versions: one for iterating against a test expression, not unlike the while loop, and a second for iterating once for each item in a collection (such as an array or Hashtable).

The first kind of for loop has the following basic structure:

```
for(initialize a variable; test the expression;

operation to perform after iteration of loop)

{

//statements to execute when loop is true

}
```

Here is a simple example:

```
for (int x = 0; x < 10; x++){

   System.out.println(" x < 10 : " + x);

}
```

Note that at this point in the code, we cannot refer to the current value of x (10) out here. That's because we didn't declare it outside of the for loop, but rather declared the variable as part of the loop construct itself.

You may optionally omit the curly braces used around the statement that executes in the event that the test condition returns true. However, you must keep in mind that only the *first* statement following the for loop will be executed; if you want to do more than one thing inside the for loop, you have to use the curly braces. Otherwise, that second line of code doesn't get executed until the loop exits entirely (the test condition returns false).

Complex For Loop

The for loop can also facilitate more complex statements. Taking the same basic structure illustrated in the preceding sections, you can add initializing statements and more operations to be performed after the loop iterates. The additional initializers and operations are separated by commas. Here is what I'm talking about:

```
//two initializers and two post-loop operations

for(x=0, y=0; x <10; x++, y++);
```

Here is an example:

```
private static void complexFor(){
   int iStop = 8;
   int jStop = 21;
   for (int i = 0, j = 0; ((i < iStop) &&
       (j <= jStop)) ;
                              i++, j=i * 3)
     {// \t is the tab character
     System.out.println("i: " + i + "\tj: " + j);
   }
}
```

Note that in Java a simple semicolon (;) usually used to mark the end of a statement is also considered a complete statement itself. That makes the following entirely legal:

```
int i = 0, j = 5; for ( ; i < j; i++, j++ ) { ; ; }
```

From the cool-and-weird-code-to-impress-your-friends department: ask them to write the simplest possible *legal* for loop that executes a statement. Here's the answer:

```
for ( ; ; ) ;
```

Then for a practical joke, have them run the code. Can you guess what happens? It's an infinite loop that runs until it crashes their virtual machine! Hilarious. No one ever tires of a really *good* joke….

FRIDGE

Note for Nerds

The for loop actually compiles into a while loop in Javaland. Because they do the same thing, the compiler finds it more efficient to just manage one of them. So in a sense, a for loop is syntactic sugar. Which is weird for such a mainstay kind of a statement. But this is not true of the foreach loop, which behaves differently.

Nested For Loop

You can nest these different constructs within each other. You can put a while loop inside a for loop, and so forth. You'll find that putting one for loop inside another is commonly necessary. To demonstrate, let's print a multiplication table through 10, making a new line after every 10 results.

```
private static void timesTable(){
  for (int i = 1; i <= 10; i++){
    for (int j = 1; j <= 10; j++){
      System.out.print(i * j + "\t");
    }
    System.out.println();//makes a new line
  }
}
```

Here, we use the \t character to separate each result with a tab.

Continue and Labels

The continue keyword is used to skip the current iteration of a loop if some test condition returns true. There are two ways to use it. The first is without a label.

```
//the continue statement must be used inside a loop,
//or your code won't compile.
private static void demoContinue(){
  for (int i=1; i <= 100; i++){
    if ((i % 4) != 0)
      continue;
    System.out.println(i);
  }
}
```

The preceding code will print out every multiple of 4 between 4 and 100, inclusive. It does so by testing: "If I divide the current number by 4 and do not get a remainder of zero, don't print out this number." The continue keyword causes program execution to jump into the next iteration of the loop.

The second way to use continue is with a label, which allows you to explicitly state the place in your code you want to jump to. These are sort of useful for inner loops. In fact, a label must be specified immediately prior to a loop definition. You can just put a label anywhere and jump to it with continue. Here's an example:

```
private static void demoContinueLabel(){
      //here outer: is a label indicating
      //a named reference point in the code
      outer:
   for (int i=0; i < 4; i++){
      for (int j = 0; j < 4; j++){
         if (i < 2)
            //go to the label
            continue outer;
         System.out.println(" i is " + i + " and
               j is " + j);
      }
   }
}
```

FRIDGE

There is no goto keyword in Java. It was determined a long time ago that goto is evil and does bad things to good programs. So, although it has no meaning in the Java language, you are not free to use it as an identifier (variable name), because goto is a *reserved* word.

This code produces the following result:

```
i is 2 and j is 0
i is 2 and j is 1
i is 2 and j is 2
i is 2 and j is 3
i is 3 and j is 0
i is 3 and j is 1
i is 3 and j is 2
i is 3 and j is 3
```

That's because the continue keyword is used to jump out to execute the next statement after the label we named outer in the event that i is less than 2. So, the print statement is only reached when i is equal to or greater than 2.

Labels are almost *never* used in Java. Neither is the continue keyword for that matter.

They're considered kind of bad practice. Although it works great in the preceding example, if you catch yourself starting to type continue inside a loop, stop, put your hands up, and step away from the keyboard. Consider if it is really necessary. You'll find it is perhaps never necessary, and you can simplify your code in more obvious ways that improve its readability and predictability.

FRIDGE

We won't talk about Collections until the sequel to this book, *More Java Garage*, but for now know that they are Sets, Hashtables, Vectors, ArrayLists, and other things that you might be familiar with. If none of that is ringing a bell, Java Collections (in a gross generalization) often act like resizable arrays. They're different classes that store lists of objects that you can iterate over.

Enhanced For Loop (for each loop)

The enhanced for loop, which acts like the *for each* loop that you might be familiar with from other languages such as VB.NET and C#, gives you way to cleanly iterate over a collection with less fuss and muss.

Here is the syntax of the *for each* loop:

for (ElementType element : expression) statement

That syntax translates to this in practice:

```
for (Object o : myCollection)

    { //do something with o }
```

Note that the expression must be an instance of a new interface, called java.lang.Iterable, or an array. The java.util.Collection interface has been retrofitted to extend Iterable. The exclusive job of this new interface is to offer the opportunity for implementing classes to be the target of the new *for each* loop.

FRIDGE

This example uses a complete class to demonstrate the for-each loop. Don't worry if this looks unfamiliar to you. There are more examples without the surrounding class code. We'll get to classes in a moment.

The authors of the Java language preferred the colon to adding a couple of keywords (foreach, and in), because adding keywords can leave a language vulnerable to a lot of retro-fitting difficulties. I sort of hated the colon syntax at first, but now I like it because it is very easy to spot in code. This kind of loop is often represented like this in other languages:

```
foreach (ElementType element in collection) statement //C#
```

The preceding syntax uses the foreach keyword, which I find very readable and intuitive. And it's not like the Java authors haven't added keywords to the language before (assert was added in 1.4). Maybe there is a ton of code out there using foreach and in as identifiers (!). Well, either way, I'm happy it's here.

Let's see it in action.

ForLoop.java

```
package net.javagarage.efor;


import java.util.*;


/*

demos how to use the enhanced for loop that is new

with Java 5.0. it acts like a foreach loop in other

languages such as VB.NET and C#.

*/


public class ForLoop {


public static void main(String[] ar){


    ForLoop test = new ForLoop();


    for (Kitty k : test.makeCollection())

    System.out.println(k);


    }


    public ArrayList<Kitty> makeCollection(){

      ArrayList<Kitty> kitties = new

ArrayList<Kitty>();

      kitties.add(new Kitty("Doodle"));

      kitties.add(new Kitty("Noodle"));

      kitties.add(new Kitty("Apache Tomcat"));


      return kitties;

    }
}

class Kitty {

   private String name;

   public Kitty(String name) {
      this.name = name;
   }

   public String toString(){
      return "This kitty is: " + name;
   }

}
```

The output is as expected:

```
This kitty is: Doodle
This kitty is: Noodle
This kitty is: Apache Tomcat
```

You can also use the enhanced for loop with arrays:

```
package net.javagarage.efor;

public class ArrayForLoop {
   public static void main(String[] ar){
   /*this says, 'for each string in the array
   returned by the call to the makeArray() method,
   execute the code' in this case println)*/
```

```
  for (String s : makeArray())

    System.out.println("Item " + s);


  public static String[] makeArray(){

    String[] a = new String[] {"a","b","c"};

    return a;

  }

}
```

This looping construct affects only the compiler, not the runtime, so you shouldn't notice any difference in speed. A for loop is compiled into a while loop anyway, so there is a lot of sharing going on under that particular hood.

The array form of the for : statement generates a compiler error in the event that the array element cannot be converted to the desired type via assignment conversion. Here is an example of using the foreach loop to loop over an array:

```
int sum(int[] a) {


    int result = 0;


    for (int i : a)

      result += i;


    return result;

}
```

This code says, "declare an int called 'result' with a value of 0 to start with." Then, for every int in the passed in array (called 'a'), call the current iteration value 'i' and add it to the result. Very simple, clean, elegant.

Notice that the new loop does not use the term *for each* like other programming languages (such as C#) do. Instead, it uses a :. The new syntax is cleaner and simpler. When you combine the simplicity of the for each loop with the new power of generics (introduced in Java 5.0, and discussed in the upcoming book *More Java Garage*), you can write very clear and clean code.

```
void processNewOrders(Collection<Order> c) {

for (Order order : c)

order.process():

}
```

This code is immediately understandable: you see what type you are using in the signature and in the for loop. It is shorter, and every line lets you know what kind of type you are dealing with. As you might guess, it works with nested loops too.

Okay. Nuff said.

Chapter 9. CLASSES

DO OR DIE:

- Keep your cool
- Act natural
- Maintain

So far we have looked at setting up your environment, and using primitives, operators, and statements. We've gotten the necessary groundwork out of the way, and now we're free to explore classes. It's been, well, I admit it, a little bit lame. I'd like to make it up to you.

From here on out, things will be a lot more interesting, exciting, and new. I promise. It will be like when Abbott and Costello met the Werewolf, or when the Harlem Globetrotters came into ScoobyDoo—just an unbeatable entertainment combo. So please turn your attention to the magnificent JumboTron. A world awaits. The Java class….

The class is the fundamental unit of expression in Java. In this topic, we will learn how to make classes, what they are for, how to modify them to suit your evil purposes, and how to design them.

What Is a Class?

A class typically represents a type of thing in the world of the application. You define a class, and in it write code that represents two things: the properties of that type of thing and how it behaves. You might think of classes as categories. For instance, Person is a category or type of thing in the world. To make a class, you simply open your editor, type the keywords public class, and give it a name. Save that file with a .java extension and compile it using the javac command that comes with the SDK.

We can make a class called Person like this:

```
public class Person { }
```

If you type the preceding code into a text file called Person.java, it will compile. However, it is obviously useless, because it doesn't hold any data, and it doesn't know how to do anything.

We'll get to how to make it do stuff in a moment.

Classes are blueprints for objects. The chief job of the Java programmer is to write classes that represent different aspects of your application. You write classes. That's all you do (pretty much—you also have to implement the ways that classes interact with each other (which you usually do in other classes)). After you have written a class, you have defined *the sort of thing that is possible in your system*.

After you write a class in a source file (a .java file), compiling the source will result in a new file being created on your file system: a .class file.

After you run your program, it will almost certainly create objects. An object is *a particular instance* of a class. A class is a definition of a thing, and an object is a specific example of that thing. Objects are what you use in your running program to get stuff done.

For example, a person is a class (or kind) of thing. You are a particular instance of that category. You are an object of type Person. Randy Johnson is a different object of type Person. Luis Gonzales is yet another particular instance of the general class of thing called Person. Blah blah blah. You get the idea.

To make a new object of a class, use the new keyword, like this:

```
Person eben = new Person();

Person pawel = new Person();
```

We now have two different objects of type Person, each capable of holding their own separate data. They will only be capable of doing the same kinds of things though (because the class defines what they are capable of).

The main things that classes do are hold data (member variables) and define functionality (methods or functions).

An object in Java should generally have some properties. Perhaps our Person has eye color, weight, height, gender, and so forth. Each person—that is, every object or instance of the Person class—will have these properties.

Every person can also do things. A person generally can walk, talk, smooch, drink, and so forth. Each of these actions could be defined in a method of our Person class.

Let's look at how to define these different components. First, we are going to talk about comments. Although they are not the most important things about a class, you will see them inside the class example code, and I want you to know why they are there.

Classes Have Comments

Comments are used to remind yourself or inform others of what is happening in your code. They are ignored by the compiler and should be used extensively.

There are two notations for writing regular comments in Java code. The first is a regular single-line comment that looks like this:

```
public class Person {
   //this is a single-line comment
}
```

The second is used for comments that span multiple lines.

```
public class Person {
   /*
    This is a multi-line comment, and
    anything inside this is hidden from the compiler.
   */
}
```

Comments are really important. Please comment your stuff. It is annoying to point out the obvious, but it is certainly a good idea to keep track of changes to the code, indicate what other code uses it or relies on this code, who wrote it and when, and other things that will be useful down the road. When you comment your code, first think, "What is not immediately apparent about why something is happening here? What would a stranger who is a good Java programmer want to know about this in case of a problem or the need to change it (because that is likely when they'll be looking at your code)?" Notice that the author's name and e-mail are often useful in this regard too. If you answer questions like these, everyone will appreciate it.

Somebody at JavaRanch.com wrote that he writes comments for the psychotic programmer who will read his code, and who knows where he lives. I like that rule. It's a good rule.

I have a template that I sometimes use to comment files. I don't always use it in this book because it takes up a lot of room and you aren't interested in maintaining this code. I use it every once in a while just to emphasize the importance of commenting.

```
/* File: Person.java
 * Purpose: Represents a human who is going to party.
 * Uses: Address

 * Used By: Some other class name

 * Author: E Hewitt

 * Date: 12.31.99

 */
```

You can legally place comments at the very beginning of the file, at the end of the file, and scattered throughout the middle of the file.

Notice that there are obvious exceptions to where you can place comments. You can't place them in the middle of an identifier or in such a way that the comment would prevent the code from compiling, like this:

```
public class Person { //this is a righteous class! }
```

Now, you can't do that, because the comment hides the curly brace that closes the class. Of course, if you move the curly brace to the next line, you're good to go. But, duh. You wouldn't try to do that. What was I thinking? I must need some more Smarties.

Java has an extensive facility for commenting your code called JavaDoc, which we discuss later in this book. This is enough for now I reckon.

Classes Have Data (Members)

A class' data means its variables. These are often called members or fields. I'll use these terms more or less interchangeably because that's how you'll hear them used in the world, and it's good to get used to how others will talk about them. The data are the properties of the type of thing your class represents. (Often the term properties is used to mean only the public member variables of your class—but we'll talk about visibility in a moment.)

It is not a requirement that a class has data. You can define a class that only contains methods, and has no variables at all. That might be a good idea, depending on what your application requires. For example, our application might specify that a Person is the sort of thing that has a name. Let's expand our class to include this functionality:

```
public class Person {

    public String name;

}
```

Typically, your class' members are made up of the nouns you can list about your object. You find out what data your class needs to have by asking yourself, "What properties does a Person have?"

So our Person might have a name, an age, and an address. These could all be properties we add to our class.

```
public class Person {

  public String name;

  public int age;

  public Address address;

}
```

FRIDGE

In software analysis, this is called a HAS-A type of relation. That is, a Person HAS-A name. Another type of relation is IS-A, and we'll talk about that in the Inheritance chapter.

Now, every time we make an object of class Person, we know that we can define each of these bits of data (name, age, and address) for every different Person we make.

Notice that classes can hold data that is of other class types, and these types can be built in to the Java API, or they can be other classes that you define yourself.

The name variable is a variable of type String. String is a class representing a character sequence that has already been written for you as part of the Java API. If Sun didn't provide the API, you would have to define the String type yourself. Because Strings are things that everyone needs, it is very convenient that they are predefined for us.

The age variable is an int primitive type. Because Java supports autoboxing and unboxing of primitives, it acts like an int primitive when that's what you need, and it acts like an Integer wrapper class when that is what you need.

The address variable is declared to be of type Address. Address is *not* a class defined in the Java API, so the assumption is that *you* have defined a class called Address that has properties of its own (such as street, zipCode, or whatever). If you don't make that class first, the Person class won't compile.

You generally can get the value of each member of an object using the dot operator. I write "generally" because you might not have direct access to data from other classes, which is a matter of *visibility*, which we'll deal with later. For now, it is enough to know that our data members in the Person class are all declared to have public visibility, which means that any other class can read or write their values.

Let's make an object, set the value of its members, and then get the value of each member and print it to the console. I'm going to leave the Address member out of this one for now, so we can compile it.

Person.java

```
//declare the class

public class Person {

//define our two variables

public String name;

public int age;
```

```
  //main method is required starting point
  //for Java programs
  public static void main(String[] args) {
    //make the object of type Person
    Person p = new Person();

    //we can now refer to this particular object
    //as "p" to do stuff with it

    //set the value of the name member
    p.name = "Eben Hewitt";
    //set the value of the age member.
    p.age = 32;

    //print out our values
    System.out.println("The person named " + p.name +
        " is " + p.age + " years old.");
  } //end of main method
} //end of class
```

The output of running the Person class is like this:

```
The person named Eben Hewitt is 32 years old.
```

Having compiled the class, you will have generated a second file on your disk called Person class. This is the executable bytecode that the Java runtime interprets. You can open that class file and view it in a text editor, though many of the characters will not be readable, and you must not edit it directly.

Let's make it a little more complicated, and define a second class in the same source file (the one that ends in .java). This will be our Address class. Now that we know what an Address is, we can specify that a Person has a member of type Address.

Hey, I know this code is starting to look a little long. But *a lot* of it is comments. I think it's important to read what is happening line by line; it helps put all of the pieces together, which is also why I'm showing the complete listing. So don't worry.

PersonWithAddress.java

```
public class PersonWithAddress {

 //define our two variables

   public String name;

   public int age;

   //added the gnarly new Address member!

   public Address address;


   //main method is required starting point

   //for Java programs

   public static void main(String[] args) {

     //make the object of type PersonWithAddress

     //PersonWithAddress person = new //PersonWithAddress();


   //we can now refer to this particular object

   //as "person" to do stuff with it


   //set the value of the name member

  person.name = "Bill Gates";

   //set the value of the age member.

  person.age = 32;

   /*

   set the address. wait! it's a class itself. so we have to make an instance of

➥  address, and set its members, and then refer to those vars

   */

  person.address = new Address();

  /*

  now the address field refers to an object of type address, and you can fill up its

➥  members also using the . operator

  */


  person.address.city = "Mann Avenue";

  person.address.city = "Muncie";

  person.address.state = "Indiana";

   //print out some values

   System.out.println(

   "The person named " + person.name

   + lives in " + person.address.city + ", " +

   person.address.state + ".");
```

```
    } //end main method

} //end of PersonWithAddress class


//Whoa—starting a different class in the same

//source file—ok, party on Garth...


class Address {

  //define a coupla public members,

  //just like Person!

  public String street;

  public String city;

  public String state;

} //end Address class
```

This outputs the following:

```
The person named Bill Gates lives in Muncie, Indiana.
```

Classes Have Behavior (Methods)

We use the word behavior colloquially to refer to actions that someone performs. If the actions are performed repeatedly, Aristotle would call that character. My wife told me today that she thinks that Aristotle is a "jerk," but that's not important now. To you, anyway. I will admit I was a bit nonplussed. But only a bit—the "unities" of time, place, and action are pretty goofy if you get right down to it. You define the behavior of a *class* in its *methods*.

Each method (sometimes called a function, or in VB, a subroutine) represents a different little bit of functionality. Here is what makes up a method.

A method does one job. Like variables, methods have *names* or *identifiers*. A method name can have the same name as a member variable, because the parenthetical argument list keeps them from being ambiguous in the eyes of the JVM.

FRIDGE

If you are having a hard time compiling and running your programs using the command-line tools (javac and java), you can get an IDE. An IDE can give you insight about what is legal code at a particular point and color-coding different parts of your programs to improve readability. I recommend Eclipse, available for free from www.eclipse.org. You can also try Sun ONE Studio or NetBeans or Forte (whatever they're calling it today), which is also available for free from http://java.sun.com. (My favorite is IntelliJ IDEA, but it's $)

They have a degree of *visibility*, which indicates what other classes (if any) can call them. Methods can have visibility of default (no visibility explicitly declared), public, protected, or private.

Methods also have a *return type* that indicates the one value that represents the result of performing the method. Methods can take zero or more *parameters*, or values that you pass into them. Parameters are declared using the type of variable you want to accept and an identifier for each variable for use inside the method body.

Methods might throw exceptions, but we talk about that in Chapter 21, "Handling Exceptions."

Methods have a *body*, which consists of zero or more statements in between curly braces that do the work of the method.

Here is an example:

```
public class Person {

   private int age;

   public int getAge() {

      return age;

   }

   public void setAge(int ageIn) {

      age = ageIn;

   }

}
```

This class has one member variable and two methods: getAge() and setAge(). The getAge() method declaration says it returns an int, and the body of the method indeed does nothing but return the value of the age member, which is of type int.

The getAge() method's visibility is declared to be public, so any other class is allowed to use the getAge() method. This method takes no arguments at all.

The second method is called setAge(), and it also has a visibility of public. But it has a return type of void, which means that the method does some stuff but does not return any result to the caller.

A constructor is a special type of method that gives you control over how you create your objects. We'll talk about those in a bit.

Return Types

It may seem obvious, but if you say that a method will return a value, it *must* return a value. You can't do this:

```
public int getAge() {

   int i = 7;

} //won't compile!
```

This method will not compile because you say you're going to do something (return a value) and you don't keep your promise. By the same token, the following is also illegal:

```
public int getAge() {

   return; //won't compile!

}
```

Here, we say we're returning an int, but we don't: returning nothing is not the same as returning something (any kind of int).

The following, however, *is* legal:

```
public void ohWellWhateverNeverMind() {

   return ;

}
```

You can do that because you are returning nothing, which is what you say you will return. That method is equivalent to this:

```
public void hahaha() { }
```

Which is also legal. You don't have to have anything in the body. Though that doesn't do much for you except help you compile your class if you aren't sure what code to put there yet. Both are equivalent to the following, which is also legal:

```
public void allYourBase(String areBelongToUs) {

   ; ; ;

   ; ;

}
```

Again, you can do it, but it's probably not too common. If you see that sort of thing a lot at your job, you might reconsider employers.

Methods can return the result of evaluating an expression that contains literals, like this:

```
public int getADumbOldNumber() {
  return 7 * 54;
}
```

Methods can return a literal, like this:

```
public String getMessage() {
  return "I love you, man...";
}
```

Methods can return the result of a call to another method, which is just another form of an expression.

```
public double getRandomNumber() {
  return Math.random();
}
```

If you think it would be a good idea to be very explicit about what you're doing to help your reader, or if your method consists of a number of statements, you can return a result that you hold in a variable, like this:

```
public double getRandomNumber() {
   double d = Math.random();
   return d;
}
```

Methods can return a type that can be converted at runtime into the declared return type. For example, a float is a floating point type that holds 32 bits of data. A double holds 64 bits of data. That means that any float can fit into a double container. That means that the following is perfectly legal, but confusing:

```
public double getIt() {

  float f = 16F;

  return f;

}
```

Although the number you create inside the method body is of type float, it is a double when the method returns. So, here's a quick quiz. Is the following legal?

```
public float getFloat() {

  return 16F;

}

//call it from somewhere:

double myNumber = getFloat();
```

Here, we call a method expecting a double to be returned, and the method returns a float. This code *will* compile. That is because the variable (myNumber) to which we assign the result of the getFloat() method can hold a double-sized number; the method returns a float, which is not a double, but which can fit into a double. The runtime automatically converts it for us. Everything is cool.

Starting Programs: The Main() Method

There is a method that is used as the starting place for all Java programs called the main method. It is so called in C++ and C# as well. When you start a Java program, you call the java command and pass it the name of the class containing the main method. The signature of this method is like this: public static void main(String[] args). Let's break that down.

The method is public so it can be called from anywhere. It is static, which means that the method can be called on the class itself instead of requiring a particular instance of the class on which to call it (which makes sense because at the start of the world you can't possibly have any objects yet). The return type of the method is void, which means that the method doesn't return anything (note that this is different in C++ where the main method returns an int). The method is called main and takes one parameter, which is an array of Strings. The array of Strings represents any arguments that you want to pass into your program when it starts up.

FRIDGE

It is possible to have more than one main method defined in your application. When you start the program using the java command, however, you must specify the class containing the main method you want to use as the entry point for this run of the application. It is not common to see multiple main methods within a single app.

Say you had a program that launched a rover to Mars. You might have numerous classes that make up the program, and have the main method that starts things off defined in a class called Launcher. Your program is designed to accept the names of the rover that you want to launch. You could invoke it like this on the command line: java Launcher Spirit Opportunity. The name of your program containing the main method is Launcher (so you know there is a public class called Launcher). Then, the String array that enters the main method as a parameter is an array of length 2, with "Spirit" as the value of the 0th cell and "Opportunity" as the value of the 1st cell.

Params.java

```
public class Params {

    public static void main(String[] args) {

        if (args.length == 2){
            String a = args[0];
            String b = args[1];

            System.out.println("Your first argument: " + a);
            System.out.println("Your second
            argument: " + b);
        } else {
            System.out.println("You must supply
            two arguments!");
        }

    }
}
```

The output of this class when called like this: java Params Spirit Opportunity, is

```
Your first argument: Spirit
Your second argument: Opportunity
```

The output when called like this: java Params (with no arguments), is

```
You must supply two arguments!
```

The main method does not explicitly throw any exceptions. If an exception occurs inside the main method, it is thrown out to the JVM, which then halts and prints a stack trace. The stack trace is the list of methods called up to the point where the exception was thrown, in descending order. You can put a try/catch block around the main method if you want, but it will have no effect.

Let's add a few regular old non-main methods to our Person class.

PersonWithMethods.java

```
public class PersonWithMethods {

  //define some members

  public String name;

  private boolean isJerk = false;


    //program starts running here

  public static void main(String[] args) {

          //create one person

        PersonWithMethods p1 = new

                   PersonWithMethods();

          //set the member's value with dot operator

        p1.name = "Aristotle";

          //call the method on the current object

        p1.setJerk(true);


          //create a different person object

        PersonWithMethods p2 = new

          //PersonWithMethods();

        p2.name = "Hegel";

          //pass in a boolean parameter to the method

        p2.setJerk(false);
```

```
          //find out what we know about them.

          //note the explicit call to the method //toString()

        System.out.println(p1.toString());

          //note the IMPLICIT call to toString!

          //both this and the previous statement

          //are equivalent

        System.out.println(p2);


  }


    //in this method, we set the value of the

    //isJerk member variable

  public void setJerk(boolean isJerk) {

    this.isJerk = isJerk;

  } //end method


  public String isJerk(){

    if (this.isJerk == true)

        return "a jerky jerk";

    else

        return "totally a non-jerk";

    }


  public String toString(){

    return name + " is " + isJerk();

  }

} //end class
```

The preceding code outputs the following:

```
Aristotle is a jerky jerk

Hegel is totally a non-jerk
```

We make two separate objects here, which we refer to as p1 and p2. Each object keeps its own data. They each have a different value for name and a different value for isJerk. The methods we call perform the same operation, but return different results based on the member data.

The toString() method is what is called an *override* of the method called toString() in the parent class of Person, which is java.lang.Object. See Chapter 12, "Inheritance," for more on that. I employ it here to demonstrate that the println() method can only print strings, so if a parameter passed to that method is not already a String type (such as the p2 object we pass to it here), the println method will implicitly convert it to a String. That is why we see that the same method is in fact called for System.out.print(p2) as for when we explicitly write it: System.out.println(p1.toString());.

Static Methods

A static method is one that is called on the class itself, as opposed to a method that is called on an instance of the class. The main method is a static method, which makes sense, cuz there cannot be any object instances before the program starts up and begins making objects! The way to make a static method is to declare it using the static keyword.

```
public static void something() { ... }
```

You can then call the method without needing a particular instance of the class. Static methods are used frequently in Java programming. They are used when they can always perform the same operation, regardless of the current program state. That's because static methods can't know anything about the current program state—they aren't called on object instances. They are said to be called on the class itself.

Consider this to help make the point: The methods of the java.lang.Math class are all static. It is easy to see why. They always do the exact same thing, regardless of what is happening in the world.

You can call a static method by using the dot operator after the class name, like this:

```
Math.random();
```

The preceding method returns a more-or-less random primitive double with a value greater than or equal to 0.0 and less than 1.0. You don't need an instance of an object to do that work, because the result will always be the same, no matter what the state of some hypothetical Math object might be. It's not even meaningful when you think about it. What would be the state of Math itself where it would make a difference what number the random method returned? There is no particular instance of the Math class that would make a difference to the sin method, which returns the trigonometric sine of an angle. That operation is always going to be done the same way. By the same reasoning, consider the following methods of the java.lang.Math class: abs() returns the absolute value of a number, atan() returns the arc tangent of an angle, asin() returns the arc sine of an angle, pow() returns the value of the first argument raised to the power of the second argument, and so on. There are a couple dozen utility methods in the Math class such as these. Check them out.

FRIDGE

You can read the Java 5.0 API documentation at http://java.sun.com/j2se/1.5.0/docs/api/. This is an invaluable resource that you will want to bookmark.

Let's now look at a few details regarding static methods.

Here's something that could throw you for a loop: you *can* call a static method on an object. However, doing so does not in any regard change its functionality. It is merely a convenience. It is preferable to call static methods on the name of the class to indicate that they are static methods. But that's a convention that will likely die as developers start adopting the new static import facility in Java 5.0 (discussed in Chapter 11, "Classes Reloaded"). Put a different way:

```
calling Person p = new Person(); p.myStaticMethod();
```

Is identical to

```
calling Person.myStaticMethod();
```

Here is a more complete example:

StaticStuff.java

```
public class StaticStuff {
  public static void main(String[] args) {
     System.out.println(myStaticMethod
        (Math.random()));
  }

  private static String myStaticMethod(double num){
     return "Your lucky lottery number is: " +
       (int)(num * 100);
  }
}
```

The output is something like:

```
Your lucky lottery number is: 63
```

The StaticStuff class features three static methods. This is a good example because it shows how to use static methods, but it also shows how you can nest method calls to simplify your program's look and streamline it.

FRIDGE

Stuff you can mark as static: methods, variables, and top-level nested classes.

It is certainly worth noting that static methods cannot access any non-static members or methods. That makes sense when you think about it. How could something defined on the blueprint level (the class level) know something about any particular instance of an object that later gets made with that blueprint? It can't.

The first static method, main(), is always static and starts the program. We don't pass any arguments into the program, so its String[] parameter is empty. We then call the method println() on the static member out of the System class. We call myStaticMethod(), passing in as a parameter the result of the third static method call—the one to Math.random(). We take the random result generated by the call to Math.random(), pass it into myStaticMethod(), which returns a String that contains the user's lucky number between 1 and 100. We cast the number, which comes in as a double to an int so that it chops off the decimal places, which we don't care about.

Final Methods

In Java, it is possible to extend the functionality of a class by subclassing. For example, we can decide that we want a more specific kind of Person, and then write a class that inherits Person variables and methods. Such a class might be called Programmer. A Programmer is a Person, with an address and a name and all, but a Programmer has other characteristics and things he knows how to do that are not common to all people. When you create a subclass, it is possible to override the methods in the superclass. That is, the Programmer class could choose to redefine a method that is implemented in the superclass. We did this earlier by having our implementation of the Person class override the toString method of the Object class.

If you don't want to allow a particular method to ever be redefined in a subclass, you declare it final. Consider the following method that might return the absolute value of an integer.

```
public final abs(int a);
```

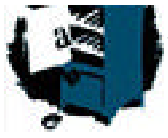
Here, we want to declare this method final, because it would be horrible if we allowed mathematical functions that should determine their results in an absolutely consistent manner to be overridden by a method in a subclass and change how an absolute is determined!

Note that classes, methods, and variables can all be declared final.

If a class is declared final, all of the methods within the class are implicitly final as well. Because the java.lang.Math class is declared final, none of the methods in it need to be declared final.

In the case of classes, it means that no other class can extend it. In the case of a method, as here, it means no subclass can override it. In the case of a variable, it means that its value cannot be reassigned after it has been assigned.

FRIDGE



There are certain things that you must always do when you declare a method. Methods must have a visibility level (default, public, protected, or private), a return type (such as void if the method returns nothing, or int, or Address, or String, and so on), a name, and a body. The body is the part in between the curly braces where you do your work.

You can declare that a method throws an exception in its signature using the throws keyword. If you care about that, it is discussed in Chapter 21.

Rules for Declaring Methods

We will discuss visibility modifiers in the next section. Classes, methods, and variables all have visibility. You can explicitly declare a method as having public, private, or protected visibility. If you do not explicitly indicate the visibility, the compiler will make your method (or class or member) to have default visibility. You don't write "default." For example, the following are all valid:

```
public void getAge() {}

void setAge() {}

private doWork(){}
```

Methods are defined by their signatures. A signature consists of

1. The name of the method
2. The number of parameters the method accepts
3. The type and order of each parameter

You cannot have two methods with the same signature in the same class; the runtime wouldn't know which one to call. You can have two methods with the same name—that is called overloading and is discussed later.

You cannot start a method name with a number or any weird characters like a carat or a % sign. Just don't be trying to do that. You can *start* a method name with an underscore, and the method name can contain numbers, they just can't come first.

Which of the following method declarations are valid or invalid?

```
public void _23() {}

void doinIt(String theThing) {}

int private HEY_SUCKER(){}

protected Address setaddress(theAddress) {}
```

The first is valid, because you can start a method name with an underscore.

The second is valid, because leaving out the visibility modifier is okay, and it will be set to "default" visibility.

The third method will not compile. Although the compiler doesn't care how you capitalize your methods, you cannot declare the return type (int) before the visibility modifier (private).

The last one will not compile because the parameter is not declared to have a type.

You can name the parameter that a method accepts according to the rules for naming any other variable.

In Java 5.0, you can also declare generic methods. These are an advanced topic, however, and if you're interested in using them, please check out the Generics topic in the companion book to this one, *More Java Garage*. Also, check out the Glossary in this book that contains some stuff about using generics.

That might be enough about those rules. Hmm. Yes, that's enough.

Conventions for Declaring Methods

Please follow these conventions for using methods. It will make your day go by with less unpleasantness. Methods are by convention named with an initial lowercase letter, and subsequent words capitalized. Like this:

```
public void someSortOfLongMethodName() {}
```

Method names should be descriptive. Although you are *allowed* to name your method x(), this is a real drag for everybody. Don't make the poor saps that inherit your code when you get rich and move to Tahiti have to deal with stuff like that. Have a heart for those poor souls left back at the office, hunched over their little P4s, in the near-dark, growing only slowly, like mushrooms.

Make your method names that are used just for returning a value get*ThingReturned*() like this: getAddress(). Such a method is called an accessor. If the value returned is a boolean value, you write is*Thing*() like this: isJerk(). Accessors are typically only this:

```
public int getAge() {

   return age;

}
```

Make your method names that are primarily used for setting the value of a member start with "set," like this: set*ThingToSet*().

A typical mutator looks like this:

```
public void setAge(int ageIn) {

  this.age = ageIn;

}
```

At the Java Academy for Super Nerds they brainwash you into saying "accessor" and "mutator" to refer to these kinds of methods. Nobody cool uses those terms though. All of the cool kids say "getter" and "setter." It somehow sounds less idiotic, which is why I think people probably use those instead.

The guidelines for naming parameters to methods is the same as for any regular variable name. Say you have a setter method. Call it setAge(), and it takes a parameter of type int like this: void setAge(int age). Typically, you will be setting the value of a class member. Name the method parameter after the class member, and name the method set*Member*(). So you have a method that ends up looking like this:

```
void setAge(int age) {

   this.age = age;

}
```

The keyword this is used to refer to the current instance. In VB, that's when you write "Me." It means "the current object," and in this context it helps us tell the difference between the member variable and the parameter. Some people slightly change the parameter name. My friend Vic does it by suffixing the word In to the variable.

```
void setAge(int ageIn) {

   age = ageIn;

}
```

Then, you don't have to use the this keyword. You can do that if you want. It might help to keep everything straight at first. We'll talk about this in a mo'.

Classes Have a Home (Package)

A package in Java is a namespace that allows classes to be identified. Packages allow you to have two different classes of the same name within the same application if the package name is different. The purpose of packages is to resolve naming conflicts.

To place your class in a package, use the package keyword. The package keyword acts as a logical namespace to separate different classes that go together. It allows you to put them in groups.

A package also corresponds to a physical location within a directory structure. This is different than in a language like C#, where the namespaces are purely virtual. In C#, you can put your class in any old directory and call the package whatever you want. Not so in Java. If you have a class called Kitty and you want to put it in a package called pets, the Java class file must physically reside in a folder called pets.

It follows that a class may reside in only one package.

If you explicitly put your class in a package, the package statement must be the first line of code in the source file that is not a comment. Comments can go first. But nothing else. Like this:

```
package net.javagarage;

public class MyClass {...}
```

The preceding statement means that there is a folder at the root of the application called net and it contains a folder named javagarage and that folder contains your class file. I can now write a class called MyClass and put it in the com.loveboat package, and the compiler and runtime will be able to tell them apart.

If you have more than one class in a source file, the package statement applies to every class in the source file.

If you do not explicitly place your class in a package, it will reside in the default package.

There is a convention for naming packages that just about everyone follows. It goes like this: Take your company's Web site and use its top-level domain as the top-level package. For example, in my case, my domain is called javagarage.net, so the TLD is net, so the top-level package is net. Then, put the domain name as the second highest-level package name. Don't put any classes at all in either of these packages. But now, after you're two deep, you usually name your application. Unless there is a subdomain that is relevant. Then, you can use the subdomain. For example, if I made an ecommerce site, I might put those classes in net.javagarage.store, and I could start defining the custom packages that I think I'll want in my app. Maybe products could have its own package, and checkout could have its own package. And like that.

For instance, I have classes in the net.javagarage.demo.inherit package that demonstrate using inheritance. I have classes in the net.javagarage.demo.classes package that demonstrate using classes. And so on. This is a good convention, based on the fact that a company's domain name is necessarily unique, so you know that your class will be unique. You just need to sort out your modules within each application after that, for organizational purposes.

It is important to take care with naming your packages, and organizing your classes within them. It is not uncommon for developers to revise and reallocate their package design multiple times throughout writing an application.

Classes Have Visibility

Up until now, we have put the keyword public in front of our class declaration, our method declarations, and our member declarations. We have mentioned different levels of visibility, but not what each of them means.

There are four levels of visibility in Java. They are listed here from least restrictive to most restrictive:

1. **public**. This is the most visible. It means that any class, in any package, can use that item. Classes, methods, and member variables can all be public. Classes declared public must be in a source file (the .java file) of the same name with matching case. Although you create multiple Java classes in the same source file, only one of them can be declared public, and that one must have the same name as the source file.
2. **protected**. Methods and variables that are declared protected are only visible to classes that are in the same package as this class, or are subclasses of this class. You cannot declare a class itself to be protected.
3. **default**. This is the level of access attributed automatically to a class, variable, or method that has no explicit access modifier on it. It means that this class can only be used by classes within the same package. The term default is not itself a keyword.
4. **private**. Methods and variables that are declared private are only visible inside the class in which they are declared. They cannot be seen (used) from any other class.

What level of access does the following method have?

```
int[] getMedicalRecords(Person p) { ... }
```

Default. Only classes in the same package can use this method.

If you try to access something that you aren't allowed to, the compiler will tell you with a message saying, "Can't access classname…". If you want to know more about classes, check out Chapter 11.

Chapter 10. FRIDGE: MMM-MMM LAMB CHOPS AND A MANHATTAN

I thought you might like my recipe for perfect lamb chops. It's a simple recipe that will make you very popular with whomever you're trying to be popular with.

Then we'll make a refreshing beverage to go with it.

When it comes to steaks, I have found that what really matters is the steaks you start with. With lamb chops, they all seem roughly the same. Which is just fine. Get enough to have four chops per person.

About half an hour before it's time to cook the chops, take them out of the fridge and put them into a ceramic dish. It doesn't need to be ceramic I guess, just not metal. Glass is good.

Drip extra virgin olive oil across each side. Generously mill rock salt and whole peppercorns over the chops.

Add a little finely crushed dill, which seems to be the key. Rub. Leave out at room temperature for 30 minutes or so.

Put the broiler on high. Cook for 4 minutes, then turn and cook for 3 minutes. Don't overcook them. This is assuming your chops are about 1 inch thick, which they should be. Serve over sticky basmati rice.

I know that this recipe is pretty simple. But sometimes, you know, that's the best thing.

Now the best thing to drink with lamb chops is a Manhattan. Everyone says so. So here is my recipe. I used to be a barkeep in New York City, which is the city where they force you to go to shows in the fake, sanitized Disneyland Times Square (ack, I'll take Shaft's 43rd and Broadway any day…) if you get their drinks wrong. So I know what I'm talking about.

Fill a rocks glass with ice. My grandmother only takes 5 solid cubes. You can do it that way if you like.

Add a jigger of bourbon. This is like the chops: getting good bourbon to start with is the key.

Add a solid dash of vermouth. Not too much now.

Now tease it with bitters.

Don't shake it or stir it or anything. None of those little cherries made by staunch men in lab coats bearing bottles of bleach. I heard when I was a kid that those things don't digest for seven years and I believe it. Anyway, that would be a bit frufru, don't you think? Just serve it with a warm smile and say, "Enjoy!" Just like my grandmother did. Believe me, your guests will just be delighted. You will make them really happy with this delicious, old-fashioned drink that you prepared lovingly.

Chapter 11. CLASSES RELOADED

DO OR DIE:

- Learn more about classes.
- Eat more, drink more, be merrier.

The Classes chapter in this book covered a lot of the basics about writing classes and making objects (instances of classes) and using variables and methods. We discussed what makes up a class, including a package, visibility, methods, and fields.

There was a lot of stuff in that last chapter that we had to cover before anything else could start making sense. But the Java class is the heart of the language. It is actually all that you ever do, write classes. So if we said everything about classes in one chapter, that chapter would be like one giant, super-long, book-length chapter, 900 hours long, like LoTR RotK, so action-packed that it doesn't even afford time to run for a refill on your mtn dew.

So, in this chapter we'll expand and deepen our understanding of classes, and demonstrate the true power of this space station. You might implement the things we cover here, such as enums, static imports, and constructors, depending on your design. We are going to do this in a kind of circular manner, coming back around like a boomerang and perhaps re-touching on things at a greater depth.

Using External Code in Your Programs: import

Almost every time you write a new class file, you will want to give that class the powers provided by some existing external file that lives in another package. You can do that in one of two ways:

1. Specify the fully qualified class name every time you reference the class.
2. Import the class or the package containing the class, and then simply reference the class using its name. You do this using the keyword import.

If you have used Microsoft .NET, you should be familiar with this concept. In C#, you might write the following:

```
using System;

class MyClass {

  public static void Main() {

    Console.WriteLine("smeagol was here");

    }

}
```

That's how you specify that you are going to use one or more of the classes in the System namespace in C#. C# namespace ~= Java package.

If you have used C#, you will find not just this section, but much of this book fairly easy going, as Microsoft has helped itself to very generous portions of Java's syntax in creating C#. You'll find that with a little tuck here and a little nip there, some of your Java code will compile directly to MSIL (Microsoft Intermediate Language used in .NET).

When writing code, the choices for using external classes boil down to the following options.

First, your code without import:

```
net.javagarage.monsters.Vampire vampire = new

          net.javagarage.monsters.Vampire();
```

Second, your code with import:

```
import net.javagarage.monsters.Vampire

...

Vampire vampire = new Vampire();
```

Note that you can import all of the classes in a package using the common wildcard character *. To import all the classes and interfaces in the monsters package, write

```
import net.javagarage.monsters.*;
```

That way, your code can do this all day long (assuming your "monsters" package has classes named Mummy and Werewolf in it):

```
Vampire vampire = new Vampire();

Mummy mommy = new Mummy();

Werewolf werewolf = new Werewolf();
```

Look, ma—no fully qualified class names!

Keep this in mind: trying to decide whether to import all of the classes in a package using * or importing only the class from that package that you need is not a very interesting dilemma. In fact, it is no dilemma at all. They both perform the same and result in the same class file size. When you import all of the classes in a package using *, the Java compiler does not do what you ask. Instead, it determines what classes in that package your code actually references, and imports only those. So don't worry about the efficiency of code that uses the * wildcard to import. My recommendation is to use the wildcard if you are lazy, or if you need more than one class from the same package. If you have a number of less-experienced developers on your team (or you yourself are easily confused), you might want to explicitly import each class, even when they're in the same package, to make the code more readable.

FRIDGE

Many IDEs will help you organize your imports so that you don't have to think about it too much. In Eclipse, for example, you can right-click anywhere on your source code file and choose Organize Imports—this kind of thing will help keep your code clear and clean.

Automatically Imported Classes

One last thing about packages. You might have noticed that you can just write, compile, and execute this code:

```
package com.monsters;

public class Vampire {
public static void main(String [] args) {
String music = "punk";
System.out.println("Tony Blair's fave music is "+ music);
}
}
```

A question arises here. Why don't we have to import String? That is a Java class made available in the API, and it is in the java.lang package. You don't have to import classes in java.lang package: all of the classes in that package are always automatically imported for you. That means that you could write this to get the same result as in the preceding:

```
java.lang.String music = "punk";
```

Static Imports

Static imports are new with Java 5.0. In previous examples, we have used the class name as a prefix to calling the static methods of the Math class, such as Math.random(). This is okay, but it is also unnecessarily verbose.

To facilitate ease of use in other classes with the static methods and static variables, you can add the static keyword to the import directive and import them all so that you don't have to use the class name as a prefix to calling them anymore.

So, you have a choice between writing this (which is how we used to have to do it)

```
public class MathClient {
   public void x() {
      double d = Math.random();
      //...blah blah
   }
}
```

and writing this:

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

public class MathClient {

   public void x() {

      double d = random(); //look! no "Math."!

      //...blah blah

   }

}
```

Using static imports is meant to help clear up code that becomes overburdened by numerous, repeated calls to constants that clutter up your code.

StaticTestConflict.java

```
package net.javagarage.statimp;


import static net.statimp.MyConstants.*;

import static net.statimp.MyOtherConstants.*;


public class StaticTestConflict {


   public static void main(String[] args){


      System.out.println(SOME_PROPERTY);


   }

}


class MyConstants {


   static String SOME_PROPERTY = "MyConstants value";

}


class MyOtherConstants {
   static String SOME_PROPERTY = "MyOtherConstants value";

}
```

Note that it is okay to declare multiple classes in the same source file as long as only one of them is declared public. But you can't compile this code if you do not fully qualify the reference to the property with the class name as we have done. Here's what the compiler says about it.

```
reference to SOME_PROPERTY is ambiguous

both variable SOME_PROPERTY in net.statimp.MyConstants and

variable SOME_PROPERTY in net.statimp.MyOtherConstants match

System.out.println(SOME_PROPERTY);


1 error
```

You might think that you would not be able to use static imports to import two separate methods with the same name in different packages. For example, the Integer compareTo() method and the Byte compareTo() method both have the same names. However, they have different signatures: the compareTo() method from Integer accepts an Integer, and the method with the same name in Byte accepts a byte. Because of this, your namespaces stay separate, and the following lines of code are legal:

```
import static Integer.compareTo();

import static Byte.compareTo();
```

The runtime will be able to sort them out.

So. Yeah. That's all I want to say about imports right now. I hope that that answers all of your questions about this stuff. If, after this, you have more questions about packages and imports, you might consider getting a Zoloft script; as my cousin August says, "Better living through chemistry…" Otherwise, try taking up a hobby, like gardening, or hot rod racing.

Constructing Objects

When you define a class, you can define one or more constructors for it. A constructor provides a way of creating (constructing) an instance of its class.

When you use the new keyword to create an object, you are calling the constructor of that class whose argument list matches the arguments provided in the call.

The following code calls the default (no-arg) constructor for the JButton class:

```
JButton btnSave = new JButton();
```

Constructors are special methods that must match the names of the classes in which they are defined, and which have no return type. They are used to control how objects are created, and often, to initialize fields to the values specified in the argument list.

When you run the preceding code, the runtime determines if there is sufficient memory space to create the object and its data. If there is not enough space, the runtime tries to find any candidates for garbage collection. The garbage collector destroys items with no references, because they can no longer be used in a program's life. If there is still not enough space, an OutOfMemoryError is thrown.

```
package net.javagarage;

public Employee {

  private String name;


  public Employee(String nameIn) {

     this.name = nameIn;

  }

}
```

FRIDGE

We haven't discussed superclasses (a.k.a. parent classes) yet. But for now, just know that the java object tree is hierarchical. A class can extend another class, which means that it can inherit its non-private data and functionality. Every Java class in the world automatically inherits from java.lang.object. That means that any class you write will be able to do what Java objects do. It means that any class you write will inherit the methods defined in that class, such as equals, hashCode, wait, and notify.

The code in bold represents the constructor. It has no return type and its name matches exactly the name of the class.

It accepts one argument, a String. The field name is then initialized to the value of the parameter nameIn. The superclass constructor with a matching signature is called. Then the constructor exits and the object is ready for use. Let's see how this works.

Constructors.java

```
package net.javagarage.demo.classes;

/**
 * Demos use of constructors, and how
 * they interact with superclass constructors.
 * @author eben hewitt
 */
public class Constructors {

    public Constructors() {
        System.out.println("Superclass constructor!");
    }

    public static void main(String[] args) {
        //what does this print?
        //in what order?
        SubClass sa = new SubClass();
    }
}

class SubClass extends Constructors {
    //overriding the no-arg constructor
    public SubClass() {
        System.out.println("Subclass constructor!");
    }
}
```

So how many constructors are called in the preceding code? Here is a misleading clue: the code outputs the following:

```
Superclass constructor!

Subclass constructor!
```

The answer is three, because every class in Java is a subclass of java.lang.object—so the matching constructor in the object class gets called too. When you construct an object, all matching constructors are called down the hierarchy.

No-Arg Constructors and Overloaded Constructors

The constructor called in the preceding code is called the no-arg constructor. This means that it doesn't accept any arguments. You can explicitly write a no-arg constructor in your class definition. If you don't, a no-arg constructor is provided for you that does nothing but create an empty object of the specified type. That constructor is called the default constructor.

The default constructor is *not* provided if you provide any constructor definition of your own. If you want to provide a no-arg constructor, you have to define it yourself. Here's how:

```
package net.javagarage;

public Employee {

  private String name;

  public Employee() {}

  public Employee(String name) {

     this.name = name;

  }

}
```

The Employee class here defines a public, no-arg constructor. Because it provides a constructor that accepts a String argument, which it assigns to the name field, it must explicitly define a no-arg constructor if you want users of your Employee class to ever be able to create Employees without also initializing the name field concurrently.

After an object has been created, its variables must be initialized (remember, that's the second purpose of constructors).

Although you can create a JButton using the no-argument constructor, as we did earlier, the JButton class defines several other constructors (remember, classes can define as many constructors as they want—or none). One of the other constructors takes a String argument that initializes the field that represents the text printed on the button. So, calling the following constructor:

```
JButton btnSave = new JButton("Save");
```

kills two birds with one stone.

You don't have to call the setText(String s) method now, because the constructor that accepts a String argument has already done that for you. If there is not a constructor defined that matches the argument list that you try to use to create an object, a NoClassDefFoundError is thrown.

To overload a constructor, just make another one that has a different argument list than any other constructor in that class, just like overloading a regular method. There are many different overloaded constructors defined in the Java API. The String class, for example, has 14 constructors.

You can create a constructor with a variable-length argument list, as shown here.

VarArgConstructor.java

```
package net.javagarage.demo;

  //demonstrates ability to create

  //constructor with variable length arguments


public class VarArgConstructor {


  //vararg constructor

  public VarArgConstructor(Integer...ar){

    System.out.println("Object created!");


    for (Integer i : ar) {

        System.out.println(i);

    }

  }


  public VarArgConstructor(Integer i, Integer j){

    System.out.println("Constructor with two Ints

      called!");


  }


  public static void main(String...args) {

    VarArgConstructor va = new

            VarArgConstructor(1,2);

  }


`
}
```

Note that the correct constructor is called in the preceding class. If you replace the object creation statement with one that passes three or five ints into the constructor, the vararg constructor will be called. Because we pass in two ints, "Constructor with two Ints called!" is printed.

Private Constructors

You can define a private constructor as well. But just like in methods, the private access modifier indicates that that constructor can only be called from within the class itself. This is often used to purposefully disallow clients of the class to create instances of the class. For more about how and why to do this, visit the Singleton pattern, discussed in the "Design Patterns" chapter in *More Java Garage*.

Calling Other Constructors with this

The Java keyword this is used to refer to the current instance of a class. It operates like the VB Me keyword. But there is another use for this. Writing overloaded constructors could easily mean repeating the same code over and over. To save you from having to do that, you can call this as you would a method, and the invocation will refer to a constructor defined in your class that matches the parameter list.

If you invoke another constructor within the same class using this, it must be the very first thing you do in your constructor.

ThisConstructor.java

```
package net.javagarage.demo;

public class ThisConstructor {

  private boolean isInitialized;
  private String name;
  private int theID;

public ThisConstructor() {
  isInitialized = true;
  System.out.println("no-arg called");
}

public ThisConstructor(int theID) {
  super();
  this.theID = theID;

}
public ThisConstructor(int theID, String name) {
  //call constructor that takes int
  this(theID);
  //set the string value
  this.name = name;

  System.out.println("int, String arg called");
}

public static void main(String[] ar) {
  ThisConstructor ts = new ThisConstructor(555,
                          "Mr. Ed");

  }
}
```

The output is

```
no-arg called
int, String arg called
```

Static Constructors

This is kind of a misnomer. You don't get a static constructor by adding the static keyword to a regular constructor. You just write the keyword static and then curly braces. Inside the curly braces, write the code you want to execute.

Static constructors inside the class that contains the main method are executed even before the main method, when the class is initially loaded in the classloader.

You can define multiple static constructors. *They will be executed in the order in which they appear in source code*. The following code shows the order in which the constructors are executed.

Static Constructors.java

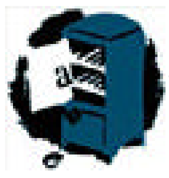
```
package net.javagarage.demo.classes;

/**

 * Demo order in which constructors are called.

 * @author eben hewitt

 */

public class StaticConstructor {


static {

  System.out.println("Inside static constructor ONE.");

}


//regular no-arg constructor

public StaticConstructor(){

  System.out.println("Inside regular constructor.");

}


public static void main(String[] args) {

  StaticConstructor c = new StaticConstructor();

  System.out.println("Inside main.");

}


static {

  System.out.println("Inside static constructor TWO.");

}

}
```

And here's the output:

```
Inside static constructor ONE.

Inside static constructor TWO.

Inside regular constructor.

Inside main.
```

Static constructors are rarely used in regular business application programming. They are used for loading native code, discussed in detail in *More Java Garage*.

Using Enums



FRIDGE

The code in this book was compiled and executed using the JDK 5.0 alpha and beta releases, and it is possible that there are changes between these and the final release. If you want to switch between different compilers, you can use the `- source 1.5` switch to the `javac` command.

Enums are new with Java 5.0. Enum is short for enumeration, and enums are useful when you want to define values that are meaningful to the user (in this case, other programmers) that can be switched against. The Enum is defined in the java.lang package.

When to Use an Enum

Programmers may be familiar with the enum, which programmers may be familiar with from C, C++, C#, or Pascal. A simple enum looks like this:

```
public enum Season { winter, spring, summer, fall }
```

An enum is an object that defines a list of acceptable values from which the caller can choose. It defines zero or more members, which are the named constants of the enum type. No two enum members can have the same name.

Java programmers have heretofore had to roll their own poor-man's enums by using constant- style names with `int` values, like this:

```
public interface Season {

   static winter = 0;

   static spring = 1; //etc..

}
```

Every field declaration in the body of an interface is implicitly public, static, and final. So the interface is used sometimes for constatnt declarations. If you want to make those constants available to your class, you implement the Season interface.

There are problems with having to do so, however. The first is that you need to write a lot of code to handle the situation, and rolling your own usually means that you are using an interface, as we do here, for something that it isn't intended for. This can cause clutter and confusion in your program.

Note that you can declare a public enum within a public class source file (remember that you can't declare more than one public class in the same source file).

You can also declare a public enum in its own .java file and compile it. Here is the enum replacement for our earlier interface:

Season.java

```
public enum Season { winter, spring, summer, fall }
```

That's all we have to do. The values for each of the seasons will be automatically assigned a numeric value, starting with 0, and incrementing one for each remaining enum value.

Let's look at the easiest way to implement the new enum construct.

EnumDemo.java

```
package net.javagarage.enums;

/*
We can loop over the values we put into the enum
using the values() method.
Note that the enum Seasons is compiled into a
separate unit, called EnumDemo$Seasons.class
*/
public class EnumDemo {

    /*declare the enum and add values to it. note that, like in C#, we don't use a ; to
➥  end this statement and we use commas to separate the values */

    private enum Seasons { winter, spring,
     summer, fall }

    //list the values
    public static void main(String[] args) {
        for (Seasons s : Seasons.values()){
            System.out.println(s);
        }
        }
    }

}
```

Running the preceding code outputs the following, as you would expect:

```
winter
spring
summer
fall
```

Switching Against Enums

The following code demonstrates how to switch against the values in an enum, which is what enums are commonly used to do:

```
package net.javagarage.enums;
/*
File: EnumSwitch.java
Purpose: show how to switch against the values in an enum.
*/

public class EnumSwitch {

    private enum Color { red, blue, green }

    //list the values
    public static void main(String[] args) {
        //refer to the qualified value
        doIt(Color.red);

    }
    /*note that you switch against the UNQUALIFIED name. that is, "case Color.red:" is a
➥  compiler error */

    private static void doIt(Color c){

    switch (c) {
    case red:
        System.out.println("value is " + Color.red);
        break;
    case green:
        System.out.println("value is " + Color.green);
        break;
    case blue:
        System.out.println("value is : " + Color.blue);
        break;
    default :
        System.out.println("default");
    }
    }

}
```

Adding Fields and Methods to Enums

You can add fields and methods to enums, just like you would a regular class. The following shows that you can use an enum much like you would a class. You can define static or non-static fields and methods, and define constructors. You can define different levels of visibility, as with a class.

EnumDemo.java

```
package net.javagarage.enums;

/*
File: EnumDemo.java
Purpose: show how to use an enum that also defines its own fields and methods
*/
```

```
public class EnumWithMethods {

//declare the enum and add values to it.

public enum Season {

    winter, spring, summer, fall;

    private final static String location = "Phoenix";

    public static Season getBest(){
        if (location.equals("Phoenix"))
            return winter;
        else
            return summer;

    }

    public static void main(String[] args) {

    System.out.println(Season.getBest());
    }
}
```

Notice one limitation of enums that is not often present in other features of the Java language, placement of the values list is limited to immediately following the enum declaration. That is, you'd see a compilation error if you switched the following two lines of code in the preceding example:

```
private final static String location = "Phoenix"; //wrong!

winter, spring, summer, fall;
```

The enum of seasons comes following the field here, which is illegal. The compiler would let you know an identifier is expected.

Enum Constructors

You can create an enum with a constructor as well, just like you would in a regular class. The following code shows how.

```
package net.javagarage.enums;

public class EnumConstructor {

    public static void main(String[] a) {

        //call our enum using the values method
        for (Temp t : Temp.values())
            System.out.println(t + " is : " + t.getValue());
    }

    //make the enum
    public enum Temp {
        absoluteZero(-459), freezing(32),
        boiling(212), paperBurns(451);

    //constructor here
    Temp(int value) {
        this.value = value;
    }

    //regular field—but make it final,
    //since that is the point, to make constants
    private final int value;

    //regular get method
    public int getValue() {
    return value;
    }

    }
}
```

The output is the following:

```
absoluteZero is : -459

freezing is : 32

boiling is : 212

paperBurns is : 451
```

Now. Just because enums share the capability to define methods and fields and constructors like regular classes does not mean that defining and using all of those gadgets is a good idea. If you want that kind of functionality, you probably should write a class. Enums are very useful as a shortcut for initializing int variables in a list of constants when the values are meaningful only within the context of the other values. They're perfect for that. They're popular in many programming languages. But if you need a class, make a class, man, not an enum.

Methods Allow Variable-Length Parameter Lists

Varargs. Yes, that is what passes for a word these days (did you ever read Ray Bradbury's "The Sound of Thunder"?). The vararg is a thing that they have in C# and other languages and it means that you stomp your foot and say, "This method has a bunch of parameters and *I just don't know how many parameters a client might want to pass it, all right?*" It means "variable length argument list."

This is an easy concept to start working with. The following class defines a method that accepts a vararg, and invokes the same method passing it first several arguments and then fewer arguments. The arguments come in as an array.

```
package net.javagarage.varargs;

/*

Demos how to use variable length argument lists.

*/


public class DemoVarargs {


public static void main(String...ar) {


  DemoVarargs demo = new DemoVarargs();


  //call the method

  System.out.println(demo.min(9,6,7,4,6,5,2));

  System.out.println(demo.min(5,1,8,6));


}


/*define the method that can accept any number

of arguments. do so using the ... syntax.

Just calculates the smallest numeric value in the argument list.

*/

public int min(Integer...args) {

    boolean first = true;

    Integer min = args[0];


    for (Integer i : args) {

      if (first) {

        min = i;
```

```
            first = false;

          } else if (min.compareTo(i) > 0) {

            min = i;

          }

        }

    return min;

}

}
```

The output is

```
2

1
```

Varargs are shortcuts that prevent the programmer from having to define the method parameter as an array. You'll notice that because the main method that starts Java applications is defined to accept an array of String objects as arguments, it can also be called using this syntax, like this:

```
public static void main(String...args) {...}
```

But that is not because the main method wants an array because it just really enjoys having arrays around to hang out with. The String[] argument is required because we don't know at compile time how many arguments the user will pass into the java command.

The benefit to varargs is that your intentions are explicit (always good), and that you don't have to worry about falling off the end of the array. When combined with the for each loop as earlier, varargs are very easy to use.

So obviously, you want to accept an array as a parameter if you're going to be doing something with arrays. Use varargs only to indicate a variable length parameter list.

Note that varargs are new to SDK 5.0, so you need to compile for that version of Java to use them.

Wrapper Classes

You can't call a method on a Java primitive, such as an int or a double. And sometimes that's a problem because you would like to be able to. Sometimes, you need to hold a list of these kinds of values, and you want to use the classes in the Java Collections Framework; the problem here is that none of those data structures, such as ArrayList, Hashtable, or Vector, will hold primitive types. They only hold objects.

So what's a girl to do? Use the wrapper classes that come with the Java API. There is a class in the java.lang package that corresponds to each of the primitive types. Table 11-1 shows these classes.

The Primitive Wrapper Classes

WRAPPER	PRIMITIVE
Boolean	boolean
Byte	byte
Short	short
Character	char
Integer	int
Long	long
Float	float
Double	double

The preceding wrapper classes allow their corresponding primitive types to be used as objects. They provide the following methods that make working with them easy and fun. The wrapper classes share these methods in general. Let's take one wrapper, Boolean, as an example, from which to extrapolate.

First, the constructors you can use to make Boolean wrappers:

- public Boolean(String b) Creates a Boolean object representing the value true if the argument is "true" (ignoring case).
- public Boolean(boolean b) Creates a Boolean object corresponding to the value of the passed Boolean primitive.
- boolean booleanValue() Call this method on an object of type Boolean when you want to get its value as a primitive. The corresponding methods in the other classes include intValue(), doubleValue(), and so forth.
- public int compareTo() Compares the values of two wrappers of the same type. It returns 0 if this object represents the same primitive value as the argument.
- public static boolean parseBoolean(String s) Takes a String argument and returns its Boolean value.
- static Boolean valueOf(boolean b) Returns an instance of the Boolean object corresponding to the boolean argument. This method is the sister of booleanValue(), which you use to get a primitive from a wrapper—the xValueOf() methods get a wrapper from a primitive. This method is overloaded to accept a String argument as well.

Boolean has three static fields: TRUE and FALSE, which represent their primitive counterparts, and TYPE, which represents the primitive type boolean.

I only show the Boolean wrapper class here as an example, because the other wrapper classes work very similarly, and share similar methods. The Integer wrapper class, for example, features an intValue method, and the Long wrapper class features a longValue method.

The wrappers are necessary when you need to use your primitive values with some API that only will accept working with objects. However, because that kind of functionality is so fequently needed in object-oriented programming, a new feature was added to Java 5.0; autoboxing, which we discuss next.

Autoboxing

As wonderful as the primitive wrappers are, they often mean we end up creating a lot of objects (say with valueOf(int i)), passing them into a list, and then putting them back into their primitive values again with intValue()) when we drag them out of the list.

Autoboxing is a feature new to Java 1.5 that takes care of this for us. Here's how it works in a nutshell. Create a primitive type. Pass it as an argument to a method that expects an Object. It magically works. The JVM converts the primitive to its corresponding wrapper on-the-fly for you.

Autoboxing wouldn't be complete without auto-unboxing. As you might guess, this means that when you ask for a primitive directly from an Object wrapper, ye shall receive.

Because this concept is fairly easy to understand and appreciate, let's go straight to the videotape.

AutoboxAssigment.java

```
package net.javagarage.autobox;

public class AutoboxAssignment {

  public static void main(String[] args) {
    //whoa! primitive reference to object
    int bottlesOfBeer = new Integer(99);

    //the other way 'round
    Float temperature = 98.6f;

  }
}
```

This simple test shows that we can move effortlessly between primitives and their wrapper classes.

FRIDGE

This is cool. It does not mean that you can start calling methods on 'primitives' as you can in some languages I won't mention (okay, C#). The old int.parse() trick won't play in this town….

This is not license to do whatever wacky thing you want, however. You can't call methods on a primitive and expect it to know what you're trying to do.

For example, say we try to call one of the Double wrapper class methods on a double primitive:

```
double d = 45D;
d.isNaN(); //Wrong!!!
```

The compiler sez

```
double cannot be dereferenced
```

Just so you know it when you see it, let's try another test, a little more complicated.

AutoboxDemo.java

```
package net.javagarage.autobox;

import java.util.*;

public class AutoboxDemo {

public static void main(String...args) {

   /**Make a map that holds String keys and Double values. Collection types can only
   hold objects, not primitives. */
    Map<String,Double> weatherForecast = new

TreeMap<String,Double>();

    weatherForecast.put("Monday", 65);
    weatherForecast.put("Tuesday", 68);
    weatherForecast.put("Wednesday", 70);
```

```
        weatherForecast.put("Wednesday", 70);

       System.out.println(weatherForecast);

   }

}
```

This class creates a Map, which holds key/value pairs, and passes in double primitive values. Even though the Map only holds objects, they are converted on-the-fly and we don't have to deal with object creation for each one of those doubles ourselves.

This functionality is convenient. I could show you the old, yucky way we had to do this, but I will spare you the gory details. They're just too awful…

If you enjoy torture, please refer to Chapter 18, "Casting and Type Conversions," for a little more on this.

The Class Named Class

There is a class called Class that represents the possibility of making and loading classes in the Java Programming Language. You can make instances of the Class class for the classes and interfaces in a running Java application.

The primitive Java types and the keyword void are represented as Class types.

Class has no public constructor. That is, you cannot write this: Class clazz = new Class();. You can get a Class object for a given class in three different ways.

First, you can use the getClass() method of java.lang.Object. If you have an instance of Object called obj, you can write the following:

```
Class clazz = obj.getClass();
```

Second, you can use the class literal, like this:

```
Class clazz = java.lang.String.class;
```

Third, you can get it by passing a String containing the fully qualified name of the class of which you want a class instance, like this:

```
try {
    clazz = Class.forName("java.lang.String");
} catch (ClassNotFoundException e) {
}
```

Getting the Package of a Class

You can also get the package that a class is in by calling the getPackage() method on the Class object:

```
Class clazz = java.lang.String.class;

Package package = clazz.getPackage();

          //represents a package

String pName = package.getName();

          // java.lang
```

If you write a class without declaring a package for it, it will be created in the default package, which has no name. If you call getPackage() on the Class object of such a class, it returns null. It also returns null for a primitive type class, or an array. Like this:

```
Class clazz = SomeClass.class;

packg = clazz.getPackage();          // null


packg = char.class.getPackage();     // null

packg = char[].class.getPackage();   // null
```

Getting an Object's Superclass

You can get an object's superclass by calling the getSuperclass() method on the Class object.

```
Object obj = new String();

Class supr = obj.getClass().getSuperclass();

              // java.lang.Object

// The superclass of java.lang.Object is null

obj = new Object();

supr = obj.getClass().getSuperclass();     // null
```

Listing the Interfaces That a Class Implements

We will find out about interfaces later. For now, know that they are a contract containing methods that you can force a class to implement. Many classes in the Java API implement interfaces. You can list all of the interfaces implemented by a class with the following code:

```
Class cls = java.lang.String.class;

Class[] interfaces = cls.getInterfaces();
```

This stores the Class objects of all of the interfaces String implements in an array.

Note that if the reference type of your item is an interface, the call to

```
intfc.getClass.getSuperclass();
```

returns the object's superclass.

<shameless-plug>This is enough for now about the very cool java.lang.Reflect package. If you are interested in finding out about classloading, dynamic proxy creation, how to dynamically invoke a method, and even how to change the programmed visibility of a member using the reflection API, you'll have to get the sequel to this book, the inventively titled *More Java Garage*. </shameless-plug>

Now that we are on more solid ground with the fundamental structures in Java, it will get a lot easier now to start working with more code, seeing more useful examples, and starting to do some fairly exciting things.

The Java world gets markedly more complex at this point, and that sophistication will pay off for you in power.

We won't waste any time. The next chapter introduces one of the three cardinal precepts of Object-Oriented programming: for the sake of clarity, they are Faith, Hope, and Inheritance….

Okay, they are Encapsulation, Polymorphism, and Inheritance.

That somehow sounds less dramatic, though. Onwards and upwards.

Chapter 12. INHERITANCE

DO OR DIE:

- Learn about Inheritance.
- Have some food.

PsychoMan PsychoMan PsychoMan

Mr Hand (furious): Just what do you think you're doing?

Jeff: Learning about inheritance, and having some food.

The thing is this: classes can inherit functionality from other classes. It is kind of important. So we're going to talk about it for a while.

Getting Your Inheritance

I bet that "getting your inheritance" pun has been made a thousand times. I think I will leave it in anyway and try to pass it off to you like, "Yeah, we both know that that dumb joke has been made so many times. But it's not simply some somnambulist tech writer digging into the dregs of his already shallow literary toolbox and pulling out yet another overused bad pun. Rather, in my deft hands, it's another brilliant and hilarious example of a master of 'high camp' confidently practicing his craft."

I'm not sure I like where this is going. Let me start all over. Okay.

Java classes all participate in inheritance, whether they like it or not, and whether they know it or not.

No! Somehow that comes off as angry—too angry. I'm not *angry* about inheritance. Geez. Try again:

Inheritance in Java, as in life, is when you get something for nothing. [This is [Following is a Continuation Paragraph...] too preachy, too absent, and drippy. Note to self: Don't liken things to "life" as if "life" is somewhere else and you know something about it. This can be annoying, despite the incredible insight. Just tone it down a bit, okay? Is that so hard? Gawd.]

Every Java class has an implicit superclass: `java.lang.Object`. That is to say, the class called Object in the `java.lang` package (which you recall is automatically imported for you into every class) is the parent of every other class in Java that has ever been, or will ever be, made. Any person we make (that is, any object of the class Person we instantiate) is an object.

It makes sense that Person needs to know something other than just how to be a Person—it needs to know how to live inside Javaland. Every object of every class needs to know that information. It needs to be able to be constructed in the first place, and it needs to be able to participate in threads and such. That behavior is necessarily shared by every Java class. So for that reason, `java.lang.Object` is the superclass of every other Java class, which we can call the subclasses of `java.lang.Object`.

There are a few terms that we can use to mean the same thing:

> Superclass = A class that is higher up in the hierarchical inheritance tree.
>
> Parent class = Superclass.
>
> Subclass = A class lower in the hierarchical inheritance tree than another class.
>
> Child class = Subclass

So Person is a subclass of Object. Person is a child class of Object. Object is the superclass of Person. Object is the parent class of Person. We'll use these terms interchangeably to try to keep you awake.

If you want to, you can declare that your class will inherit the data (members) and functionality (methods) from one other class. You do so using the `extends` keyword.

But wait! When we declared the Person class, we did it like this:

```
public Person { ... }
```

Where's the part about Object? Where's this "extends" you speak of? They are implicit. The following code is equivalent to the preceding code:

```
public Person extends java.lang.Object { ... }
```

Classes can have only one immediate parent class. The following is *not legal*:

```
public Person extends LivingCreature, Homosapiens {

... }

//illegal! busted!
```

Multiple inheritance is a thing allowed in C++, and the Java language designers thought that it made code very confusing and difficult to debug and maintain. So they decided not to allow it at all. Internet people hanging around in forums complain on occasion that disallowing multiple inheritance makes it hard to do certain things in the language, and that it was a bad decision. I say, "Ah, hog wash." Java allows interfaces, which you can read about in the exciting chapter 17, intriguingly named, "Interfaces." You can implement multiple interfaces, and that kind of solves that problem, which I don't necessarily recognize as a problem, because in my experience, if you've got a good design, you won't feel any need to inherit from more than one class. In fact, I'll play this card: If you find yourself at your desk thinking, "Man, I can't do what I need to this way. I wish I could extend more than one class…," you should rethink your design. Really. The issue of only allowing single inheritance is a non-issue.

FRIDGE

Remember that children cannot see their parent's private members. In psychiatric circles such an episode is referred to as "The Freudian Scene," and can have long-lasting effects on the child. :)

To help you think about this kind of thing, make sure to look at the "IS-A Thing" and "HAS-A Thing" sections later in this chapter. Meanwhile, back at the ranch…Because every class extends Object, they each get to call all of the *public* methods that Object implements, as if they were Object itself.

Object has the following *public* methods:

```
toString(), equals(), hashCode(), getClass(),

notify(), notifyAll(), wait().
```

That means that you can call those methods on every Person object, even though they aren't defined in the Person class. You get that functionality for free.

Here are some things to remember about inheritance. If a member has protected visibility, any subclass of that member's class can access it because protected = package + children. A subclass can only see the protected members through inheritance. Private members are not visible to subclasses.

Methods Can Override Superclass Methods

Sometimes, what you get from a parent isn't exactly what you wanted or needed. Perhaps you can relate to that. :(. In that case, you might want to make a new implementation of the method you inherited from them. In Java, this is called overriding. In the real world I believe it's called therapy.

You can implement a method that is defined in a superclass however you want to. Nevertheless, the method signatures must match when you do this. If the signatures don't match, you aren't overriding anything; you're simply making a new method that has the same name, but different signature. If the signatures don't match, the compiler will be able to sort out which method you mean to call when. But if a subclass overrides a method that is implemented in a superclass, the runtime will call the subclass method on all instances of the subclass type. It just slides it right in there and replaces it. It makes the superclass method invisible to the subclass. Here's an easy example to demonstrate.

Let's take our Person class that we already have and extend it to make a Programmer class. We'll then extend Programmer with JavaProgrammer. The code, as always, is heavily commented so you can see what's happening line by line.

Programmer.java

```
package net.javagarage.demo.inherit;

/*

 * We'll just do some dumb printlns.

 * Notice that Person is in another package,

 * so we either need to import it or simply write

 * out the fully qualified name like this in the

 * extends statement.

 */

public class Programmer extends

              net.javagarage.demo.classes.Person {

    String onlineName;


    void writeCode(String someCode){

        System.out.println("The code: " +

            someCode);

    }


    void slackOff(long duration){

        System.out.println(

            "Now slacking in the hallways " +

        "for the next " + duration +

            " minutes.");

    }

    void screwAroundOnInternet(String site){

        System.out.println("Now screwing around

            at " + site);
```

```
          at " + site);
      }

      void portProgram(String fromPlatform, String
           toPlatform){
        System.out.println(
           "Spending the rest of the year " +
        "moving the app from " + fromPlatform +
            " to " + toPlatform);
      }
}
```

JavaProgrammer.java

```
package net.javagarage.demo.inherit;

/*
 Define a subclass of programmer that is a specific
type of programmer that can do all the same things
as Programmer, and other, more specific things.
 */
public class JavaProgrammer extends Programmer {

      //this method is the override of portProgram
      //defined in the superclass Programmer
      void portProgram(String from, String to){
         System.out.println("No port necessary: " +
          "the same Java code runs on both " +
         from + " and " + to);
      }

      //now define some new method that
      //only Java programmers do
      void goToJavaOneConference(){
         System.out.println("Going to JavaOne...");
```

```
    }
}
```

OverrideDemo.java

```
package net.javagarage.demo.inherit;

/**
 * All this does is make an object of type
 * JavaProgrammer and demos how the methods it
 * inherits (and the one it overrides) work
 */
public class OverrideDemo{

    public static void main(String[] args) {

        //create new instance of JavaProgrammer
        //the JavaProgrammer jp = new
                JavaProgrammer();
        //inherited member from Person
        jp.name = "Jane";
        //inherited from Programmer
        jp.onlineName = "Veruca Salt";

        //defined in Programmer
        jp.writeCode("<blink>Buy now!</blink>");

        //inherited method from Programmer
        jp.screwAroundOnInternet("javagarage.net");

        //overridden method defined in both
        //Programmer and JavaProgrammer
        //but this object's reference type is
        //JavaProgrammer, so guess which method
        //gets called...jp.portProgram

        //("Windows", "Linux");


        //also from Programmer

        jp.slackOff(147);


        //just a regular method defined only

        //in JavaProgrammer.

        //well, my work here is done...

        jp.goToJavaOneConference();

    }

}
```

Here is the output of running the class.

```
The code: <blink>Buy now!</blink>

Now screwing around at javagarage.net

No port necessary: the same Java code runs on both

Windows and Linux

Now slacking in the hallways for the next 147 minutes.

Going to JavaOne...
```

This code clearly shows how things work when one class extends another and overrides its methods. I hope it clearly shows that anyway, because I think that's enough talking about it.

IS-A Thing

This is more of a design issue, not something that is proper to the Java programming language. But design is important. I think that elsewhere in this book I am going to suggest that the design is the implementation. Think about that for a moment. It washes over you like a mantra: the design is the implementation….

Java code is written as a collection of classes. Some programs are 1 class, many are between 100 and 1000 classes, and some are more than that. When you start dealing with a lot of classes, you need to think carefully about how to divide up your classes. One way to do that is the subject of this topic: inheritance. If class B inherits from class A, we say that B "is an" A. Say you have a class called Vehicle, which is the superclass of Car. Car inherits from Vehicle. Car extends Vehicle (those mean the same thing). A Car is a Vehicle. Car is a specific kind of Vehicle. A Cat IS A Feline. A Feline IS A Animal (excuse my grammar). You know that Animal is a superclass that Feline extends, and Cat extends Feline.

Thinking about IS-A relationships will help you get your inheritance issues straightened out, help you communicate with your analysts and other programmers, and help you make a well-designed program. Don't change the channel before reading HAS-A Thing. They kind of go together.

HAS-A Thing

The other kind of relationships in Java dare I say programming is the HAS-A relationship. HAS-A means the properties of this class. A Car HAS-A engine. A Car HAS-A collection of tires. A Car has a maximumNumberOfPassengers. A Car HAS-A type, such as sedan, compact, sporty, or whatever. You get the idea.

When designing your applications, say to yourself, "What is this thing? What properties does it have?" After you know the answers to the IS-A and HAS-A questions, you should first make sure that some of the properties don't actually belong to a superclass, and then you have your basic class skeleton. Like this:

```
public class Car extends Vehicle {

    private Engine engine;

    private Tire[] tires;

    private short maxNumberOfPassengers;

    private String carType;

    //... getter and setter methods

}
```

Now all you need to do is figure out what a Car does in the context of your application, and you're off.

Classes Can Ensure That No Class Inherits from Them

FRIDGE

The only modifiers allowed in a class declaration are public, final, and abstract. A class cannot be both final and abstract.

The final keyword can be applied to classes, methods, and variables, and means roughly the same thing for each.

Sometimes, you want to make sure that no class ever will be able to inherit functionality from your class. Because of polymorphism, which you can read about in Chapter 16, "Abstract Classes," and others, situations may arise when you want to ensure that the improper substitution of one class for another does not break your application or cause a security problem.

In these cases, you can declare a class to be final. You do so in the class declaration, like this:

```
public final class String {...}
```

That's right. The class java.lang.String is a final class. The designers of the Java language were prescient enough to see that allowing Strings to be subclassed would lead to security risks. We'll examine this in greater depth in Chapter 13, "Strings," but for now, it is enough to understand the following possible scenario: you run an application that gains access to the local file system by substituting a subclass of String with a rewritten value (containing a path) that the user might not knowingly provide. Because Strings are immutable (cannot be modified), and because they are final, this sort of thing does not happen in Javaland.

Unless you are writing fairly low-level code that you anticipate causing a possible security breach, you may not find many occasions to declare your classes final.

In the following chapter we switch gears a bit, and discover how to use Strings in some depth. Strings are among the most common programming elements you will use, and there are deceptively subtle aspects to their behavior. So we switch out of talking about classes in general and how they behave, and move into talking about a specific and very popular class that comes with the Java API: that lovable sequence of characters known as java.lang.String.

Chapter 13. STRINGS

DO OR DIE:

- Don't tell me what to do

A String is a set of zero or more characters. A String can be empty, or its reference can be null, or it can contain the entire text of *King Lear*. Strings are objects in the java.lang package, and are used so frequently that they are the only class in Java with their own special constructor syntax. In this topic, we put the moves on a few String objects and see how they like it.

Creating String Objects

Strings are objects. Their superclass is java.lang.Object (they inherit all of the properties and methods defined in the Object class, and then define some of their own). As we know, everything in Java is an object (except for the primitive types such as int, boolean, and char), and they all have corresponding object types that are often called "wrappers".

If you want to be a big nerd and make everyone hate you, create a String like you would a regular object:

```
String s = new String("Sucker!!!");
```

If, for some reason, you do not want to be a big gigantic nerd and force everyone to hate you, use the following simpler, more common, and more efficient syntax to create a String:

```
String s = "All the cool kids do it";
```

If Java provides us with this different constructor syntax, what is the problem with creating it the longer and more familiar way?

The easy reason is that no one does it, so it is unexpected behavior. And although you are lovely and singular and creativity is your strong suit, the declaration and initialization of your Strings is not the place to make your artistic statement. In other words, predictability is very good. Be predictable. The poor schmoes who inherit our code will be just a little less poor. Let's look at the possibly more important technical reasons in the following sections.

One String for the Price of Two

The following is called a String literal:

```
"This is da creeda of Jacques Derreeda";
```

It's a bunch of characters between double quotes. The compiler will automatically create a new String object for every literal it stumbles across in your program. Sound weird? Think about what happens when the new keyword is invoked? Java does what it's told and creates a new object on the heap. But what does it do when you use the new keyword *and* give it a String literal in the same statement?

Consider. You're the virtual machine. You're supposed to create a new object when you see new and you're supposed to create a new object when you see a String literal. So, you create *two* objects, just as you're told to do.

As you might imagine, specifying only the String literal, instead of using the new keyword, is far more efficient, because only the object literal is created.

That begs the question, "Do I have two Strings then, each with the same value?" The answer, as you might not expect, is, "No, you don't." See the "String References and Immutability" section to find out why.

Converting a Char Array into a String

Despite the performance penalty described previously, sometimes it is necessary to create Strings using the new keyword. And that's just fine. I'm not getting all weird about it. I'm just saying don't do it if you don't have to. It's perfect, for instance, when you have an array of characters, and you need to get that array into a String object. In these cases, you must use the constructor. Here's what to do:

```
char[] myCharArray = {'g','a','r','a','g','e'};

String myString = new String(myCharArray);

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

Note that you cannot convert directly to a String. That is, you *cannot* do this:

```
String myString = myCharArray; //won't compile
```

(See the "Strings and Security" section to find out why.)

Note that the String class contains a couple of different ways to do this; the other way also takes an int letting you specify the offset.

Converting a ByteArray into a String

There is also a String constructor that accepts a byte array. Say you use the New IO library introduced in Java 1.4 to read in the contents of a file to a ByteBuffer, like this:

```
    Row row = null;

  rows = Collections.synchronizedList

(new ArrayList(numberOfRecords));


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

    row = new Row(metaData);


    Charset charset = Charset.forName("US-ASCII")
```

```
      CharsetDecoder decoder = charset.newDecoder();

      byte isDeleted = bb.get();
      row.setIsDeleted(isDeleted);

      ByteBuffer bIn =
          ByteBuffer.allocate(LENGTH_OF_LINE);
      CharBuffer cb =
          CharBuffer.allocate(LENGTH_OF_LINE);
      cb.rewind();
      charset.newDecoder().decode(bb,cb,false);
      cb.rewind();
      String s = cb.toString();

      synchronized(datarows){
        //put record into list that the app will use
            rows.add(convert(row, s));
       }
    }
```

There is obviously code here that we haven't discussed yet. I'm just providing you with a code snippet that will read in an ASCII file, which only requires seven bits to render a character (not 16 like Unicode) into an array of bytes so that you can make a String out of it. If I didn't include the code, then it might be hard to see where you would get a byte array in the first place. But the above code example reads each line in the file, calling a separate method called convert as it uses the data to create a Row object.

Once we have that byte array, we can covert each line into a String. Like this:

```
Column c = new Column();
c.setSomeShortData(bb.getShort());

byte[] arrFieldName = new byte[c.getLenFieldName()];
bb.get(arrFieldName, 0, arrFieldName.length);
try {
    String data = new String(arrFieldName,"US-ASCI");
    c.setData(data);
} catch (UnsupportedEncodingException e) {
    e.printStackTrace();
}
```

If this all seems like too much right now, definitely do not sweat it. Onto the next. But if later you discover you need some byte-array-readin-string-makin' code, there you have it.

String References and Immutability

Something that is immutable means that it cannot be changed. One immutable thing in this universe is a Java String, which means, after it is assigned a value, that value never changes—*ever*. To which you might reply, "Oh yeah? Well what about this?" and then whip out some ninja code like this:

```
String s = "kitty";

s = s.concat(" cat");

System.out.println(s); //prints kitty cat
```

The concat method of the String class takes the string value of the object on which you call the method and concatenates (adds to the end) the literal parameter you pass the method. So that looks like all kinds of mutability, right?

Here's what really happens:

When we call concat, the virtual machine takes the value of the String s, and hangs onto it. Because it can't make any other character string except for "kitty" be the value of s, *it creates a second, new string object, assigns it the value of " cat", and then creates a third String object with the value of "kitty cat" (both literals mushed together), and then reassigns the object reference of s to that new object!* Pretty shifty. In other words, there are now *three* String objects, and s refers to the third one. Technically (this is at least trying to be a technical book), s didn't change its string value; it only changed its object reference, which gives the appearance of changing that String object's value.

So how many references are there?

Only one. s.

But—hey! What happened to the "cat" object? Isn't it on the heap? Yes. But nothing refers to it, and so it is lost forever, with no way for anyone to access it.

In the preceding example, we took a String and changed its characters by calling the concat method on it. That code looks like this:

```
s = s.concat(" cat");
```

Notice that we assign s again here. That is, we have a String s, and then assign it to the String object created when the concat() method call returns, which is how we get the new value "kitty cat". Can you tell what this code will output?

```
String s = "kitty";

s.concat(" cat");

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

If you haven't fallen asleep yet, good show. If you were paying careful attention, you might have guessed that this code prints "kitty" because we never reassign the object reference of s to point to the result of the method call. So it points to the same value it did when it was initialized. This code makes a String object for "kitty", and the concat() method makes two more (" cat" and "kitty cat"). No one ever asks the JVM about it, and now no one ever can.

Let's look at another example that is a little more complicated.

```
1. String s1 = "first";

2. String s2 = "second";

3. s1 = s2;

4. s2 = "new value for s2";

5. System.out.println("s1: " + s1);

6. System.out.println("s2: " + s2);
```

What output will this code produce?

The first two lines are very straightforward: they create two separate String objects with different values. At line 3, the reference for s1 is reassigned to point to s2, and there's the trick. Will line 5 print "second" or "new value for s2"?

If Strings are immutable, and s2 is given a new value on line 4, what happens? A new object is created and s2 is reassigned to that new object's address. s1 still points to the previous address for s1, and "second" is printed. Line 6 prints "new value for s2" as you would expect.

Why go to all of the trouble to make Strings immutable? One benefit is speed. The runtime can reuse a literal from the pool of Strings without having to create a new one, and because object creation is one of the more expensive endeavors the runtime will undertake, it's good to avoid it if possible. There is another consideration. Security.

Strings and Security

A Java String inherits from Object, but no class inherits from String, or ever will. That's because String is marked final, which means that it is the last of its kind. No class can ever inherit from a class marked with the final keyword in its declaration. I'll prove it. Try typing this in Eclipse (or whatever IDE or vi or what have you):

```
public class StringSubclass extends String { }
```

Here's what the compiler tells you:

```
The type StringSubclass cannot subclass the final

    class String

StringSubclass.
```

So what does that have to do with security? Well, our inability to subclass String means that we cannot get a value from a host and rewrite that reference using our custom (and malicious) subclass, substituting the runtime type for ours (which otherwise would be legal).

Luckily, in day-to-day business programming, I have yet to find myself at my keyboard thinking, "Damn! It will never work! If only I could subclass String…" Of course, I'm not in the business of writing viruses…

There is the related issue of converting other objects to Strings. Some things that you think might directly convert to Strings (such as an array of characters) do not. You must use the Java keyword new, and pass it the char array. Now why on earth would that be necessary? All of these hoops they make us jump through! Let's take a step back.

What happens when you do this:

```
String x = "This guy";

String y = "Some other guy";

x = y;
```

Two String objects are created: x and y. Then, the value of the y String is assigned to x (meaning, now x has the value of y). Big deal. Well, in order to see the big deal, let's recall that the value of an object is its reference! Sure, it might look to us like the value of s String is the characters we write, but that is a human-centric view (which I learned in graduate school is indelicate). Think about it from the virtual machine's perspective. We just replaced the object reference for x to point to y's address in memory. If we could assign arbitrary references (or at least practically arbitrary types, such as byte arrays) to Strings, we could easily create a buffer overflow, ruin the integrity of the application, and gain access to local resources that we were not supposed to have access to. This small march of progress is the origin of many viruses that exploit less strongly typed languages. I won't mention any names.

String Assignments

The StringBuffer will change the object's value without using the assignment operator. This is slightly different than you might expect given how String works. Here's an example:

```
String s = "Elvis";

s + " Presley";

//s = Elvis


StringBuffer sb = "Elvis";

sb.append(" Presley");

//sb = Elvis Presley
```

To make the String in the preceding code operate like the StringBuffer, you need to assign s to the *result* of the operation, like this:

```
String s = "Elvis";

s = s + " Presley";
```

Recall that the shortcut for primitives works for Strings here as well:

```
String s = "Elvis";

s += " Presley"; //same difference
```

You can read about StringBuffers a little later in this topic.

Doing Stuff with Strings

Now that it is clearer how Strings assignments and concatenation work, we will turn our attention to performing miscellaneous but frequently useful String operations.

Convert Different Variables to Strings

In this section we'll look at a couple of ways to convert different kinds of things into Strings.

Using valueOf()

Often you need to convert certain types to Strings in order to pass them to a method that accepts only a String. You can do this using the valueOf() method of the String class.

```
System.out.println(String.valueOf(Math.PI));

//prints 3.141592653589793
```

There are several valueOf() methods defined for the String class, each of which takes different parameters (which means that the valueOf() method is *overloaded*). All of them are static. That means that they are called on the class, not on any particular instance of the class. That is, you don't need to make an object, you can just write: String.valueOf(Math.PI); as in the preceding code. There is a separate valueOf() method for each of the primitive types, for char arrays, and one for Object. Math.PI returns a primitive double in case you were curious.

FRIDGE

Psst. Remember this from Chapter 9? When you pass an object of any type into System.out.println(), that method automatically calls the toString() method of that object. So, if you don't override the toString() method, you might get something less than useful printed out.

Using toString()

A second way to convert objects to Strings is by calling the toString() method on the object. An obvious difference from the valueOf() method is that for toString() you need to have an object, whereas valueOf() is static. Here's an example.

```
Object o = new Object();

String s = o.toString();

String d = new Double(12).toString();

System.out.println("s: " + s + " D: " + d);

//prints s: java.lang.Object@194df86 D: 12.0
```

We create a generic object and call its toString() method. Then, we create an object of type Double, and pass a primitive double to its constructor. Its toString() method returns "12.0". What is the difference between the two?

Object's toString() method prints the name of the class, followed by the @ sign, followed by a hexadecimal representation of the hashcode for the object, which means it prints a value equivalent to the following:

```
getClass().getName() + '@' + Integer.toHexString(hashCode());
```

You can do something more meaningful by implementing toString() in your own way (overriding it). Like this:

```
public String toString() {

  return "Some unique-ish field:" + this.someField;

}
```

PREV < Day Day Up > NEXT

Character Data Encoding

All Java character Strings are rendered as 16-bit Unicode. Unicode is a standard specifically created for computer processing of character data. Its purpose is to provide a consistent manner in which to encode character data, so that users throughout the world, writing in multiple languages, can share a single system.

The problem that Unicode solves is the problem introduced by ASCII character encoding, which represents our Latin alphabet beautifully, but nothing else. This is no longer an acceptable mode for character data exchange in the Internet age. ASCII Latin characters can be represented by only 8 bits each, but have a very limited range; Unicode represents all of the characters from every major written language in the world. It also includes many mathematical symbols, ideographs, technical marks, and other items. Doing so requires two bytes per character. One of the features of Java that propelled it to popularity in the early 1990s was its enforcing of the use of Unicode throughout the language in order to embrace the global economy.

FRIDGE

Go to www.unicode.org and click on the link labeled Code Charts. This will let you select from all of the graphematic symbols that Unicode supports, including Greek, bi-directional Hebrew, currency and safety symbols, and more. You can read the symbol charts to find out what code you need to call to print certain characters.

Most of the time you don't have to worry about this. But if you need to present data written in Arabic, or present special symbols, you can write Unicode directly by specifying its symbol.

All Unicode characters map to a single hexadecimal number. You can specify a Unicode literal using the escape backslash followed by a u. For example, \u0123 prints a ! (bang). Here is an example using some currency symbols.

```
String s = new String("Euro: " + "\u20AC");

s = s.concat("\nDollar: " + "\u0024");

s = s.concat("\nYen: " + "\u00A5");

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

This outputs the following:

```
Euro: €

Dollar: $

Yen: ¥
```

Note that many Unicode characters may not be printable on your particular system. Note, too, that any given symbol is actually implemented by a font vendor, and so there may be great variation between its appearance between systems.

You can read more about character encoding and its use in I/O operations in Chapter 22, "File Input/Output."

Converting a Byte Array to a String

Java provides a few different constructors in the String class to accommodate creating String objects from byte arrays. This is particularly useful when interacting with legacy systems or native code that provide you only with byte arrays for character data.

The following code snippet converts text between Unicode and UTF-8 using a byte array:

```
try {
      //go from Unicode to UTF-8
      String string = "abc\u5639\u563b";
      byte[] utf8 = string.getBytes("UTF-8");

      //go from UTF-8 to Unicode
      string = new String(utf8, "UTF-8");
   } catch (UnsupportedEncodingException e) {
      //handle. we'll talk about exceptions later
}
```

Comparing Strings

The following code, which demos how to compare strings in different ways, is pretty straightforward. There is one weird thing, however. The compareTo() method of the String class returns an int representing the result of the operation: it returns a 0 if the strings compared are equal, a value less than 0 if the first string is lexigraphically less than the string argument, and a value greater than 0 if the first string is lexigraphically greater than the string argument.

CompareStrings.java

```
package net.javagarage.demo.String;

public class CompareStrings {

  public static void main(String[] args) {
     String littleName = "elvis";
     String bigName = "Elvis";

       //is the character content of the string
       //this returns false, because of
```

```
      //case-sensitivity
    boolean isIdentical = littleName.equals(bigName);

      //check without case-sensitivity:
      //returns true!
    boolean b = littleName.equalsIgnoreCase(bigName);

      //check order of two strings
      //lowercase FOLLOWS uppercase
    int i = littleName.compareTo(bigName);
    if (i < 0) {
      // big comes before little
    } else if (i > 0) {

    } else {
        //they are the same
     }
  }
}
```

This is the big thing to beware of when comparing Strings: don't use the == operator unless you really mean it! That operator compares the object references of two Strings—not whether they have the same characters in them. Sort of.

Remember that there is a String pool, and creation of a String that already exists should reuse that reference.

The == operator is used to compare primitive values. You can't use the equals method on primitives because you can't use any method on primitives. When used with reference variables (objects, like Strings), the result of x == y is a Boolean value that is the result of the following test: do these two reference variables refer to the same object—that is, the same space in memory? Said a different way, are the bit patterns of x and y identical?

You use the equals() method to compare two objects to determine if their meaning is equivalent. With Strings, it has been decided that their meanings are equivalent if the characters they store are the same—that is, "J.B. Lenoir" is meaningfully equivalent to "J.B. Lenoir". It has been decided because the String class is final. If you have a way you think is an improvement over this method of comparing String meanings, you're out of luck in Javaland. However, you can go nuts and enjoy overriding the equals() method for your own objects. This is a good idea, by the way.

To determine if two Strings are not equal, you can use s1 != s2 or (! s1.equals(s2)).

Here is something that will come in handy. Remember that you will get a runtime exception if you try to call a method on a null reference:

```
String s1 = null;
if (s1.equals("")) ...//no!
```

This code will blow up because you can't call a method on a null reference: the runtime will throw a NullPointerException. The way to fix it is to get in the habit of checking the other way around: pass the possibly null value into the equals()

method as its parameter. Like this:

```
String s1 = null;

if ("".equals(s1)) ....//okay
```

That's a good way to test if a String is empty.

PREV < Day Day Up > NEXT

Useful String Methods

In keeping with the habit I'm trying to develop, of focusing on code that you can run and change and mess around with, and get you in the habit of doing instead of reading, let's just plunk down some code that shows off some of the useful methods of the java.lang.String class.

StringDemo.java

```
package net.javagarage.demo.String;

/** <p>

 * Demonstrates how to use Strings to your advantage.

 * </p>

 * @author HewittE

 * @since 1.2

 * @version 1.0

 * @see

 **/

public class StringDemo {


private static void print(String s){

System.out.println(s);

}

private static void print(char c){

System.out.println(c);

}

private static void print(int i){

System.out.println(i);

}

private static void print(boolean b){

System.out.println(b);

}


public static void main(String[] args) {

StringDemo sd = new StringDemo();


//make a String to work with
```

```
String s1 = "Java Rockstar";

//return the character at a given position
print(s1.charAt(2)); //prints "v"

String s2 = "zzzzz";
print(s1.concat(s2)); //prints "Java Rockstarzzzzz"
print(s2.concat(s1)); //prints zzzzzJava Rockstar

print(s1.endsWith("r")); //prints "true"
print(s2.endsWith("y")); //prints "false"
print(s1.equals("Java Rockstar")); //prints "true"
print(s1.equalsIgnoreCase("java rockstar"));
//prints "true"
print(s1.equals(s2)); //prints "false"
print("muscle car".equals("cool")); //prints "false"

//prints java.lang.String
System.out.println(s2.getClass().getName());

//does string match this regular expression?
//this regex checks if String is any digit
print("7".matches("[0-9]")); //prints "true"
print("whoa!".matches("[0-9]")); //prints "false"

print(s2.length()); //prints "5"

double d = 21D;
//this is a static method, so we use the class name
//(String) and then call the method on the class
System.out.println(String.valueOf(d));
//prints "21.0"
//you can do this with floats, ints, chars, etc.

char[] chars = {'m','a','m','b','o'};
String charString = String.valueOf(chars);
print(charString); //prints "mambo"

//removes trailing and leading whitespace
```

```
s1 = " s p a m ";

print(s1); //prints "s p a m"




//check the pool

print(s2.intern());




}

}
```

The weirdo here is definitely intern(). That method checks the String pool and sees if there is already a String containing a meaningfully equivalent String, and if so, returns it. If not, the String is added to the pool and a new reference is returned.

Here is the output of running the preceding code:

```
v

Java Rockstarzzzzz

zzzzzJava Rockstar

true

false

true

true

false

false

java.lang.String

true

false

5

21.0

mambo

s p a m

zzzzz
```

So to recap, the JVM maintains a pool of unique Strings, which is empty when the JVM starts. As you create Strings, they are added to the pool. All Strings, including constant expressions with a String value, are interned. According to the Java 5.0 documentation, "for any two strings s and t, s.intern() == t.intern() is true if and only if s.equals(t) is true." You don't typically need to call intern yourself.

PREV < Day Day Up > NEXT

Better Efficiency with StringBuffers

The StringBuffer class is useful for making strings that need to be concatenated, appended to, or otherwise tangled with more than two or three times. Because regular String objects are immutable, changing a String actually creates a whole new String, which is an expensive operation. To make String-changing more efficient, we use StringBuffer. If you've done any work in C#, you'll recognize this as System.Text.StringBuilder.

There are generally only two things you do with a StringBuffer: append text to it, and then, when you're all done appending, call the toString() method to use it as a String. Like so:

```
package net.javagarage.demo.String;


public class StringBufferDemo {


public static void main(String[] args) {
  //do it manually
    StringBuffer sb = new StringBuffer("Sideshow");
    sb.append(" Bob");
    sb.append(" Frames Krusty");
    System.out.println(sb);

    System.out.println(doLoop());
}


    private static String doLoop(){
        StringBuffer sb = new StringBuffer();
        int i = 10;
        while(i > 0){
            sb.append(i + ",");
            i—;
        }
     return sb.toString();
    }
}
```

For this sort of operation, StringBuffers provide better efficiency. You often need to do this sort of thing when building SQL query strings to send to a database or when building file paths. It might be important to you to know that StringBuffer does not override the equals() method.

Calls to StringBuffer append operations can be chained. That is, the return type of a call to append is the newly appended StringBuffer object. So we could rewrite the code above in the following manner:

```
StringBuffer sb = new StringBuffer("Sideshow");

sb.append(" Bob").append(" Frames Krusty");

//...
```

This has the same result as the more verbose code in the previous example.

Chapter 14. ARRAYS

DO OR DIE:

- Learn how arrays work in Java. If you think you have a better objective for this topic (after all, a topic on arrays, duh), I'd be happy to hear about it (good luck, buddy [as if])

The array is a standard feature of programming languages. An array is a collection of objects or primitive values, each of which is of the same type, and each of which is associated with an ordinal place in the collection. This ordinal place is called an *index*. As a feature of the Java language, they offer quick storing and retrieval of data, in large part because their size is always known. That is, after you create an array to store 150 items, that particular array can never be resized.

Creating Arrays

You can declare a new array variable, just like you would any other member variable (like this)

```
String[] args;
```

The square brackets mean "array." An array in Java is an object, which means that you initialize an array with the keyword new. Like this:

```
String[] args = new String[10];
```

This creates a new array object, called args, with 10 buckets for putting Strings into. You can fill an array with whatever kind of thing you want: Object, int, boolean, Address, char, another array, and so on.

The first element of a Java array has an index of 0. To reference the element in the second bucket of an array named args, type this:

```
args[1]
```

FRIDGE

Remember that arrays are the size that you make them, no bigger, and that the first element's index of an array is 0. That means that if your array holds 10 elements, you can happily access elements with index 0 through 9 all day long. Those are the 10 you get. Try accessing the element at index 10, and you'll get an ArrayIndexOutOfBoundsException, meaning the element you tried to access doesn't exist. Try accessing the element at index -56. Same thing. This can be confusing, so try not to be confused by it. Remember that the first element of an array is 0, so the last element of an array is *array length minus 1*!

That's because the array starts at 0, dammit!

There is a shorthand to declaring and initializing and populating a Java array all with one line of code. You have to already know what all your values are, but this frequently comes in handy.

```
String[] cats = {"Noodle", "Doodle", "Little Mister"};
```

Now you have an array with three buckets, containing one each of the stated values. We can get a value out of an array by referencing the array name, followed by square brackets containing the index of the element you want. Like this:

```
cats[2]; //returns "Little Mister"
```

The following code demonstrates different ways of creating regular and multidimensional arrays in Java.

Creating Arrays.java

```
package net.javagarage.demo.arrays;

/**<p>

 * Demonstrates how to create arrays

 * and refer to their elements.

 * </p>

 * @author eben hewitt

 **/

public class CreatingArrays {


public static void main(String[] args) {

//1. instantiate new array with 5 cells

int[] myInts = new int[5];


//put a value into the 2nd cell

myInts[1] = 1;
```

```
System.out.println("1st cell: " + myInts[0] +
". 2nd cell: " + myInts[1]);
/*
 * prints 1st cell: 0. 2nd cell: 1
 * notice that means that the first cell
 * (the 0th cell was initialized to the default
 * value for its type (int)
 **/

//2. now show another way to make them
//notice that spaces don't matter
//populate array with the original members of //Motorhead
String [] myStringArray = {"Lemmy","Larry","Lucas"};

//this is one of the few times in Java you
//can use ; after }

//3.
//you can also do this (create without initializing)
Object[] objects;

//note that you can declare it by putting the [] in a
//different location. the following is LEGAL:
Object myThings[];

/* but no one does it that way, and i'd avoid it.
 * sometimes convention is just easier for everyone.
 * it is more natural to say "Object array called
 * myThings" as you read—it makes sure the complete
 * type is in the declaration, not the identifier.
 * After all, your value on the heap is an Array
 * object called 'myThings' that holds Objects,
 * not a "myThings array".
 */

 //try to access a value:
 //System.out.println(myThings[0]);
```

```
//COMPILER ERROR! Not initialized!


//create a 2-dimensional array with 10 elements

//in each dimension:

int[][] square = new int[10][10];


//create a "ragged" array, with each dimension

//holding varying elements

int[][][][][] ragged = new int[2][4][3][10][5];




}

}
```

One thing to note about creating arrays (and, perhaps more to the point, the loose language that is sometimes used, even by Java book authors, as in the preceding code) is that there is no such a thing as a real multidimensional array in Java. Every Java array has only one dimension. You can create an array that holds another array, however, thereby achieving two-dimensional array representational capability.

About array values: You can insert into an array any value that can be automatically promoted to the declared type. For example, say you have an array of ints. You can put a byte or a short in there, because an int is 32 bits, and those are smaller than 32 bits, and so can snugly fit inside an int. No problem. What you can't do is add a long to that array because a long is 64 bits, which is too big to squeeze in there. You can't put an object in there either.

Using Arrays

Even if your array is filled with primitive ints, every array in Java is an object, and is treated as such. Arrays are objects. So there are a number of methods that you can put to work to get value out of them. It also means that you should comply with object assignment and equivalency tests, as well as state and local laws.

You can think of two-dimensional arrays as holding a row and column number, in that order. For example, take Table 14-1.

A Two Dimensional Array

	0	**1**	**2**
0	**78**	—	46
1	**92**	78	92
2	**98**	81	66
3	**71**	56	89
4	**55**	43	45

This table represents a two-dimensional array that holds 5 rows and 3 columns. You would create such an array like this:

```
int[][] grades = new int[5][3];
```

The element at position [3][2] is 89. The element at position [0][0] is 78. The element at position [4][2] is 45. The element at [0][1] is empty. The element at position [4][3] doesn't exist, and trying to access it gets you an ArrayIndexOutOfBoundsException.

```
grades[0][1] = 90 ;

// puts 90 into row 0, column 1.

grades[2][ 2]++ ;

//increments value at row 2, column 2 by 1
```

Copying Array Elements

If you want to do this, you can simply assign a new array to the old array, and then you have your data in the new array, right? Well, yes, but this is shared data. If somebody updates that data, both arrays get the update, and this might not be desirable. So the API designers were thoughtful enough to allow us to do this via the aforementioned System.arraycopy() method.

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

//...

System.arraycopy(sourceArray, 0, destArray, 0,

        min(sourceArray.length, destArray.length));
```

Remember that length is a field of the Array class; it isn't length() or getLength(). The preceding code uses the static import feature new to Java 5.0, so it won't compile on earlier versions. But that's easily fixed.

The following class shows how to use two-dimensional arrays and how to use the System.arraycopy() method.

UsingArrays.java

```
package net.javagarage.arrays;


/**<p>

 * Shows how to use arrays after they are created.

 * </p>

 * @author eben hewitt

 * @see

 **/

   //import all static methods of the Math class as

   //convenience

import static java.lang.Math . *;


public class UsingArrays {


public static void main(String[] args) {

//create an instance (object) of this class

UsingArrays demo = new UsingArrays();


//now call the methods defined in this class

//on the object reference

demo.useSquareArray();

demo.copy();
```

```
        }

/**
 *Creates an array of 100 elements (10 * 10 = 100),
 *and populates them with values.
 */
private void useSquareArray(){
int[][] square = new int[10][10];

//loop over to populate.

for (int i = 0; i < 10; i++){
for (int j = 0; j < 10; j++){
square[i][j] = j+1;
System.out.print(square[i][j]);
}
System.out.println();
}
}



private void copy(){
char[] sourceArray = {'s','e','c','r','e','t'};
char[] destArray = new char[6];
//the arraycopy method copies the contents and not
//merely the references
System.arraycopy(sourceArray, 0, destArray, 0,
    min(sourceArray.length, destArray.length));

System.out.println(destArray[2]);
}



}
```

The result is as follows:

```
12345678910
12345678910
12345678910
12345678910
12345678910
12345678910
12345678910
12345678910
12345678910
12345678910
c
```

Shifting Array Elements

You can shift all of the elements in an array over to the right or the left with a little simple code. This uses the arraycopy() method, which we just saw.

```
//say you've got an array called array
//shift all elements right by one
System.arraycopy(array, 0, array, 1, array.length-1);

//shift all elements left by one
System.arraycopy(array, 1, array, 0, array.length-1);
```

The arraycopy() method (which should have been called arrayCopy to follow Sun's own naming conventions) takes the following parameters:

src	The source array
srcPos	Starting position in the source array
destination	The destination array
destStartPos	Starting position in the destination data
length	The number of array elements to be copied

Note that you have to know what you're doing when you use this method. It throws NullPointerException, IndexOutOfBounds-Exception, or ArrayStoreException, if the source or destination arrays are null, or if the system can't squeeze your big, honkin' data into that little array. Whew.

Using Stacks

Stacks are like Pez dispensers. Stacks are very similar to arrays, and so we'll discuss them here. Many internal data structures are stored as stacks within Java. The stack operates on a last in, first out basis. You can create your own stack, or you can use an instance of java.util.Stack.

The java.util.Stack class extends java.util.Vector, which is a type of collection. The two main operations you use with a stack are pop() and push(). You can also peek() to see the item on the top of the stack. Finally, you can search the stack for an item, and you can determine if the stack is empty.

UsingStacks.java

```
package net.javagarage.arrays;

import java.util.Stack;

/**
 * <p>Does stuff with java.util.Stack
 * @author eben hewitt
 */
public class UsingStacks {

public static void main(String[] args) {
Stack<Integer> stack = new Stack<Integer>();

stack.push(41);
stack.push(42);
stack.push(43);

//get first in line
System.out.println("First: " + stack.peek());
//prints 43

//is stack empty?
System.out.println("is empty? " + stack.empty()); //false
System.out.println("search: " + stack.search(42));

//search returns 2—this is 1-based!!!



}

}
```

Note that you need to compile this code with the 5.0 compiler because it takes advantage of the new generics autoboxing feature (we're adding int primitives to the stack, which is declared to accept only Integer objects—they get automatically wrapped).

ArrayLists and toArray

Say that you have an ArrayList (a java.util.ArrayList is a type of List that works like an array, but is automatically resizable). ArrayLists hold java.lang.Objects.

```
ArrayList myObjArrayList = new ArrayList();

//put stuff into myObjArrayList without generics,

//you can retrieve them only as type Object


//initialize a new array that has as many elements

//as the ArrayList has:

MyObject myObjArray[] = new MyObject[myObjArrayList.size()];


//instantly put all of the elements from

//the ArrayList into the array,

//and they are of the proper type:

myObjArray = myObjArrayList.toArray(myObjArray);
```

You now have a regular array that holds objects of type MyObject. You get this without having to loop over each item in the ArrayList and call its get method and perform a cast.

Neat.

Chapter 15. DOCUMENTING YOUR CODE WITH JAVADOC

Here is some stuff about documenting your code. I know this is something that you do regularly, and always make sure to keep up-to-date. Not.

The Java SDK comes with a tool that is easy to use (really) called Javadoc. It does an amazing amount of work for you by taking the API documentation for your applications off of your shoulders. This out-of-the-box functionality lets you write special comments directly in your code, and then it generates formatted HTML files to document your application's API.

Writing Javadoc Comments

Javadoc comments are an advanced form of comments. The purpose of regular source code comments is to indicate to readers of your source code files what you're up to. The limitation here is obvious.

What if I don't wanna distribute my source code files? Even if I did, it's inconvenient and difficult to sort through all of the source code just to find out what a class' contract is.

Then you have to distribute documentation files separately. Duh.

But it's a draaaag to make separate documentation files. They get out of synch with my source code, and then they're worse than useless.

True. You're starting to sound a little whiney, dear, but these are good points. Javadoc addresses your concerns by providing a facility for generating complete code documentation externally, in HTML. It's a terrific and easy documentation tool, because HTML is platform independent and anybody can view it with pleasure.

How to Do It

There are two requirements for generating documentation. First, you need to add the appropriate comments to your file. Second, you need to run the Javadoc command-line tool that creates the files.

The Javadoc tool generates documentation for the following items:

- Packages
- Public classes and interfaces
- Public and protected methods
- Public and protected fields

No private items are included unless you explicitly specify them.

Just write your special comments immediately above the items in your files that they comment (that is, right above your class, right above each field, and right above each method). All of your Javadoc comments must be outside a method body.

Start Javadoc comments with /** and end them with */. Since Java 1.4, the two asterisks are not required, but are convention.

Write a short sentence summarizing a method, or a short paragraph or two to summarize a class.

Within your summary text, you can use the inline tag {@link URL} to have Javadoc create a hyperlink for you. You can also use any HTML tags you want, though it is typically kept to basic items such as <P> and <CODE> to format your comments.

Using Javadoc Tags

You can use the following tags in your Javadoc comments. These get picked up by the tool to serve as the text for your documentation.

- @author Who wrote this class. Use this only with classes and interfaces, not methods. Use separate @author tags for each author if there's more than one.
- @version Identifies the version of the class. Use only with classes and interfaces. Use the following values:

```
/**
 * @author eben.hewitt
 * @version 1.0
 */
```

- @param Indicates a parameter to a method or a type parameter to a class. Use a separate @param tag for each parameter. For use in methods and constructors only. Don't include the param's type, just its name and description, as in the following:

```
/** for method:
 * @param salary The boss's current salary.
 * @param schmoozability How much this boss can
 * schmooze, represented on a scale of 900-1000
 */

/** for class type parameter:
 * @param <T> Some class parameter.
 */
```

- @docRoot Path to the root directory of the documentation, as in the following:

```
* This class is a member of the
* <a href="{@docRoot}/../guide/collections/index.html">
* Java Collections Framework</a>.
```

- @return Identifies a method's return value. Obviously, this is for use only with methods.

```
/**
 *@ return The boss's bonus
 */
```

- @throws Indicates exceptions thrown by this method. Should be used to describe under what circumstances this exception might be thrown. @exception is also acceptable in place of this, but I prefer the active verb.
- @see Points to other relevant classes with a hyperlink, as in the following example:

```
/**
 * @see packageName.ClassName#member text
 */
```

- @since Indicates the release of your software that first introduced this feature. Can be used with classes, methods, and fields.

```
/**
 * @since 1.2
 */
```

- @deprecated Indicates that a method or class is deprecated and shouldn't be used. Write it like this:

```
/**
 * @deprecated As of JDK 1.1, replaced by
   {@link #setBounds(int,int,int,int)}
 */
```

- {@value arg } Accepts the name of a program element and label. This way, it can be used in any doc comment, not just constant fields.

```
/**
 * @value package.class#member label
 */
```

- {@literal tag } Denotes literal text so that the Javadoc tool does not attempt to interpret enclosed text as containing HTML or Javadoc tag.
- {@link link} You use the @link tag inline in paragraph text to have the Javadoc tool automatically generate a hyperlink to the Java class or method you specify, as in the following example:

```
import ...
/**
 * Uses NIO to read in the datafile initially using a
 * @link java.nio.ByteBuffer} and a {@link
 * java.nio.CharBuffer}, and then stores the
 * complete file in a synchronized list.
**/
public class SomeClass { ... }
```

The above will generate a hyperlink to the Javadoc for the class indicated. Notice that you can add a # following a class name to indicate a method you want to link to directly. Doing so means that you have to supply the types of each parameter to the method, but not the parameter names, as in the following example of Javadoc for a public method:

```
/**
 * This client-facing method should be
 * preferred for clients over the {@link
 * mypackage.db.DataClass#updateRecord(long,
 * java.lang.String[], long)} method.
 */
```

Using the Javadoc Tool

Now that you are ready to generate your documentation, whip out a console and get ready to rumble.

Here's the usage for the Javadoc tool:

```
javadoc [options] [packagenames] [sourcefiles] [@files]
```

To see a full list of usage options, simply get a console and type javadoc –help and hit Enter. Here's a basic way to generate your docs:

1. At the console, navigate to the directory that contains the package you want to document.
2. Type javadoc -d DocOutputDirectory PackageName

To document the net.javagarage package and put it into the /eben/home/MyJavaDocs directory, I type this:

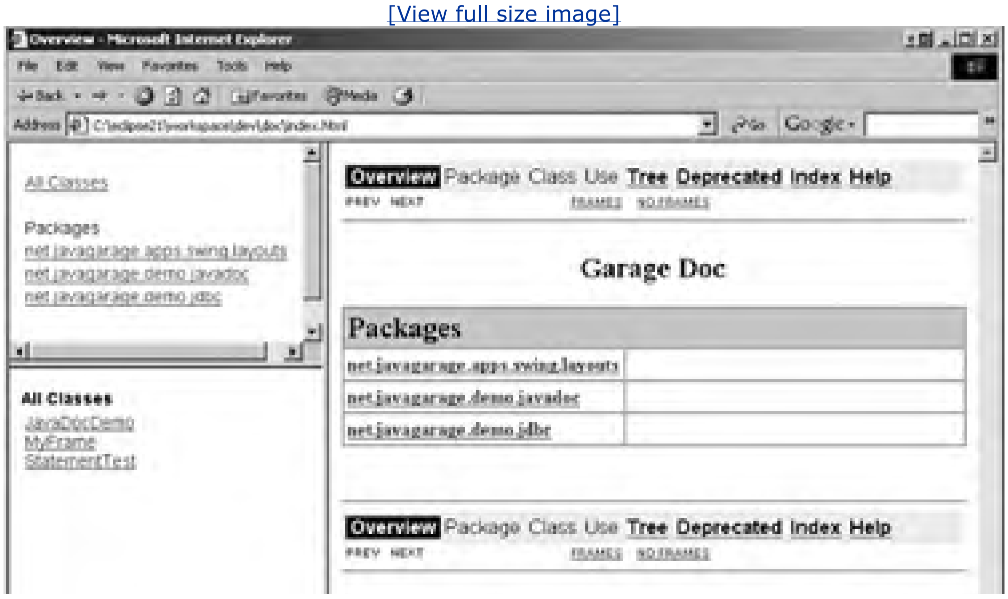
```
>javadoc -d /eben/home/MyJavaDocs net.javagarage
```

The utility runs outputting what it is doing along the way. It creates several directories and many files, and when it's done, you have a folder with all of your documentation. Click the index.html file to view it.

Note that you have several options. You can include all of the classes and packages in your application, or just select a few (see Figure 15.1).

Figure 15.1. The index page of the generated documentation.

[View full size image]


Overview - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Favorites Media
Address C:\eclipse21\workspace\dev\doc\index.html
Go Google
Overview Package Class Use Tree Deprecated Index Help
PREV NEXT FRAMES NO FRAMES

All Classes

Packages
net.javagarage.apps.swing.layouts
net.javagarage.demo.javadoc
net.javagarage.demo.jdbc

All Classes
JavaDocDemo
MyFrame
StatementTest

Garage Doc

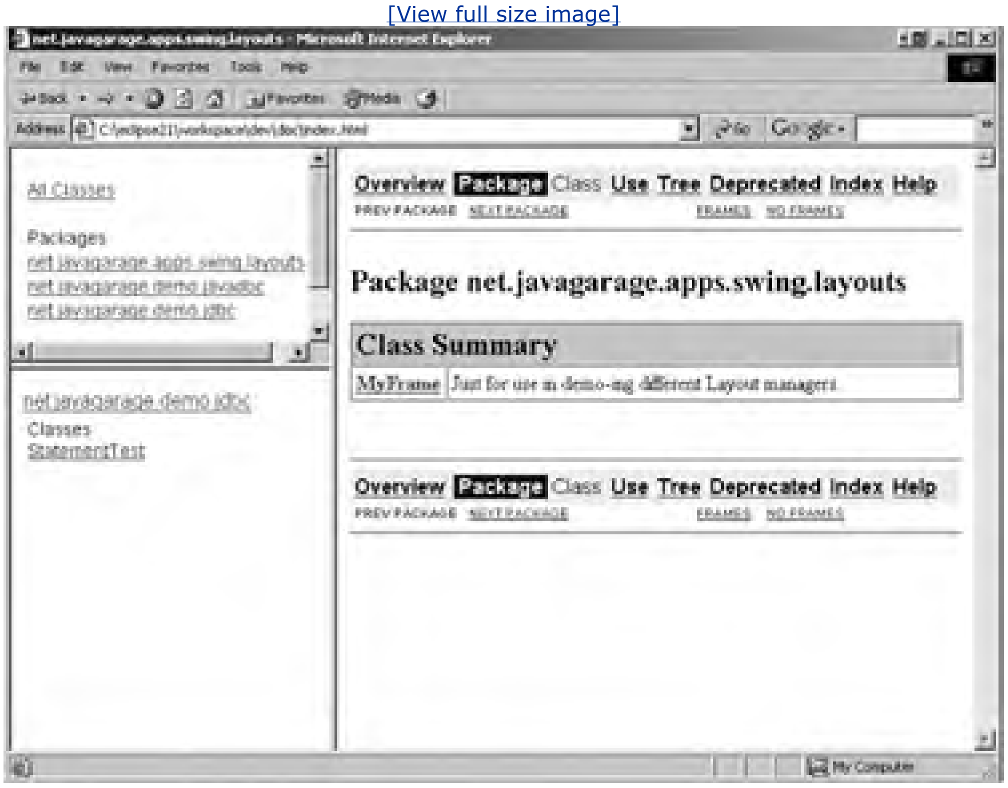
Packages	
net.javagarage.apps.swing.layouts	
net.javagarage.demo.javadoc	
net.javagarage.demo.jdbc	

As you can see, I have only included a few classes from the entire garage kit-and-kaboodle, for simplicity's sake. Notice that it looks exactly like the Java API documentation used online by Sun.

Let's click on the package name to get the class summary for the net.javagarage.apps.swing.layout package. It gives you a list of classes with their descriptions in the main frame (see Figure 15.2).

Figure 15.2. The package's class listing and description.

[View full size image]


net.javagarage.apps.swing.layouts - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Favorites Media
Address C:\eclipse21\workspace\dev\doc\index.html
Go Google
Overview Package Class Use Tree Deprecated Index Help
PREV PACKAGE NEXT PACKAGE FRAMES NO FRAMES
My Computer

All Classes

Packages
net.javagarage.apps.swing.layouts
net.javagarage.demo.javadoc
net.javagarage.demo.jdbc

net.javagarage.demo.jdbc
Classes
StatementTest

Package net.javagarage.apps.swing.layouts

Class Summary	
MyFrame	Just for use in demo-ing different Layout managers

Changing Javadoc Styles

There is a default stylesheet that Sun's Javadoc tool creates when you use it to generate Javadoc. It looks like this:

```
/* Javadoc style sheet */

/* Define colors, fonts and other style attributes

   here to override the defaults */

/* Page background color */

body { background-color: #FFFFFF }

/* Headings */

h1 { font-size: 145% }

/* Table colors */

.TableHeadingColor     { background: #CCCCFF } /*

 Dark mauve

*/

.TableSubHeadingColor  { background: #EEEEFF } /*

 Light mauve

*/

.TableRowColor     { background: #FFFFFF } /* White */


/* Font used in left-hand frame lists */

.FrameTitleFont     { font-size: 100%; font-family:

 Helvetica,

Arial, sans-serif   }

.FrameHeadingFont   { font-size: 90%; font-family:

 Helvetica,

Arial, sans-serif   }

.FrameItemFont      { font-size: 90%; font-family:

 Helvetica,

Arial, sans-serif   }


/* Navigation bar fonts and colors */

.NavBarCell1      { background-color:#EEEEFF;} /*

 Light mauve */

.NavBarCell1Rev   { background-color:#00008B;} /*
```

```
 Dark Blue */
.NavBarFont1      { font-family: Arial, Helvetica,
 sans-serif;
color:#000000;}
.NavBarFont1Rev   { font-family: Arial, Helvetica,
 sans-serif;
color:#FFFFFF;}

.NavBarCell2    { font-family: Arial, Helvetica,
 sans-serif;
background-color:#FFFFFF;}
.NavBarCell3    { font-family: Arial, Helvetica,
 sans-serif;
background-color:#FFFFFF;}
```

I include it here so that you can see the elements that you would need to change if you want to specify your own values.

To change the style your docs use, just use the Javadoc tool in the normal way, but also add the – stylesheet flag.

```
>javadoc -d /my/docs -stylesheet /home/styles/docs.css
     net.javagarage
```

Generating Javadoc in Eclipse

If you are using an IDE such as Eclipse to do your business, you can use a quick weezard to do your dirty work for you.

1. In Eclipse, click Project > Generate Javadoc…
2. Select the packages you want to include.
3. Specify the visibility level you want to include.
4. In the Destination text field, specify the location where you want the docs to go. Click Next.
5. Tweak the documentation options if you want—usually it is best to leave them all selected.
6. Specify a different stylesheet if you like, using the CSS classes as shown previously. Click Finish.

Generating Javadoc with Ant

You can also generate your Javadoc using the Ant utility. Ant is available as a free download from http://ant.apache.org. The current version, as of this publication, is 1.6.1. Ant is used like the make tool to automate the software build process. It offers an XML syntax featuring tags that, among other things, wrap common Java tools, including the javadoc command.

While in this section it is not appropriate to go into the Ant tool in detail, my assumption here is that this will serve as a useful—if quick—reference later when you want to generate Javadoc using Ant.

If you are using Eclipse (3.0 is the current version), you already have Ant installed with it. All you need to do is create a file (conventionally called build.xml), and add it to your project. You can then add Ant tags to that file, and then execute the script using the built-in Ant plugin.

Writing the Build Script

Here is a sample script that you can use in your projects to easily generate your Javadoc.

```
build.xml

<?xml version="1.0" encoding="UTF-8"?>
<project name="My Project Build" default="init">
<property name="src.dir" value="code/source"/>
<property name="docs.dir" value="./docs"/>
    <!—Create documentation. —>
    <target name="Generate JavaDoc" depends="">
        <javadoc packagenames="myrootpackage.*"
             sourcepath="${src.dir}"
             destdir="${docs.dir}/javadoc"
              access="private"
              windowtitle="My Window Title"
              verbose="true"
              author="true"
              version="true">

             <bottom>
                 <![CDATA[<b>Java Garage, Eben
                    Hewitt, 2004<b>]]>
              </bottom>
          </javadoc>
     </target>
</project>
```

```
</project>
```

This script will pick up all of the Javadoc tags you have written down to the "private" level. If you specify access="public", then only Javadoc comments on public methods will be written out to the Javadoc.

The text between the "bottom" tags is optional and represents some arbitrary text that you can place at the bottom of every page. It is appropriate for copyright.

When this script is executed, it creates all of the Javadoc for you, and places the files in the docs/javadoc directory—the value of the destdir attribute.

For extensive, helpful documentation on using the Ant tool to automate your builds, check out http://ant.apache.org/manual/.

Executing the Ant Script in Eclipse

While you don't need Eclipse to execute Ant scripts, Ant is often used as a plugin to an IDE, and comes built-in with Eclipse, and is the simplest way to demonstrate without going into detail here. Note that you need to have a directory structure that matches the one you specify in your build.xml file for the Javadoc destination.

Once you have the script modified to your liking, here is how to run it:

1. In the Eclipse IDE, choose Window from the main menu, and click Show View > Ant.
2. Click the three yellow plus sings to add the build.xml file to your project.
3. Once you have done so, click the green forward arrow in the Ant window to execute the script. The console window will open and show you the verbose output that Ant generates as it executes the script.

Ant is explored in more detail in the *More Java Garage* book.

That's it! Happy Javadoc-ing.

Chapter 16. ABSTRACT CLASSES

DO OR DIE:

- Clean my room.
- Take out the trash.
- Double RAM in server.
- Take over the world by ruling a race of mindless robots that I control through my Tivo.

An abstract class is mostly potential. A regular class must implement all of the methods it declares. An abstract class can implement some of the methods it declares, and leave some for a subclass to implement.

The abstract class can be a useful thing. In this topic, we'll take a look at what they are, how to write them, how they behave, and when to use them.

Dude, Where's My Implementation?

An abstract class is a class whose declaration includes the keyword abstract. Based on that definition, you should be able to write the simplest possible abstract class. Okay…geez…*I'll* do it. How about this:

```
public abstract class SimplestAbstractClass { }
```

That meets the definition, and so it compiles.

You might never write an empty class such as this. But there are occasions when you will come across such empty classes; such usage aims to provide a formal organization to an application for later possibilities for extensibility.

The following is also an abstract class:

```
public abstract SecondAbstractClass {

   public abstract void someMethod();

}
```

The preceding code declares an abstract class that includes an abstract method. The class should be subclassed, and when it is, an implementation for someMethod() must be provided. Notice that abstract methods don't have any curly braces (which is where the implementation would go). They just end in a semi-colon. If you end a method declaration in a semi-colon, you must also declare it abstract.

The following is also an abstract class:

```
public abstract class ThirdAbstractClass {

    int counter = 0;

    void printMessage(String msg) {

         System.out.println("Your message: " + msg);

    }

}
```

Notice how we have what looks to be a totally regular class. It has a member variable that is initialized to a value, and it has an implemented method. But it's declared abstract, and it compiles. What gives?

Abstract classes can consist of implementations, such as in the preceding ThirdAbstractClass, or they can *defer* their implementation entirely, or they can have a combination of stuff they implement and stuff they defer to let someone else implement.

What Is an Abstract Class?

You cannot instantiate an abstract class. That is, you cannot make an object of it. You can't write new SecondAbstractClass(). Ever. This is sensible, as it is the way the world works. The world has things in it, like balls and books and rocks. The world also has abstract concepts that help us organize actual things, such as rocks and people, and to organize other concepts. The economy is a good example. There is no such thing as "the economy." Anywhere. It doesn't exist. We talk about it enough. "Blah blah blah…how's the economy…. Ah it's horrible…. Ah well what're you gunna do…." Because "the economy" is a useful placeholder for a collection of other things—some of which are physical objects, and some of which are other concepts. You can't touch the economy, or hang it out the window, or kick it. It doesn't exist. You can't say new Economy(). You can only allocate 20 pounds British Sterling to Jill, and transfer 1,000 shares of Sun Microsystems stock to Phil, and collect $387.65 interest on your specific mutual fund run by a specific company containing specific stocks of specific companies that exist in the world. Those are the actual things of the economy. And the interest rate and whether the NASDAQ is generally bullish or bearish, these are abstract. So it is well and just that abstract classes should support both a little implementation (thing-ness) and a little idea (abstract-ness). Or neither. Or one. Or both.

That's why SimplestAbstractClass, SecondAbstractClass and ThirdAbstractClass above all make sense as possible representations of an abstract class within the Java programming language.

Some Guidelines About Abstract Classes

- When you want to declare a method to indicate something that should be done, but you want to defer the implementation to a subclass, you use an abstract class. That's what they're for. Any method that you want to let a subclass implement you declare using the abstract modifier. We can call this an abstract method.

- You can mix implementing methods and abstract methods in an abstract class. However, a class must be declared abstract itself if it contains *even one* abstract method.

- You cannot instantiate an abstract class. That is, you cannot make an object of it. You can't write new SecondAbstractClass(). Ever.

- A class cannot be both abstract and final at the same time. Think about it: abstract means that you must extend the class and implement it somewhere, and final means you are totally not allowed to ever extend this class. They are mutually exclusive. Note that when I write, "you must extend it…," I don't really mean it. That is, you can create an abstract class in your app and never have any other class extend it. It just wouldn't do anything then.

Using Exceptions with Abstract Classes

A common question regarding abstract classes is how to use exceptions with their method declarations. What exceptions can an inheriting method throw with respect to the inherited method? Consider the following class, whose single method throws one exception: IOException.

```
public abstract class AbstractPrinter {

abstract void printMessage(String msg) throws IOException;
}
```

The implementation of this abstract class is shown in PrinterImp1.java. To implement an abstract class, you simply extend it, just as you would extend any other class, using the extends keyword.

PrinterImpl.java

```
package net.javagarage.demo.abstractclasses;

import java.io.IOException;

public class PrinterImpl extends AbstractPrinter {
   void printMessage(String m) throws IOException {
      System.out.println("Your message:
          " + m);
   }
}
```

FRIDGE

This is different than in C#, in which classes are extended using the : operator. That is the same way that interfaces are implemented in that language. To implement an interface in Java, you use the keyword extends.

The preceding shows that the rules are the same as when you extend a regular class.

- You cannot declare that your implementation method throws an exception if the superclass method does not.
- If your abstract method declares that it throws an exception, your implementation method must throw that same exception, or any subclass of that exception.

PREV < Day Day Up > NEXT

Using an Abstract Class Polymorphically

A chief benefit of object-oriented programming is polymorphism. Abstract classes are one way that Java supports polymorphism. That is an issue best left for another topic all to itself. But here, let's look at what is perhaps the most common reason people make abstract classes in the first place: to get flexibility in their design.

Say you have a little Dungeons and Dragons type game where some hero fights monsters. In this adventure, it is possible for the Hero (we'll make him a class) can encounter Serpents or Goblins. Looking at our requirements, we might determine that a Serpent and a Goblin have certain traits in common. They are both monsters. Let's put a capital M on that and make it a class: Monster. Good. Now the Monster class will be the abstract superclass of both Serpent and Goblin, and they will be concrete subclasses. We don't allow any instances of Monster itself—you have to make some particular *kind* of Monster.

This is a pretty good design. Because now when we need to make a monster do something in the game, we don't have to clutter up our design worrying about making the Goblin move and fight and then practically doing the whole thing over again making the Serpent move and fight. They will both move and fight, but they will move and fight in different ways. The Goblin will just walk to move. The Snake will slither, which is a more specific kind of movement, different from how it will most frequently be implemented. The Goblin will fight by swinging a sword, and the Snake will fight by biting. Again, they both need to do it, but each does it in its own way.

Why not just make a Goblin class and a Snake class and forget about Monster? Who cares about Monster if the Hero will never actually encounter anything that is a Monster, but only ever subclasses of Monster? Here's why: There are certain things that the Hero does when he or she encounters a monster, regardless of what type of monster it is. These things probably include "fight." Consider what that means in terms of your design.

It might seem okay right now to make a method signature that looks like this: `heroInstance. fight(Goblin goblin);`. But you have to replicate most if not all of that functionality making another method called `heroInstance .fight(Serpent serpent);`. Now you're in Copy and Paste land. And that's a dark forest indeed. See where the Monster comes in handy?

Here's a rule: If you find yourself highlighting a method implementation and typing `Ctrl+C`, you probably need an abstract wedgie in there somewhere. Just don't do it—don't copy that code. Just release the keys and step away from the laptop. You don't have to live like that. You can be abstract.

Adventure.java

```
package net.javagarage.demo.abstractclasses.dungeon;


/**

 * <p>

 * @author eben hewitt

 */

public class Adventure {

    String name;


    //constructor that accepts a name for this adventure

    public Adventure(String nameIn){

        this.name = nameIn;

    }


    //program starts here
```

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

    //make a hero to go on adventure
    Hero harry = new Hero();
    //make a monster:
    //dig the polymorphism—
    //the reference type is the abstract superclass!
    Monster gizmo = new Goblin();
    //make a new adventure
    Adventure chamberOfHorrors
         = new Adventure("Chamber of Horrors");
    //start them off
    chamberOfHorrors.go(harry, gizmo);
}

//do the adventure
private void go(Hero hero, Monster mst){
    System.out.println("Welcome to The " + this.name);
    //blah blah blah
    System.out.println("You hear a
               horrible sound...");

    //the important part: call the private method
    //that represents the encounter
    doCombat(hero, mst);

        //all finished
    System.out.println("The " +
        this.name + " is done.");
}

//we can pass in any subclass of Monster,
//and it will act correctly.
private void doCombat(Hero hero,
                      Monster monster) {

     String monsterType =
            monster.getClass().getName();
        //dig the polymorphism...
        System.out.println("Hero sees a " + monsterType + "!");

        System.out.println("Will it approach you?");

        //dig the polymorphism again
        monster.move(12);

        //whoa! again!
        monster.fight();

        hero.fight();

        //okay, we get it now.
        if (hero.strength > monster.strength)
        {
        System.out.println("You defeated the
              " + monsterType + "!");
        } else {
        System.out.println("The " + monsterType + "
              defeated you!");
        }
    }
}
```

Hero.java

```
package net.javagarage.demo.abstractclasses.dungeon;
/**
 * <p>Some nerdy do-gooder who fights with Monster
 * subclasses in an Adventure
 *
 * @author eben hewitt
 * @see Adventure
 * @see Monster
 */
public class Hero {

    //used to determine who wins fights.
    int strength;

    //Constructor to make Heroes with.

    public Hero (){
        //randomly determine strength
        //so every hero object is different
        this.strength = (int)(Math.random() * 100);
    }

    //we'll pass a Monster to fight with into
    //this method just to show how polymorphism works
    //and keep this as simple as we can.
    public void fight() {
        System.out.println("The Hero attacks
        with a sword with
        strength " +
        this.strength + "!");
    }
}
```

Monster.java

```
package net.javagarage.demo.abstractclasses.dungeon;
/**
 * <p>Abstract class, represents an enemy of Hero.
 * @author eben hewitt
 * @see Hero
 */
public abstract class Monster {
    //we'll use this as a simple way to
    //determine winner of fights
    int strength = (int)(Math.random() * 100);

    //this will be different for most monster types
    String attack;



    //most monsters will walk, so we will provide
    //a default implementation for them to do that
    //so every walking monster in the world doesn't
    //need to implement this, since it would be
    //roughly the same.call getClass().getName()
    //to prove the runtime type
    public void move(int numSpaces){
        System.out.println(getClass()
                getName() + " walking " +
                numSpaces + " spaces.");
    }

    //abstract method. we don't know enough about how
    //each monster will do this, and it will probably
    //be at least a little different for each kind
    abstract public void fight();
}
```

Goblin.java

```
package net.javagarage.demo.abstractclasses.dungeon;
/**
 * <p>A subclass of Monster that is a terrible beast.
 * @author eben hewitt
 * @see Monster
 * @see Serpent
 * @see Adventure
 */
```

```
  /
public class Goblin extends Monster {
    //constructor. We don't have a member
    //var called 'attack' but remember
    //that our superclass Monster does

    public Goblin() {
        this.attack = "club";
}

    //implement fight in some Goblin-specific way
    //for this example, that just means hard-coding
    //the String 'Goblin' in there

    public void fight() {
        System.out.println("The Goblin attacks
              with a " + attack
              + " with strength " + strength + "!");
    }
}
```

Serpent.java

```
package net.javagarage.demo.abstractclasses.dungeon;
/**
 * <p>
 * @author eben hewitt
 */
public class Serpent extends Monster {
    public Serpent(){
        this.attack = "bite";
    }

    public void fight() {
        System.out.println("The Serpent
     attacks with a " + attack
          + " with strength " + strength + "!");
    }

    //remember that Goblins just walk—they can
    //just inherit that functionality from Monster.
    //Serpents move a different way:

    public void move(int spaces){
        System.out.println("Slithering " +
              spaces + " spaces...");
    }
}
```

Here is the output when we make a new adventure and make the Hero fight a Serpent.

```
Welcome to The Chamber of Horrors
You hear a horrible sound...
Hero sees a net.javagarage.demo.abstractclasses. dungeon.Serpent!
Will it approach you?
Slithering 12 spaces...
The Serpent attacks with a bite with strength 28!
The Hero attacks with a sword with strength 61!
You defeated the net.javagarage.demo.abstractclasses.dungeon.Serpent!
The Chamber of Horrors is done.
```

The Hero wins! Here is the output when we make a Goblin monster.

```
Welcome to The Chamber of Horrors
You hear a horrible sound...
Hero sees a net.javagarage.demo.abstractclasses.dungeon.
```

```
Goblin!

Will it approach you?

net.javagarage.demo.abstractclasses.dungeon.Goblin walking 12 spaces.

The Goblin attacks with a club with strength 14!

The Hero attacks with a sword with strength 5!

The net.javagarage.demo.abstractclasses.dungeon.Goblin defeated you!

The Chamber of Horrors is done.
```

Oops. The Hero lost that time.

The important thing is to notice that we can make any kind of Monster subclass—Skeleton, Gelatinous Cube, Wraith, Necromancer, whatever we want—and it won't break any code. We can use Monsters of types we haven't thought of yet, or that someone else thinks of later, and our Adventure class methods can stay the same. Very cool.

Look for places in your application where you can make use of this kind of design. It is extra work up front, but you will see productivity and ease of maintenance benefits later. As my friend from Connecticut always says, "You're going to pay now, or you're going to pay later—and it's usually cheaper to pay now."

PREV < Day Day Up > NEXT

< Day Day Up >

Chapter 17. INTERFACES

EL OBJECTIVOS:

- Learn It
- Live It
- Love It

The interface is Java's answer to multiple inheritance. It is a Java type that defines *what* should be done, but *not how* to do it. Interfaces are perhaps most useful when designing the API for your program. In this topic, we'll find out how to define an interface, how to implement one, and how to use it in program design.

< Day Day Up >

Let's Get One Thing Straight

Let's get one thing straight. Right off the bat. There are really two ways that people in Javaland use the term interface. One is conceptual and one is concrete.

People sometimes talk about a program's interface, or even a class' interface. Remember that API is an acronym for Application Programming Interface. When used this way, it means *the thing with which we interact*. It's what is exposed to us that we can work with. It is the visible boundary of a class or a program or a programming language's libraries. This is the conceptual version of the term interface. It means the public (I really mean non-private) methods that we can call to do something.

On the other hand, an interface is a Java programming language construct, similar to an abstract class, that allows you to specify zero or more method signatures without providing the implementation of those methods. Remember the implementation is the code block that does the work. Let's look at the difference.

```
public void printMessage(int numberOfTimes);

// a method declaration in an interface.
```

The preceding code is an example of an interface method declaration. Notice how the signature ends in a semi-colon. There is no code that does the actual work of the printMessage() method.

An interface will consist of method signatures that look like this; there can be no implementation at all. You do the implementation in a class. An implementation of the printMessage() method might look like this.

```
public void printMessage(int numberOfTimes) {

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

    System.out.println("Current message number " + i);

  }

}
```

Or the implementation could be different. Maybe a different implementation uses a while loop, and has a different String message, or doesn't print the current iteration of the loop as represented by i.

Imagine an interface with some method signatures, and a class that will implement the interface.

> Interface: I have 5 method signatures.
>
> Class: I want to implement them.
>
> Interface. Okay. But then you have to implement all of them. You are not allowed to say that you implement me without implementing every single one of my methods.
>
> Class: It's a deal.

An interface is a contract. It is binding between the interface and the class that implements the interface.

But why in the world would you want to do that? A class can implement whatever methods it wants. Why not cut out the middleman, go ahead and implement the methods you want, and then forget about the interface altogether? Well, you could do that. But there are actually several reasons why interfaces are very cool.

If it helps, you can think of an electrical outlet.

An electrical outlet is a wonderful invention. It is really, really a cool thing, the electrical outlet. The interface for every electrical outlet is *exactly the same* (okay, they're different in the United States than other places, and sometimes you get three little holes to plug stuff into, and sometimes only two; work with me here. Geez). You know that you will be able to use the electricity you need for your laptop, PlayStation, hair drier, or electric dog polisher as long as each of them have standard cords that will plug into an outlet. We could imagine a hypothetical interface called Pluggable that means that it has a standard set of prongs that will plug into a standard outlet. A bowl does not implement the Pluggable interface. Can't plug it in. The book "The Complete William Shakespeare" doesn't implement the Pluggable interface. Can't plug it in. However, an e-book reader does implement the Pluggable interface. You could read "The Complete William Shakespeare" on e-book or in paper form. Same text. Different implementation.

So back to our story.

Some Reasons Why Interfaces Are Very Cool

1. **They encourage smart application design**. Say that you have an application like an e-commerce site. You have to access a database a lot for different reasons. On one occasion, you need to retrieve products and display the catalog; another time, you need to store customer information. Now, these operations aren't related much in terms of the domain. But they do the same thing: interact with the database. You might create an interface called DataAccessor that defines all of the operations that can be done with the database: select, insert, update, delete, and so forth. Then, you have one standard way that all of the programmers on your project can write to. All of your database access contracts, even if written by people working on unrelated parts of the app, will look the same.

2. **They promote readability**. I have seen enough people come and go in different jobs that I am a firm believer in doing what you can to make your code readable. Interfaces promote readability because readers of your code know what to expect of a class that implements an interface. Also, it gives readers a second, concise location to overview what the class does.

3. **They promote maintainability**. The reason that they promote maintainability is more complicated, and we'll get to it in a moment. In a nutshell, if a class implements an interface, you can use the interface type as the reference type of instances of that class. That means that later you can swap out the actual implementation without breaking any of your code. That ability is an instance of polymorphism, one of the pillars of object-oriented programming. More on this in a mo.

4. **They allow flexibility in your class definitions**. In Java, you are only allowed to explicitly extend, or inherit from, one class. This is different from languages like C++ and C# that allow you to extend multiple classes. By allowing Java classes to extend from one class and also implement an interface at the same time, we can in effect skirt the fact that we're not allowed multiple inheritance, because of polymorphism. In fact, Java programmers are often encouraged to implement an interface instead of inheriting from a class when faced with that choice. Programmers probably come to such a crossroads when creating a new thread. You can create a thread by extending the Thread class or by implementing the Runnable interface. They both get you a thread. But you can implement as many interfaces as you want—you can only inherit from one class. So the idea is to prefer interfaces over inheritance unless you really are extending the definition of what your superclass can do. That is, nine times out of ten we aren't actually extending the functionality of a thread when we write extends Thread. We're not making a more specific, more functional kind of thread; we're just wanting to spawn a new instance of a regular old thread so that we can execute some code separately from the main program thread. In this case, you should implement Runnable and get your thread that way. That leaves you with the ability to extend a different class if you want to, one whose functionality your class is closer to, or that you really need to inherit from.

How to Write an Interface

The Java API is full of terrific interface definitions. One commonly used interface is java.lang.Runnable, which can be implemented to create a thread. To write an interface, you can look at the many good examples in the Java API.

When writing your own, you just use the keyword interface instead of class, and then don't include the method implementations. Like this:

```
public interface MyInterface {

      public void someMethod(long someParam);

}
```

As you can see, you just use the interface keyword in place of class. And then you substitute a semi-colon for the curly braces holding the method implementation. That's all you have to do in a nutshell. In the following sections, we'll see the many twisting, cavernous corridors that lurk beneath the deceptively simple interface.

After you have an interface, you implement it. That means you declare a class that implements the interface, and then write the implementation for each method in the interface. And the method signatures must match exactly. Like this:

```
public MyClass implements MyInterface {

   //since I said "implements MyInterface",

   //I must include this method, or

   //this class won't compile

   public void someMethod(long someParam) {

       //this is my implementation.

   }

}
```

You can add methods and other stuff to your implementing class if you want to.

Interfaces Versus Abstract Classes

There are differences between an interface and an abstract class, though commonly the two are confused. Why do you need an interface? When should I use an interface and when an abstract class? Let's let the two duke it out for themselves, and you decide.

Monsters fight!

Round One: *An interface is a totally abstract class*. In an abstract class, you can define some methods that are abstract, and some methods that are fully implemented. Any class that extends that abstract class must implement the abstract methods, but it inherits the functionality as implemented in the abstract class. This is very cool if you need that kind of structure in your program, but can make your API a little snaky.

Round Two: In an interface, there can be no implementation whatsoever. You can say this:

```
List starWarsFigures = new ArrayList(500);
```

Then, your reference type is the interface type (List)! That means that later in your code you can change the implementation without breaking any of the code that does something with your starWarsFigures object. Like this:

```
List starWarsFigures = new Vector();
```

This is very cool if you need that kind of flexibility, which is often a good thing.

Round Three: An interface is less flexible in how its variables and methods are declared. Here are the rules about writing an interface.

Rules for Writing an Interface

- Declare an interface using the keyword `interface`.
- An interface may extend zero or more interfaces if it likes, but it cannot extend a class, because then it would inherit functionality, and interfaces cannot have functionality. They can only talk about it.
- Interface methods cannot be static.
- Interface methods are implicitly abstract. For that reason, you cannot mark them `final` (duh), `synchronized`, or `native` because all of these modifiers tell how you're going to implement the method, and you're voluntarily giving up that ability when you write the method in an interface.
- `strictfp` is okay on an interface. It is okay because you can evaluate compile-time constants using `strictfp` rules within an interface.
- All interface methods are implicitly abstract and public, *regardless of what you write in the interface definition!* It is true. The interface method declaration `void biteIt(int times)`, despite its apparent access level of default, actually has `public` access. Try it. If you write a class in another package beyond the visibility of default access,

and include the seemingly legal implementation of void biteIt(int times) { ; }, the compiler will tell you that you cannot reduce the visibility of the method from public to default. They're all abstract; they're all public.

- An interface can define variables. But all variables defined in an interface must be declared public, static, and final. Many Java programmers have adopted the practice of defining only variables within an interface and putting constants in it. This works to get at shared constants, but it is a workaround and is no longer necessary if you're using J2SE SDK 5.0. It features a new static import facility that allows you to import constants just as you would a class or package.
- It should be obvious by now that an interface cannot implement another interface or a class.
- You may modify your methods using the keyword abstract if you like, but it will have no effect on compilation. Methods in an interface are already abstract, and the Java Language Specification says that its use in interfaces is obsolete.
- Likewise, the interface itself is already abstract. So you can do this if you want: public abstract interface Chompable {}. But there's no point; it's redundant.
- Interfaces have default access by default (!). So this is legal: interface Chompable { }. But if you want your interface to have public access, use that modifier in the interface declaration.
- You cannot declare an interface as having private access. It doesn't make any sense. No one could implement it. So private interface Chompable { } gets you a compiler error for your trouble.
- public, static, and final are implicit on all field declarations in an interface.

There are some weird things to keep in mind.

Interfaces can be declared private or protected if they are declared nested inside another class or interface. The following will compile, though its usefulness is dubious at best.

```
public class interface test {

    private interface myinterface{ }

}
```

Only an inner class of the class of which the interface is declared can implement the interface.

Is and Does

Remember that the job of an interface is to designate a role that a class will play in the society of the application. Whereas a class (especially an abstract class) defines what something is, an interface defines what it can do.

Which of the following would you make into an interface, and which would you make into an abstract class?

Person

Employee

Programmer

Skier

WildAnimal

DataAccessor

Swimmer

WestCoastChopper

You'll see a 3-dimensional pattern emerge if you relax your eyes and stare at it long enough. This sort of thing is important to keep in mind as you design your application. You can do all of your applications without ever using an interface. But again, they clarify your API and provide you with flexibility down the road.

Constants

Java programmers commonly define constants inside their interfaces, if it makes design sense. You can do so using variables in an interface because the values will be present instantly at runtime and their values shared among all classes implementing your interface, because they are static and final.

Here is how you do it.

```
public interface IDataAccessor {

  String DB_NAME = "Squishy";

}
```

FRIDGE

Like the public and abstract deal, interface variables are implicitly public, static, and final. That is, the following are equivalent within an interface: public static final String DB_NAME = "Squishy" and String DB_NAME = "Squishy". Likewise, you are not allowed to do this: private String x;

Note that it only makes sense to define interface constants if the variables are tightly related to your interface definition. There is a pattern that has been popular among developers where they use interfaces for the sole purpose of defining constants, so you might come across a lot of code to that effect. But J2SE 5.0 introduces other mechanisms that are more appropriate for this sort of thing—namely typesafe enums and static imports.

The reason that you don't want to use this old pattern anymore (assuming you're using it) is that it confuses the API. To access the constants defined in the interface, you must include implements InterfaceName in your class definition—and that isn't really true. You aren't implementing any functionality defined by such an interface; you just want the variables. It's a workaround, and now there are language features to handle that situation, so you don't have to resort to it.

Interface Inheritance

Yes, an interface can extend another interface. Just say that it does, like this:

```
public interface ISpy extends

    IInternationalManOfMystery {...

}
```

This is rarely used in practice. It means just what you would expect: Now the class that implements ISpy must also implement all of the methods from IInternationalManOfMystery.

Also, an interface is allowed to extend more than one interface. Just separate them by commas. Notice that this is different than regular classes, which are not allowed to extend more than one class.

Implementing Interfaces

You have to do two things to implement an interface. You have to announce that your class will implement the interface in your class declaration, like this:

```
public class SuperHero implements IFly {...}
```

Then, you have to implement each method defined in an interface. Most IDEs, including Eclipse, will tell you the names of the methods you need to implement as a convenience. Note that whether you have implemented each method you're supposed to is checked at compile time, not at runtime.

FRIDGE

It is a naming convention that you'll often see to prefix "I" before the name of an interface in order to identify it easily as an interface. If you've programmed in VB, you're probably used to Hungarian notation, which serves a similar purpose.

Remember that the purpose of an interface is to say that the implementing class must have methods that match the signatures. That doesn't say anything about the quality of the implementation. Say we have an interface called IFly (which we'll define next), and it has one method `void accelerate(int speed);`. The following is legal, and will compile:

```
public class SuperHero implements IFly {
    public void accelerate(int i) {}
}
```

This can be convenient if an interface has a method or two that you don't really need. But if you find yourself doing this sort of thing with any frequency, you might take a second look at whether you really ought to be implementing that interface, and if there isn't some other way to solve the problem. If you are the designer of the interface, you might want to reexamine your interface design to make sure that you are defining what users (in this context, programmers who want to implement your interface, including you) really need. By the same token, if you find in your code a number of methods that programmers need to define over and over again, consider moving them into the interface and making them part of the contract. This could have the effect of tightening and strengthening your abs. I mean your API.

A class can implement more than one interface. Separate each interface with a comma in the class declaration, like this:

```
public class RockStar implements IGuitarPicker,
    ILeatherPantsWearer { }
```

Abstract Class Implementation of Interfaces

This is something very weird. Actually, I'll pose it as a question and see what you think. Previously, we had our IFly interface, and our regular old class SuperHero implemented it.

(Deep, oily announcer voice) For 50 points, can an abstract class implement an interface? *(After slight pause, with greater significance)* Can an abstract class implement an interface?

Think about it (after all, it's worth 50 points). Can an abstract class implement *anything*? Actually, yes, it can. It can have both abstract and concrete methods. So the answer must be yes. Because you can do this:

```
public abstract class SuperHero implements IFly {
   public void accelerate(int speed) {
      // some code here to do accelerating...
   }
}
```

Any class that extends SuperHero will inherit this functionality, or alternatively, it can decide to override the accelerate() method if it wants to, and define a different implementation.

That was easy enough. But here's the killer: Is the following legal?

```
public abstract class SuperHero implements IFly { }
```

Notice that there is not only no mention of the accelerate() method, there is no code at all. How could an abstract class say it implements an interface that it doesn't implement? It must be wrong!

Actually[1], that code compiles without a problem. Because there is another contract at work when you define an abstract class, and that contract is that the interface methods must *have been implemented* by the first concrete subclass of the abstract class. The first concrete class is not required to implement all of them, just those that have not been previously implemented by some abstract class that implements the interface up the hierarchy. The following series of classes is legal and compiles.

[1] I hate it when people say, "actually." It sounds soooo snotty. As if there is some other life that you're living over there in happy fantasyland where everybody has all wrong ideas about stuff and you're so dumb that it can't be imagined how you even function.

IFly.java

```
public interface IFly {

     void accelerate(int speed);

     void slowDown(int speed);

}
```

Now an abstract class will say it implements the IFly interface.

SuperHero.java

```
public abstract class SuperHero implements IFly { }
```

JusticeLeagueSuperHero.java

```
public abstract class JusticeLeagueSuperHero

          extends SuperHero

{

     public void accelerate(int speed) {

       //some real implementation code!


       System.out.println("Accelerating to " + speed + "mph");

     }

}
```

Notice that JusticeLeagueSuperHero implements the accelerate () method, but not the slowDown() method, which is also required by the interface. The code compiles because it is also declared abstract, and it is expected that one day someone will want to make a concrete subclass of JusticeLeagueSuperHero, so that you can say new SuperHero() and get to work. Of course, if no one ever makes a concrete subclass that implements the slowDown() method, the interface is not really implemented, even though we said it was. But that couldn't matter less, because we can't ever make any objects.

So now let's look at the final class in our implementation of this interface, the concrete class SomeMoreSpecificSuperHero.

SomeMoreSpecificSuperHero.java

```
public class SomeMoreSpecificSuperHero

        extends JusticeLeagueSuperHero {



  public void slowDown(int speed) {

    //some real implementation code here!

    System.out.println("Slowing down

        to " + speed " mph");

  }

}
```

In our little hierarchy, we have an interface, an abstract class (SuperHero) that implements the interface but doesn't actually implement any methods, a second abstract class (JusticeLeagueSuperHero) that extends SuperHero and provides a concrete implementation of one method in the interface, and then our first concrete class (SomeMoreSpecificSuperHero) that extends JusticeLeagueSuperHero and implements the remaining method. By the time we get to the concrete class, everything has an implementation. Cool.

Well, I'm going to get a beer and make a pizza. Thanks for talking about interfaces with me. I'm feeling much better now.

Chapter 18. CASTING AND TYPE CONVERSIONS

DO OR DIE:

- Stop
- Drop
- Roll

Here we go, into the land of casting. This is where we tell you to empty the rusty old coffee can containing your meager savings. You'll need a bus ticket, since we're sending you to Hollywood to be in pictures.

It's going to be just terrific.

But in case our blockbuster career gunning down everything in sight alongside an ever-balder Bruce Willis doesn't work out, let's do what every out-of-work Hollywood actor does: get a day job by learning runtime type information and casting in Java. But just until we get our big break.

Casting

The purpose of casting is to allow conversions from one type to another.

To get your mind around casting, it is helpful to think of the primitive types as being buckets of bits. Some buckets are bigger than others: long is a bucket of 64 bits, float is a bucket of 32 bits like int, and so forth.

Here is an example using primitives:

```
short s = 16;

//this is okay without cast, since int

//is a bigger bucket (32 bits) than short (16 bits)

int i = s;
```

So we can do this, right?

```
short s2 = s + 26;

//Wrong!! Here's the output:


net\javagarage\casts\Casting.java:21:

    possible loss of precision

found : int

required: short

            short s2 = s + 26;

                   ^

1 error
```

Where does Java say the error is? At the + operator. And what exactly is wrong with trying to add two shorts together to make another short? Nothing at all. Except, despite appearances, that's *not* what we're doing.

When you perform an operation like this on integral primitive types, the two operands are automatically promoted to ints, even though the result would fit inside a short. That's what always happens when you perform mathematical operations on integral types. To be promoted in this context means that they get to be a bigger sized bucket. And you can't fit an int-sized bucket into a short-sized bucket, can you?

Well, no you can't. In this case, we know that our resulting number (16 + 26 = 42) is plenty small enough to really be stored in a short. So we want to tell the compiler that we know it will be okay. Little E.T. will get home in the end. The cast in Java is the programmer's way of saying, "I know that you don't want me to do this, and you're usually smarter than I am, especially about Java, but I just need to do this right now, so you're going to have to trust me. If it blows up at runtime, I'll take responsibility."

The cast operator is (…). In between the parens, you put the name of the type to which you want to cast.

So here's how we solve it:

```
short s2 = (short) (s + 26);
```

This gets us what we want: a short containing the result of 42. Notice that we have put parentheses around the addition operation. Why's that? Because if we didn't, the (short) operator would cast the variable s to a short (which it already is) and would then try to perform the addition, and what would happen? You'd be adding two shorts together again, and the operation would promote the result to an int implicitly, and the compiler would blow up. Well, it wouldn't exactly "blow up," but it would become momentarily unhappy, and take it out on you in the form of console complaints.

When you cast, you leave yourself vulnerable to blowing up at runtime. Why's that? Because you are taking the reins. And you might not know as much about bits at runtime as the JVM. Not to rain on your parade (and we were just getting so excited about casting), but can you guess what happens here?

```
//this is the most an int can hold:

//2147483647

int j = Integer.MAX_VALUE;


//knock it over the edge

int result = j + 1;


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

This is an error, right? No! It's perfectly legal. So what in the world does it print? An int can't, by definition, hold more than it can hold. It prints this:

```
-2147483648
```

Overflow is what happens when a result is too big for the type of variable that is meant to hold it (that is, the int's bucket o' bits isn't big enough to hold +2147483648). What we've got here is an overflow.

If an integer overflows, only the least significant bits are stored. So exceeding your bucket's limit causes an overflow that results in a negative number—in this case, the smallest possible value for an int. So you can guess what is printed here.

```
j = Integer.MIN_VALUE;

result = j - 1;
```

Yep: 2147483647. It's like those revolving fireplace doors in those old Vincent Price movies. It just comes back around.

Java handles this situation differently for integer types than for floating point variables. When a double or a float overflows, the result is positive infinity. When a double or float underflows—that is, when it is too small to be represented properly—the result is 0.

Illegal operations, such as dividing by 0, result in NaN (Not a Number). Otherwise, floating point expressions will not raise exceptions; they will simply default to one of these predefined modes.

Note: You cannot cast a boolean primitive to any other primitive type in Java. It is "special." Shhhhh. This means you cannot convert a boolean to a 0 for false or a 1 for true as in some languages. A boolean's values are the literals true and false, and those values aren't strings. They're booleans.

Also note: If what you need is to perform some hard-core mathematical operations including unbounded arithmetic, you can use the five classes in the java.Math package (not java.lang.Math) BigInteger and BigDecimal. These guys perform more slowly than their counterparts in primitiveland do, and so they aren't used unless necessary.

And now, back to our previously scheduled topic. We were talking about casting.

This code is a compiler-error-making bit of code:

```
int I = 37F; //No!
```

The compiler will get mad, saying that you stiffed it. You told it you were going to provide an int, but instead you gave it a float.

The compiler doesn't like the following either (sort of picky, this compiler…):

```
float f = 69.99;
```

Why in the world not? Remember how Java likes its integral types to be integers by default? It likes its floating point numbers to be doubles by default. So here, we said that we'd give f a float, but instead, we gave her a double.

You have to make it explicit then, using the F following the float. This fixes it:

```
float f = 69.99F;
```

You can also write D following doubles.

```
double d = 87.65D;//ok jose

double d = 122984.8604 //ok jose
```

Note that casting from decimal to integral types can result in the parts following the decimal simply getting lopped off, as with a guillotine.

```
double d = 1405.9876;

int i = (int) d; //gives 1405
```

Of course, hacking apart your floating point numbers in the manner of the chain saw may not prove aesthetically pleasing enough for you. In this case, you can try the java.lang.Math.round() method, which returns a passed float as an int.

Casting Between Primitives and Wrappers

Here are the rules about it:

A value of a primitive type can be cast to another primitive type by identity conversion, if the types are the same, or by a widening primitive conversion or a narrowing primitive conversion.

A primitive type value can be cast to a reference type by boxing conversion.

```
int i = 5; //5 is a primitive int

Integer iRef = i; // convert to wrapper

//reference type: ok
```

A reference type value can be cast to a primitive type by unboxing conversion.

```
Integer iRef = new Integer(5); //starts life as

reference type

int j = i; //ok, unboxed to primitive
```

See the topic on Autoboxing for more about moving seamlessly between primitive and wrapper class types.

Casting with Reference Types

This is very similar to casting with primitives. But it's different. And there are some rules, young lady. Deviation will not be tolerated.

Simple Casting

A reference of any object can be cast to a reference of type java.lang.Object.

You can cast up the class hierarchy, and down the class hierarchy. But you can't cast laterally. This means that a reference to any object O can be cast to a class reference that is a subclass of O if, when it was created, O was a subclass of that class.

You can cast up the class hierarchy automatically—that is without using the cast operator.

```
Object o = "some string";
```

Object is up the hierarchy because String extends Object. String is down the class hierarchy from Object. If you cast down the class hierarchy, you must use the cast operator.

```
Type t = (Type) (someObject);
```

A cast could compile and yet still fail at runtime if the cast turns out not to be legal. With the new generics facility in Java 5.0, some of this problem should be alleviated. (See the topic on Generics in *More Java Garage*.) If it turns out that a compiled class cannot cast at runtime as it hoped it could, a ClassCastException is thrown.

Subclass Casting

Let's say we have the following class hierarchy:

```
class A {}

class B extends A {}

class B1 extends A {}

class C extends B {}
```

Those definitions are legal, and you can type that into a source file. As long as one of them is public, and its name matches the source file name, it will compile. Here's what we can do:

```
A a = new B();//ok—B is subclass of A!

B b = a; //Error!! can't go up the
         //hierarchy w/o casting

B b = (B) a; //could possibly fail at runtime

b = (B1) b; //nope—lateral cast not allowed

A a1 = new C(); //ok—C is indirect subclass of A!

C c = new A(); //no—C is "narrower" than A

C c1 = (C) (new A()); //compiles and fails at runtime
```

Interface Casting

A reference to any object O can be cast to an interface reference if one of three things is true.

1. If O's class implements that interface
2. If O is a subinterface of that interface
3. If O is an array type and the interface is the interface Cloneable or Serializable

Let's say we have these classes and interfaces:

```
class A {}
class B extends A implements IInterface {}
class B1 extends B implements ISubInterface{}
class C extends B1 {}

interface IInterface {}
interface ISubInterface extends IInterface {}
```

Here's a class showing what we can do.

IntCast.java

```
public class IntCast {

public static void main(String[] args) {

IInterface b = new B();
//ok, B implements IInterface

//b = (B) new A();
//compiles, but fails at runtime
//since A doesn't implement IInterface

IInterface b1 = new B1();
//works great—B1 implements a subinterface
//of IInterface

//C c = new IInterface();
//NO! you can't instantiate an interface
//they're all abstract!

A a = new C();
//that's just fine.
System.out.println(a.getClass());
//prints C

}
}
```

Perhaps the most common version of using the interface reference type is with the Collection interface.

```
Collection c = new ArrayList();
```

Because the ArrayList class implements the Collection interface, you can do this. Then, you can make your methods accept Collection types; if later you decide you want to implement your shopping cart or whatever you're using the ArrayList for as a different kind of list, you can do that without breaking any contracts with clients of the method.

```
public void doIt(Collection c) {

//Now I can use an ArrayList here or a Stack or a
//LinkedList or...
}
```

Array Casting

You can cast an array, but the data they hold must be of compatible types. For example, this code will not compile:

```
double arr1[] = {1.5,3.14};
int   arr2[] = (int) arr1;   // compile-error
```

But arrays holding compatible types can be reassigned to a different reference type. This is all legal:

```
int[] i = {1,2,3};
int[] j = {3,4};
i = j;
java.io.Serializable s = i;
java.lang.Cloneable c = j;
```

Arrays cannot be converted to any class other than Object or String.

Arrays can be converted to interfaces only of type java.io.Serializable and Cloneable.

Primitive Widening Conversions

Different than casts, which the programmer must perform explicitly, the widening conversion happens automatically.

Widening conversions happen when one primitive type is promoted to another primitive type of greater capacity. For instance, an int can hold only 32 bits of data. A long, on the other hand, can hold 64 bits of data. That makes the following code legal:

```
int i = 10;

long lg = 20;


lg = i; //ok, lg now = 10
```

Likewise, primitive integer types are implicitly converted to floating point types, like this:

```
int i = 10;

float f = i;//ok, i = 10.0
```

If you want to squish a big bucket o' bits into a smaller bucket o' bits, you have to cast.

Every type can be converted to String, even parameterized types.

```
ArrayList<Integer> a = new ArrayList<Integer>();

a.add(9);


    //remember that passing any object to println

    //automatically calls its toString() method


System.out.println(a); //prints [9]
```

Note that in general, these conversion rules can be extended for parameterized types as well. The following, as you might guess, is illegal:

```
ArrayList<Integer> a = new ArrayList<Integer>();

a.add( 67L );//No! looking for int, found long
```

Reference Conversions

You can convert from any class, interface, or array (arrays are objects…) to an Object reference, or to null.

You can convert from any class to any interface that it implements.

You can convert from any interface to any of its superinterfaces.

For more info on array casting, see the Arrays chapter.

Narrowing Conversions

Narrowing conversions cause you to lose information about the primitive being converted. You can lose precision and magnitude data. So, the compiler requires that you explicitly cast in order to do this.

```
int i = 10;

float f = i;

i = f;//illegal without explicit cast
```

When you perform a narrowing conversion, you lose all but the lowest order bits.

You cannot convert any primitive type to null. The compiler complains of "incompatible types" if you try to do this.

You cannot convert boolean types to any other types.

I know I already said that. I don't mean to be a nag or anything. It's just, you know, maybe you're reading out of order, or didn't think that was the most fascinating thing you ever heard and needed a little reminder.

Wrapper Conversions

Any primitive to any reference is disallowed except for the following.

Primitive to String is always okay.

Primitive conversion to its corresponding wrapper class is okay, due to autoboxing, and a wrapper class to its corresponding primitive is okay (due to auto-unboxing). The following will not work:

```
Float f2 = 10;//No! 10 is an int

Long g = 20;//No, even though int is smaller than

            //long, this is the Long wrapper class
```

The int is the default integer type. The following doesn't work either:

```
Float f2 = 10.99;//No! 10.99 is a double
```

The double is the default floating point type. Both of the following work:

```
Float f2 = 10.99f;

Float f3 = 10.99F;
```

Cool.

Chapter 19. INNER CLASSES

STARRING...

- Inner Classes
- Method Local Inner Classes
- Anonymous Inner Classes
- And Special Guest Appearance by Static (Top-Level) Inner Classes as The Beav

Over-read on IM…

Zilly: can't you get him to come out? He never comes.

Elmo: he hates movies.

Zilly: well. So do I I still go :P

Elmo: yer not as internally motivated as MisterLister :)

Zilly: i'm not internally motivated period duh.

Elmo: since he got javagarage all dude wants to do is mope around his apt reading java books

Zilly: tell him theres hot chix there. then he'll come out with us.

Elmo: hot chix at de passion of de christ? uh.

Zilly: tell him hot chix with java books.

Elmo: lol…

This part is all about inner classes. Inner classes are not the kind of thing that you need to deal with much of the time. In Swing programming, where you make graphical user interfaces, you will likely need to use them. But if you go on to write Java on the server side using servlets and JSPs, or if you write a console application, you aren't likely to ever need them.

But they are important in GUIland, and that is a non-trivial land to us here. So let's check them out, see how to use them, and get on with our lives. My eyes are burning. Aren't yours?

Regular Inner Classes

Recall that you can write many different classes in one source file. You can also write a class inside of another class.

Why in the world would you want to do that? Well, they sort of allow you to scope your classes. The inner class becomes a member of the outer class. That is, the inner class acts as a member of the outer class, just like its methods and fields.

Important thing: Objects in an inner class have access to even the private variables in instances of the outer class.

You typically use an inner class to define a helper—that is, a class with a very specific purpose. In GUI development, you need to create listeners that handle events such as mouse clicks. Often, these events are handled using inner classes.

There are four kinds of inner classes that are tailored for different purposes. The first is the easiest.

It is simple business, defining an inner class. Try something like this:

```
class MyOuterClass {

  class MyInnerClass {

  }

}
```

To instantiate an inner class, you must first have a reference to the outer class. In general, an object of an inner class is created by the outer class itself.

A class can define as many inner classes as it feels like putting up with.

An advantage to using inner classes is that they can be hidden from other classes in the same package, so you can restrict access to their functionality.

Inner classes can be declared private. This is different than regular classes, which must be default or public.

Using Method Local Inner Classes

The second type of inner class is the method local inner class (MLIC). The name sez it all. We're talking about a class defined inside of a method body.

The primary difference between a regular inner class and an mlic is that the MLIC cannot use the local variables of the method that contains the inner class definition. Only in the event that local variables or arguments are declared final can an object of an inner class access them.

OuterMethodLocal.java

```
package net.javagarage.inner;

/**<p>

 * Demos Method Local Inner Class

 * </p>

 **/



public class OuterMethodLocal {

public final int x = 5;

public int outY = 10;


//the method

int someMethod() {

final int inX = 5;

int inY = 10;


class InnerClass {

//do some wicked stuff

int getLucky() {

//return inY; //NO! inY is not final!

//return outY; //OK

return inX;//OK —it's final!

}

}


//instantiate method local inner class

InnerClass inner = new InnerClass();
//call a method while you're at it

return inner.getLucky();


}//END OF someMethod


//I can't do this here—this class

//is only available inside the method!

//InnerClass inner = new InnerClass();


public static void main(String[] ar){

OuterMethodLocal out = new OuterMethodLocal();

System.out.println(out.someMethod());

}

}
```

Using Anonymous Inner Classes

Method local inner classes define a class inside the body of a method. But you can also define a class as a method parameter. The anonymous inner class is called anonymous because it is never given a name.

The key distinguishing characteristic of anonymous inner classes is that they are declared and instantiated at the same time.

The most common place to see anonymous inner classes is in GUI event handlers. The idea is to keep the code that handles an event in the same place where the control is defined. You create them when you need a class that you only need to call from one place, when one specific thing happens. Because they're defined as part of a method, they must adhere to the same restrictions that method local inner classes do.

Here is an example:

```
myButton.addActionListener(
    new ActionListener() {
        public void actionPerformed(
            ActionEvent e) {
        //do the work here!

        } //end actionPerformed method
    }//end anonymous inner class
); //end addActionListener method call
```

The addActionListener() method takes only one argument, of type ActionListener. ActionListener is an interface that listens for action events. You process action events in Swing by writing classes that implement this interface. After an object of that class is created, it is registered with a component using the addActionListener() method. That method is the same one that any class that implements the ActionListener interface would need to implement. So that is a little weird: it seems that you're making an instance of an interface, which you normally can't do. You can also do the same thing with abstract event adapter classes as well, such as WindowAdapter or MouseAdapter. It's the only place you're allowed to use the new operator on interfaces and abstract classes. That's because your anonymous class automatically becomes a subclass of that class.

The class can be anonymous because you'll never use this class again. It is okay—perhaps even desirable—that it has no name or definition outside of this context. The clicking of the myButton control is the only conceivable time you would want to invoke this code. Any other GUI control on your page is going to have to define its own action handler.

Static Inner Class

Inner classes can be static, meaning that they have access only to the static member variables of the outer class. The reason is that they don't share much of a relationship with the outer class—they're tied to the name of the outer class, but not an instance of the outer class. A static nested class can be instantiated without an instance of the outer class. You can access it just like any static member variable. These are also known as top-level nested classes because they really boil down to a way to control namespace.

For example, in C#, you can define arbitrary namespaces that have no correspondence to actual directory structure (not so in Java), and you can nest namespaces within a class or a bit of code almost willy-nilly (again, not so in Java). So, the static nested class lets you get a similar level of namespace control. To which I say, "Big deal."

So, the static nested class has a life all its own, without an outer object enclosing it, and is finally free to pursue its amateur ham radio license. I have to say, though, that these things ain't used much.

StaticNestedClass.java

```
package net.javagarage.inner;

/**<p>

 * Demos static inner class usage.

 **/

public class StaticOuter {


    //an instance of inner object cannot

    //directly reference this, nor can its methods

    int outerVar;


    //an instance of inner object cannot

    //directly reference static members—

    //BUT the members

    //(methods and fields) defined inside the

    //StaticInner class CAN

    public static int outerStaticVar;


    //Here is the Static inner class, defined

    //like a member of the StaticOuter class

    public static class StaticInner {


   int innerVar;


    //The StaticInner class methods can
    //access only the static members of
    //the enclosing class
    public int getOuterStaticVar() {
        return outerStaticVar;
    }
}//end StaticInner

public static void main(String[] args) {

    //this statement creates a new instance
    //of the StaticInner class ONLY
    StaticOuter.StaticInner inner =
                    new StaticOuter.StaticInner();

    inner.innerVar = 5;

    //Instantiating the outer class does
    //NOT give us any instance of the
    //StaticInner class
    StaticOuter outer = new StaticOuter();

    //yeah, yeah outer can reference its
    //OWN members per usual...
    outer.outerVar = 10;
    StaticOuter.outerStaticVar = 20;

    //Wrong! I can only access this guy
    //from WITHIN the methods of
    //the StaticInner class
    //inner.outerStaticVar = 20; //wrong!!

    //prints 20.
    System.out.println(inner.getOuterStaticVar());

  }
}
```

The methods of a static inner class can access only the static members of the outer class.

So that ends our adventure through the fascinating land of inner classes. They are often useful when applied in conventional ways, such as implementing an actionPerformed method for a JButton. Just be careful not to overuse them and possibly introduce unnecessary complexity and confusion in your code. As my friend Erin from Indiana used to say, "eschew obfuscation."

PREV < Day Day Up > NEXT

Chapter 20. BLOG: INNER CLASSES AND EVENT HANDLERS

Beware when using inner classes, anonymous inner classes in particular. They are the subjects of much debate within the Java community. Developers who don't like their use say they break encapsulation—one of the key principles of object-oriented programming—and that they're hard to read and difficult for beginners to understand.

Developers who promote their use during late night local television shows point to their convenience and elegance. You need to know about them to read and write Swing application code. But in general, you don't need them, and your code is probably clearer and more flexible without them.

My rule is this: Use anonymous inner classes to handle actions and events. Don't use the other kind of inner classes. I think that's where the true confusion can come in, given how inner and outer classes share their members.

But anonymous inner classes solve the problem of the Monolithic Switch Block—an antipattern familiar to coders throughout the land. Come on. I know you've done it. The first step to getting help is admitting you have a problem....

Whenever you see code like this, where the frame itself implements the ActionListener interface....

```
public class MyGUI extends JFrame implements

        ActionListener {

  protected JButton btnSave;

protected JButton btnCancel;

//and so on

//have to implement this guy because he

//showed up with the ActionListener interface

  public void actionPerformed(ActionEvent event) {

String command = event.getActionCommand();

//use giant if/else block to determine

//who fired this event

     if (command.equals("Save...")) {

        // do something:

handleSave();


     } else if(command.equals("Cancel")) {

        //and so on...


private void handleSave() {

//do work here

}
```

…you know there's too much Liquid Drano in the code designer's latté. This is not good object-oriented design. In general, this kind of construct can grow fairly large, and changes in one place in the code often require changes in another part of the code. Moreover, if/else blocks require the runtime to test the conditions, and this means that your app could slow down the larger it gets.

Why run a bunch of tests you don't have to?

The same deal implemented with an anonymous inner class would look like this:

```
//look ma! no "implements ActionListener"

//on the class!

public class MyGUI extends JFrame {

createGUI() {

//create the buttons

   JButton btnSave = new JButton("Save...");

JButton btnCancel = new JButton("Cancel");


//add action handlers

btnSave.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

//do something for save

}

});


btnCancel.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

//do something for cancel

}

});

//...
```

Anonymous inner classes solve some problems: The code is always where it should be, and you'll get better performance. It is probably polite to seriously limit the amount of code you put into an anonymous inner class if you can. Of course, you gotta do what you gotta do. But if you are going to put a lot of code into an event handler, you should just collect the data in the ActionListener, and then start a new thread inside it to do your work. Otherwise, your GUI will be frozen waiting for your ActionListener to get done.

Following is an example of how to start a new Thread in your ActionListener, which makes your GUI more responsive. It also delegates the actual work of the action to a handler, and does not attempt to do all of the work in the GUI class itself.

```
JButton btnSearch = new JButton("Search");

btnSearch.addActionListener(new ActionListener() {

    public void actionPerformed(ActionEvent e) {

      Runnable runner = new Runnable() {

        public void run() {

                  try {


ArrayList results = searchController.search();

    createAndShowResultsPanel(results);

           } catch (BandSException bse) {

             //handle

           } catch (Exception ex) {

             ex.printStackTrace();

             //handle

           }

         }

       };


       Thread t = new Thread(runner);

       t.setName("Search Thread");

       t.start();

     }

  });
```

We call a controller (which I assume above we already have a reference to) as a delegate to do the work. This is the way you will usually want to handle button clicks and such—it separates the GUI (code that creates the user view) from the business logic.

Chapter 21. HANDLING EXCEPTIONS

DO OR DIE:

- Find out about exceptions.
- This is all about exceptions.

Exceptions

The two lean men sat coiled, perched across from one another, their boots stained with calves' blood. Their hands were rough, like raw leather. The man facing the bar's door dug his knife further into the raw pine table. The man with the moustache eyed the twisting knife only half afraid.

Moustache: Tell me again.

Knife: There's two kinds.

Moustache (shoving the chair out from under him): I know that! Don't you think I know that already?

Knife (stretching back his arms in a peacock display of relaxation—just enough to reveal the six shooter under his arm): Two kinds. Checked. And unchecked.

Moustache: And it is something that happens—

Knife: An *event*.

Moustache: Yes, an event that can occur during the execution of code.

Knife: It's not just any event. It's the worst kind of event. It could mean the end…

Moustache pales in horror. The men slug their near-frozen beers.

Moustache (eyes suddenly alight with fire, he pounds both fists on the table): You will tell me about them! Tell me about these exceptional events……

What an Exception Is

An exception is some bad news that you get told about during the execution of your program. There are many reasons for this. Your database could be down, and then the application that is trying to connect to it would not be able to do what it wants. Network problems are common causes of exceptions. Your program could rely on a file that cannot be found, or a user could enter something that is just ridiculous. You have to deal with these problems because they happen.

An exception is one result of poorly written code. If you write quick and dirty code, you should expect to have problems later. A good example of this is, oh, anything written in VB (which explains the comparatively large number of VB programming jobs on the planet).

An exception is different from an error. An error is something that you can't do much about. That is, your program probably would not be able to recover from an error at all. This is something like you're completely out of memory. In some circles, it is considered pedantic to make a distinction between exceptions and errors, and you'll commonly see the two terms used interchangeably. In the interest of preserving what little is left of our collective sanity, let's just call it six of one, half dozen of the other.

An exception is something that you can create to take care of an ailing situation, and reroute the flow of the application.

An exception is something that you *throw*, or that the runtime itself throws. Throwing means that you make it available again to another part of the program, make it bubble up the call stack—it means you.

An exception can be part of the method signature. You can write, "I know that I'm doing potentially dangerous stuff in this method, but I do not want to deal with anything bad that happens. I am going to pass the buck back up to the method who called me." Here's how you do that:

```
  public void someMethod(int someID) throws
        SQLException { ...
}
```

An exception is an instance of a java.lang.RuntimeException or java.lang.Error, or an instance of a class that extends one of them. Runtime exceptions are also called *unchecked exceptions*.

An exception that you create is an instance of a java.lang.Exception, which implements the java.lang.Throwable interface.

When an exception occurs, you should deal with it if you can.

Dealing with Exceptions

At nearly any point in your code, you could just do the following:

```
throw new RuntimeException();
```

And off your program goes, to the next block of code in your app that is prepared to handle such a statement. That is definitely changing the flow of your application abruptly.

What do you do when an exception crops up? You can do one of two things: you can deal with it or you can make someone else deal with it.

To deal with it, catch the exception in a catch block and do some real work to handle the situation. Perhaps this just means printing a user-friendly message to the screen so that the user knows what is happening and can notify someone. It could mean logging the problem and sending an e-mail—whatever is appropriate for your application. You can do that like this:

```
try {

// code that might throw an exception

} catch (Exception e) {

// handle the exception here

}
```

My dad used to tell me that if I didn't take care of my own business, somebody else would. I think that's true. I know it's true in Javaland. If an exception is never caught by you, it is caught by something called the default handler. Just like, you don't want somebody else taking care of your own business, you don't want the default handler handling your exceptions. If it does, it will halt execution, shut down the VM, and print a giant error message and stack trace to the screen.

But we can cause an exception in one method, ignore it, and then the exception will bubble up to the calling method. This is the behavior all the way back to the main() method that started your program.

Try not to let yourself get confused by the name RuntimeException, despite the fact that it does not distinguish this method from any other. That is, any exception that you would deal with inside a try/catch block would happen at runtime.

Built-in Exceptions

The Java programming language comes with many exceptions built into the language. These cover many different types of common, unfortunate situations. For example, ArrayIndex-OutOfBoundsException is popular. It is thrown to indicate that you have attempted to access an index that does not exist in an array. The system will throw this exception if you have an array of 4 elements, and you type x = myArray[100] (no 100th element). Or if you type x = myArray[-1] (cannot be negative value). Or if you type x = myArray[4] (there are only elements 0-3 in a four element array). This is a good example of the kind of thing that can happen easily, but from which it also should be easy to recover.

UncheckedExceptions

The ArrayIndexOutOfBoundsException is not something that the compiler will force you to plan for when you are writing your application. If you write some code that works with an array, the assumption on the part of the language designers is that you will be careful enough about what you are doing that you should not need to write a complete try/catch block every time you access an array. If you *do* get an ArrayIndexOutOfBoundsException, it is almost certainly because you have some poorly written code in there that needs permanent fixing.

Because the compiler doesn't check if you are handling the possible occurrence of this kind of exception, it is called an unchecked exception.

It means that you can compile the following code:

```
int x = myArray[i];

//look, ma—no try catch statement !
```

NullPointerException is another unchecked exception that you run into during development. Try typing the following:

```
Object obj = null;

obj.toString();
```

You can't call the dot operator on null. Your program blows up. The language designers know that you know this, and they don't want you having to clutter your code with tons of handling statements. So, NullPointerException is unchecked.

If you try to divide integers by zero, the virtual machine will generate an ArithmeticException. Again, you are trusted not to do this, and therefore spared the red tape and inconvenience of planning for it.

You are not required to handle any exception that is a RuntimeException, or one of its subclasses.

CheckedExceptions

Checked exceptions, on the other hand, must be dealt with. The compiler checks to see that you have at least something that looks like it will handle exceptions any time you invoke an operation that could generate a checked exception.

An IOException, a MalformedURLException, and a SQLException are all good examples of checked exceptions.

You cannot compile a class that contains this code on its own:

```
File f = new File("s");

f.getCanonicalFile();
```

If you try to compile the preceding code, your compiler complains that you have an unhandled exception lurking in there. Something inside that code block declares that it throws an exception of type IOException, and so you need to deal with it. The following will fix it:

```
public void someMethod () {

    File f = new File("s");

    try {

        f.getCanonicalFile();

    } catch (IOException ioe) {

       //do something here

}

}
```

The getCanonicalFile() method of the File class might throw an IOException because it is possible that the call would require reading the file system. The creation of the new File object does not need to be wrapped inside the try/catch block because it just creates an object of type File—it doesn't actually do anything on the file system. So, what you're doing is no more potentially problematic than the following:

```
String s = "well, hello Mr. Fancy Pants";
```

The compiler cannot judge the quality of your plans to handle the potential IOException. In fact, your catch block can be empty—you can write no code at all between the curly braces. But you have been forced to acknowledge a potential danger.

Throwing Exceptions

Your method might not be the very best place to handle an exception. It might make more sense to let the exception pass through your method unhandled, and make some other guy deal with it. You can do that. But you have to say that that is what you're doing. You do so with the throws keyword. The following modified method will also compile and run:

```
public void someMethod () throws IOException {

    File f = new File("s");

    f.getCanonicalFile();

}
```

Do not declare that your method throws unchecked exceptions that inherit from Runtime-Exception. Do not write the following:

```
void do(int i) throws NullPointerException

   //BAD DOG! NO!

{

   //blah blah blah

}
```

Why should you never do that? Sometimes the truth is tough. If you are that worried about your code generating a NullPointerException, you probably have an idea of where the problem is, and you should do something to deal with it at that point. Remember that because unchecked exceptions mean that you are trusted, don't advertise to the world with a great big throws sign, "Hey! Look at me! I wrote some crappy code! I might divide integers by zero!" It's just unseemly. Alternatively, sometimes unchecked exceptions occur because of things that are totally out of your control. I say, "If there's nothing you can do about it, there's no point in talking about it."

One other note: You can throw multiple exceptions by separating them with a comma when you declare them. Like this:

```
public void myMethod() throws FileNotFoundException,

    EOFException
```

Of course, it is legal to write something like the following:

```
private void myMethod() throws IOException,

FileNotFoundException...//Don't do this!
```

But don't do that. Because it is redundant. IOException is bigger (higher up the inheritance hierarchy than FileNotFoundException), and you might as well just deal with what you should deal with, and stay on track.

You can also throw an exception using the new keyword if you want to reroute the flow of your program. You do so like this:

```
public void calculateBillTotal(float bill,

                          float tax,

                          float tip)

             throws CheapskateException {

if (tip < bill * .15)

     throw new CheapskateException(

          "Fork it over, cheapskate");

return bill + tax + tip;

}
```

In the preceding code, if the tip is less than 15% of the bill, the exception is thrown up to the caller. The return statement that adds the parts of the bill together will never be executed.

Note one other thing. Sometimes you have to say stuff like this, just to make sure. There's a nut job in every crowd: *you can't throw anything that ain't throwable*. That is, don't try the following, smart guy:

```
public static int myMethod throws String

//you know better
```

How would you like it if people came to *your* house and threw a JFrame? Somebody could get hurt on one of those things.

> *"Is plastic okay? Maybe. Maybe nobody can say for sure…"*
>
> *—J. Benton, 1.16.04*

Catching Exceptions

In the preceding examples, all of the methods throw an exception up to the caller. But you catch exceptions to handle them in your code.

You can declare one or more catch blocks for each try block. There can be no code between the end of the try block and the beginning of the catch block.

Inside a try block, if your code generates an exception, processing stops immediately, and the runtime checks each subsequent catch block to match the exception type declared with the exception type thrown. Upon finding a match, it enters the catch block and executes whatever handling code is in there. Like this:

```
void someMethod() {

   throw new UnknownHostException();

   System.out.println("Uh-oh!");

   //this line will never be printed.

} catch (UnknownHostException uhe) {

     System.out.println("I will be printed"); //ok

}
```

Code inside a catch block may throw exceptions. Like this:

```
void someMethod(){

try {

    //connect to a database here

    //a SQLException happens. Oh no.


    throw new SQLException("Things are not

              what they seem");

} catch (SQLException sqle) {

    //code here to tell the user about the problem

    //with their query


    //Since there was a problem with the database,

    //let's shut down the connection to it.

    //But guess what—closing the connection throws a
    //SQLException too!

    //so we have to do it again...

    try {

        connection.close();

    } catch (SQLException se) {

        //handle the potential

        //connection.close() exception

    } // end inner catch

} // end outer catch

} // end method
```

Code inside a catch block can be well-employed to clean up any resources such as database connections or file streams that should be closed if an exception is thrown. Consequently, catch blocks can also contain try/catch blocks.

If you define a try/catch block, and none of the code in the try block generates an exception, the code inside all catch blocks is completely ignored.

You can rethrow an exception after you catch it. The following is okay:

```
try {

    throw new KittyException();

} catch (KittyException ke) {

    throw new DogException("My dog message. " +

ke.getMessage());

}
```

Using Finally

The finally keyword is used after a try/catch block to indicate that the code inside the finally block should be executed whether the code inside the try block generates an exception or not. No matter what happens (unless someone pulls the plug on your box or externally kills your JVM process), the code in your finally block will run.

Do it like this:

```
try {

...

} catch (SomeEx se) {

 ...

} finally {

 //put the code you want to run to matter what here.

}
```

Note that although the preceding code is how you will see it used 99% of the time, you can use the finally block *without* a catch block.

The finally statement is very useful, especially for doing complex operations involving files, a database, or a network connection, because there can be a lot to clean up if anything goes wrong (or even if it goes right!).

However, be careful of one subtle use with the use of finally. If your code throws an exception *other* than the one you are catching, and enters a finally block, that original exception will be totally lost if you throw a new, different exception inside your finally block. That is a mouthful. Or a mindful. Or whatever. Look at this:

```
InputStream in;

try {

  //do something with in

} catch (IOException ioe) {

  JOptionPane.showMessageDialog(null, "Error! " +

ioe.getMessage());

} finally {

     try {

        in.close(); //could throw

     }catch (IOException ioe) {

        // this hides the original exception

        }

`
}
```

Note that the finally clause will not execute if the catch block calls System.exit();.

So a finally clause is a good thing, and can help keep you from duplicating code.

Different Ways of Handling Exceptions

Let's look at a short demo class that allows us to see everything in one place. This example shows different ways of working with code that could generate exceptions. You might not be familiar with working with Java's networking libraries or database libraries. It doesn't matter here. What matters is noticing the different ways that exceptions can be used in your code.

```
package net.javagarage.demo.exceptions;


import java.io.IOException;

import java.io.InputStream;

import java.net.MalformedURLException;

import java.net.URL;

import java.sql.*;

/**

 * This class demonstrates different ways to write

 * your methods when you need to deal with exceptions

 * in them. Any one of them could be most appropriate,

 * depending on the situation.

 *

 * @author eben hewitt

 */

public class DemoExceptionDeclarations {


  //ONE THROWS. this method does an network

  //operation, but doesn't want to deal with it.

  //so pass the buck...

public static InputStream getDocumentAsInputStream(URL url)

throws IOException {

InputStream in = url.openStream();

return in;

   }

  //NO THROWS, MULTIPLE CATCH. doesn't throw

  //anything. it deals with the exception itself.

  //(This method must be the first born.)

  private void myNetMethod() {

  URL url = null;
  try {
//if this throws an exception, we jump down to
//MalformedURLException
  url = new URL("http://javagarage.net:80/index.html");

  //if this throws an exception, say, because of
  //network problems, or because there is no resource
  //at the URL, then processing jumps to the IOException
  url.openStream();

  } catch (MalformedURLException mue) {
  //getMessage() tells you details about what happened
System.err.println("MalformedURLException: " +
mue.getMessage());
//prints the stack trace to the standard error stream
mue.printStackTrace();
  } catch (IOException ioe) {
  System.err.println("Could not open stream to
             URL " + url);
  }
  }

  //CATCH NESTED IN FINALLY. this method does
  //operations that make her catch
  //multiple exceptions. that's okay, just add
  //multiple catch blocks. Make sure that your catch
  //blocks go from narrowest to widest
public void databaseSelect(int idIn) {
ResultSet rs = null;
Connection connection = null;//get connection

try {
PreparedStatement getStuff;
String s = "SELECT itemName, price FROM Products
            WHERE id = ?";
getStuff = connection.prepareStatement(s);
getStuff.setInt(1, idIn);
rs = getStuff.executeQuery();
//do something with result set here.

//if the above code generates an exception of type
//SQLException, this catch block runs
} catch (SQLException sqle){
System.err.println("Could not execute query:\n" +
sqle.getMessage());

//NESTED TRY/CATCH
try {
rs.close();
} catch(SQLException se){
System.err.println("Could not close result set.\n" +
se.getCause());
}

//if the code above in the first try block of the
//method generates any other type of exception,
//this will catch it, since Exception is the parent
} catch (Exception e){
e.printStackTrace();

//whether or not any exceptions were thrown above,
//this code will execute no matter what
} finally {
try {
//clean up after ourselves—but this also throws
//SQLException, so we will just deal with it here.
connection.close();
} catch(SQLException se){
System.err.println("Could not close connection to DB.\n" +
se.getCause());
}
}
}
}
```

Each method in the preceding class shows a different way of dealing with exceptions, and should serve as a reasonable guide as you write your own exception handling code.

Wrapping an Exception

Well-designed applications are often designed in tiers. Perhaps you have a data tier, and all of the objects of a similar type in a similar package are dedicated to accessing the database, and returning the result of the operation up to another tier of the application.

The purpose of the tiers is to separate different functions. N-Tier development dictates that you don't mix your user interface code with your workflow code or your data access code. It stands to reason that you do not mix an exception generated in the data tier with client tier code.

But you *do* need to handle the exception. So it's a predicament. Well, the thing to do is wrap up your SQLException object in an exception object that hangs out with the client tier crowd.

Let's take a look at an example of a custom exception wrapper for doing this very kind of thing.

Toolkit: A Custom Exception

ApplicationException.java

```
package net.javagarage.demo.exceptions;

/**<p>

 * This class is a subclass of java.lang.Exception,

 * and can be reused to represent any application-

 * specific exceptional state. For instance, if you

 * are using a database, it is not a good idea to

 * allow database exceptions to bubble up to the

 * user, or to require handling by more than one

 * layer in your application.

 * <p>

 * For these reasons, you typically want to make your

 * own exception wrapper that will serve to represent

 * application-level exceptional states. For example,

 * an online store might have a CartException that

 * gets thrown explicitly if the user tries to check

 * out with no items in their cart, or tries to

 * specify a quantity of less than 0.

 * </p>

 * <p>

 * Note that it is good practice to actually extend

 * the functionality of a class that you extend. That

 * is, you should add a method or two that is

 * specific to your situation.

 *

 * @author eben hewitt

 * @see java.lang.Throwable

 * @see java.lang.Exception

 **/

public class ApplicationException extends Exception {

private int status;
```

```
/**
 * constructor no-args
 */
public ApplicationException() {
super();
}

/**
 * Overloaded constructor Exception wrapper—anything
 * that implements the Throwable interface
 */
public ApplicationException(Throwable t) {
super(t);
}
/**
* Overloaded constructor with message describing the
* exceptional state
*/
public ApplicationException(String message) {
super("ApplicationException: " + message);
}

/**
 * Overloaded constructor with message and
 * wrapped root cause
 */
public ApplicationException(String message,
       Throwable cause) {
super(message);
initCause(cause);
}

/**
 * Overloaded Constructor with int message if your
 * underlying layer (such as a database) returns some
 * meaningful integer status such as -1.
 */
```

```
public ApplicationException(String message,
      int status) {
this(message);
this.status = status;
}

/**
 * Sets the status message
 */
public void setStatus(int status) {
this.status = status;
}

/**
 * Gets the status int.
 * @return int The status number
 * representing some state
 */
public int getStatus() {
return status;
}

/**
 * Gets the message from the root cause
 * @return String The root cause message
 */
public String getCauseMessage() {
if(getCause() != null) {
return getCause().getMessage();
} else {
return "Cause unknown";
}
}

}
```

This custom exception features a number of overloaded constructors. It is nice to provide a few overloaded constructors, because it gives the user of your API choices about how to interact with it.

Now that we've seen how to define a custom exception, let's look at how to use one. Check out ExceptionClient.java. His job is to run some code that will cause the custom exception to be thrown.

ExceptionClient.java

```
package net.javagarage.demo.exceptions;

/**<p>

 * ExceptionClient is simply for testing. It is a

 * main method that calls two static methods to

 * demonstrate how to use custom exceptions. The

 * custom exception we use is ApplicationException.

 * </p>

 * @author eben hewitt

 * @see

 * net.javagarage.demo.exceptions.ApplicationException

 **/

public class ExceptionClient {

/**

 * In your business method, just throw exception

 * wrappers. That way, you can keep your code clean,

 * flexible, and easy to maintain.

 * @throws ApplicationException

 */

private static void testWrapper() throws ApplicationException {

    try {

        //do something that will generate an exception

        //you don't want to proliferate

        int[] nums = {1,2,3};

        int x = nums[5]; // ERROR! out of bounds!

    } catch(ArrayIndexOutOfBoundsException obe){

        throw new ApplicationException(obe);

    }

}


/**

 * Test explicitly throwing an exception when

 * something bad happens. Say we have some business
 * rule where you aren't allowed to take more than 10

 * consecutive holidays. We want to generate an app

 * exception if someone tried to.

 */private static void testExplicit(int x)

throws ApplicationException {

    System.out.println("You are asking for " + x +

    " days off.");

if (x > 10){

    throw new ApplicationException(x +

    " is TOO MANY DAYS!\nYou will be beaten.");

}

//only prints if the exception is not thrown.

//if it is thrown, execution jumps out of the method

//and up to where it is caught in main()

System.out.println("Have a nice time!");

}


/**

 * Shows two different uses of your custom exception.

 * Obviously, we cannot test both of these at once,

 * because once an exception is printed,

 * execution stops.

 */

public static void main(String[] args) {

//to demonstrate usage

try {

//1. test wrapping API exceptions

//testWrapper();


/*

 * calling (1) produced this output:

 * Message: java.lang.ArrayIndexOutOfBoundsException: 5

 */


//2. IF user does something bad, you want

//to send customized notice of it

testExplicit(5);//okay


testExplicit(100);//EXCEPTION!
} catch(ApplicationException ae){

System.out.println("Message: " + ae.getMessage());

}

}

}
```

Here is the output result of running ExceptionClient.java as is:

```
You are asking for 5 days off.

Have a nice time!

You are asking for 100 days off.

Message: ApplicationException: 100 is TOO MANY DAYS!

You will be beaten.
```

Looking at the code, you'll notice that there is a second method defined, called testWrapper(), that catches an ArrayIndexOutOfBoundsException and simply takes that entire exception and passes it as the constructor of our custom exception. We can do that because ArrayIndexOutOfBoundsException extends IndexOutOfBoundsException, which extends RuntimeException, which extends Throwable. Whew.

PREV < Day Day Up > NEXT

Exceptions and Inheritance

Look. Let's be honest. I don't like this anymore than you. I just want to go go go. Maybe get a truck and drive. Out on the open road. I look out at the pool and it looks fantastic. I know that it's ice cold. I know it is a burning freeze of water. But it looks so serene and inviting. Maybe I could like being a giant block of ice. Drop in and freeze and then bob up and down, forever, like a cat dangling its tail on a fence, lazy, slow, back and forth. This clock wouldn't tick anymore. In a meaningful way, for me, anyway.

To:=?big5?q?JavaGarage=A4=A4=A4=E5=A5=CE=?=<java-mid.ea-@lists.javagarage.dude>

Subject: excpetion inheritance

From: "Shoeyong" <shoeyong@sonuvagun.com>

Date: Wed, 15 Dec 2003 22:47:49 +0800

Message-id: <002941c18965$4cdf7666$0100a8c0@uyong>

Old-return-path: shoeyong@sonuvagun.com

Sender: Totally Fokd <sauroman@evilempire.org>

I have come to an impasse in my evil plan to rule the world. I need to know about inheritance and exceptions.

OK. What's the name of your box?

"Boomdeay". What difference does that make?

None. I just like to know the names of guy's boxes. What is el problem?

How does it work, this thing you call method overriding and the exceptions methods throw?

Are you on something dude? An overriding method cannot be declared to throw checked exceptions EXCEPT those that are declared in the superclass method OR subclasses of those checked exceptions.

Example pls?

Say you have a super class that defines

```
void myMethod() throws FileNotFoundException...
```

Then all of these alternatives in the subclass are ok:

```
void myMethod() throws FileNotFoundException
```

the following are all NOT ok:

void myMethod() throws PsychoticBreakException //not ok: this is not in the IOException hierarchy

void myMethod() throws FileNotFoundException, EOFException //not ok: the overriding method cannot throw methods that aren't thrown by the overidden method

void myMethod() throws IOException //not ok! IOException is higher up the inheritance hierarchy than FileNotFoundException, which the superclass throws

Explain it to me. This is very good…

IOException is the mama exception up the hierarchy tree of exceptions having to do with input/output ops. EOFException (End of File), and FileNotFoundException, for example, extend IOException. FileOutputStream constructors throw FileNotFoundException.

Let's say it again in a slightly different way, since it can be confusing at first.

So say you have this method "someMethod" in superclass "ClassA".

someMethod() throws IOException.

"ClassB" extends ClassA.

Now you override someMethod() in ClassB. What can it do? What can it not do? What must it do?

It can declare that it throws IOException, just like the superclass method.

It CANNOT throw *more* exceptions like this: someMethod() throws IOException, PsychoticBreakException

It CAN throw *fewer* exceptions, like this: someMethod() { } //no 'throws'.

Step right up and test your deductive reasoning skills!

True or False: If a superclass method throws no exceptions at all, an overriding method in a subclass may not throw any exceptions either.

True!

Now if you are familiar with C#, exceptions are different in Java. You are not forced in C# to handle exceptions that you aren't interested in. Java forces you to do that. So it adds a little more code, but it keeps your API exposed and documented as it should be, and makes your program more easily readable.

Cheers,

TF

This message was re-posted from java-chinese-big5@lists.javagarage.dude and converted from gb231 to big5 by an automatic gateway. Ha ha ha ha.

Exception Handling Pretty Good Practices

This is a list of things that are a good idea to do when writing code in the real world (he writes, as if there is any other world). I don't know if they are *the best* practices ever. That's pretty hard to determine concretely.

- **Do not squash exceptions—do something useful with them**. It is common to see code that catches an exception, and then does nothing. The following will compile:

  ```
  ...catch(Exception e) {}.
  ```

  Usually it isn't quite that angry, and you see a call to System.out.println(). If your application is a console application, this might be appropriate, but would it be better to save that message to a log file along with a timestamp, class name, and username? You bet. If you are writing a desktop application using the Java Swing API, you could still do that, but in addition, trying passing the exception.get-Message() or exception.getCause() to a JOptionPane.showMessageDialog(). It takes two parameters: the parent component (it can be null) and the String message to display. This is like a MsgBox in VB or a JavaScript alert.

- **Catch exceptions with the narrowest type match practical**. That is, design your program keeping in mind that one person's program is another person's API. If it matters to make a distinction between FileNotFoundException and IOException, make that distinction and handle them differently in different catch blocks. More specific is usually better. It's more complete and explicit, and easy to understand.

- **Don't proliferate checked exceptions into your API**. That means be careful if you find yourself writing loads of methods that throw everything. The reason is this: It makes your API contract more brittle, and you might find your client code becoming overburdened if the client can't do something meaningful with the exception. You don't have checked exceptions in languages like C++ and C#, so programmers coming from those languages may find it a bit of a hassle.

- **Don't throw an exception when you can perform a test**. They are meant for exceptional situations, and in all likelihood will cause your code to execute more slowly. A lot of times you can perform a test that saves you from the trouble, time, and reduction in readability that adding exception handling code causes. For example, say you have a form allowing a user to log in. If the password is left blank, you don't need to throw an exception. This isn't really exceptional. Users do dumb things all the time, kind of at a breath-taking pace actually; it's like there's a race with a big important prize if you can be the dumbest user. But if you can do if (password != null)…that's way better (simpler, clearer, faster) than throwing an exception about that little oversight.

- **Reasonably organize your try blocks with related statements**. That means that you shouldn't make a separate try/catch block for each item that throws an exception. Many situations do not call for atomic transactions. Typically, you don't just do one thing in a vacuum, and then do something else totally unrelated to what you just did. You open the file because you want to write to it. You build the URL because you want to connect to it. And then you probably want to read the stream, and do something with that. Sheesh, there's a lot of stuff to do in the world. These things go together. If you didn't build the URL, you don't have much chance of opening that connection, do you? It is okay (good) to put these statements together in *one* try block, and then deal with whatever problem arises in a catch or two. The lesson: do not write one try block per one potentially-checked-exception-throwing-method if it makes sense to group them together.

- **Keep your code encapsulated**. As mentioned earlier, you will hopefully design your applications in tiers, your user interface tier separated from your business logic tier, separated from your persistence tier, and so forth. Do not allow exceptions that are specific to the implementation in one tier bubble up to another. That is, do not allow a SQLException up in your client. Because then your client has to deal with it, and it means that it has been thrown up through each of the proceeding tiers. If you change your persistence layer to an XML file or some financial system or something other than a SQL-compliant database, you won't be throwing SQLException anymore, and yet you will have broken the encapsulation that is the very purpose of layering your code in tiers in the first place. To make matters worse, the API is outdated and clogged up with unnecessary code. The solution is easy: Wrap exceptions in code that makes sense for the tier on which it will actually get handled. Do not let this confuse you, though. It is just fine to propagate exceptions, because often a higher layer is the one you want to present the problem to the user. Just do it smart.

And so ends our business with exceptions. I hope that you have enjoyed it as much as I have. See you next time.

PREV < Day Day Up > NEXT

Chapter 22. FILE INPUT/OUTPUT

STUFF WE'LL DO:

- Get ahold of the java.io package
- Learn how to perform useful File and Directory operations
- Read data from files
- Write data to files

Files and Directories

The first thing to note when dealing with files and directories in Java is that both files and directories are objects of type file. Don't look for java.io.Directory, cuz it doesn't exist.

The other confusing thing about working with files in Java is that when you create a new object of type java.io.File, you do not actually create a physical file on the file system. You have only created an object that can be *used* as a file. There are many classes and methods useful for doing this, which we'll look at in a moment, after we get used to dealing with File objects.

Fields

There are a number of fields in the java.io.File class that are useful when working with files and directories:

- static String pathSeparator The system-dependent path-separator character, represented as a string for convenience. It's : on Unix and ; on Windows. This corresponds to System.getProperty("path.separator");.
- static char pathSeparatorChar The system-dependent path-separator as a character.
- static String separator The system-dependent default name-separator character, represented as a string for convenience. This is \ on Windows and / on UNIX.
- static char separatorChar The system-dependent default name-separator character.

The following command gets the location in the file system from where the java command was invoked:

```
String pwd = System.getProperty("user.dir");
```

Creating a Directory

Using the mkdir() method will create a directory for you. You must use the mkdirs() method (with an "s") if you need to create all non-existing ancestor directories automatically.

```
//create dir without ancestor dirs

    boolean isCreated = (new File("directoryName")).mkdir();

    if (!isCreated) {

        // dir not get created

    }
```

Note that this manner of creating a directory does not create any necessary ancestor directories. That is, if you try to specify a directory like /usr/eben/downloads/tomcat/ and the downloads directory does not exist, the mkdir method will not create it for you, even though it is a requirement for creating the tomcat directory. So it does nothing but return false.

The following code shows how to remedy that situation:

```
//create dir and all ancestor dirs

   boolean isCreated = (new File("directoryName")).mkdirs();

   if (!isCreated) {

       //dir not created

}
```

Deleting a Directory

Deleting an *empty* directory is done with the delete() method called on the File object representing the directory you want to delete. It returns a boolean indicating whether the directory could be deleted.

```
//this will delete an empty directory

boolean isDeleted = (new File("directoryName")).delete();

    if (! isDeleted) {

        // could not delete dir

    }
```

If there are files or directories in the directory you want to delete, you must recursively delete each one.

```
/**

 * Deletes a directory recursively. If the directory

 * isn't empty, you have to first delete all

 * subdirectories and files underneath it.

 * <p>

 * Note that this guy takes a file object instead of

 * String because he calls himself (that's the

 * recursive part). It prints out what it is doing
 * as it does it.
 * <p>
 * Just call <code>new File("dirName")).delete();</code>
 * if you know it is empty.
 * @param directory The directory to delete.
 */
public static boolean deleteDir(File directory){
    if (directory.isDirectory()) {
        //get array of strings of contained
        //files and directories
        String[] subdirs = directory.list();

        //loop over entire array
        for (int i=0; i < subdirs.length; i++) {
          //call this same method, but create the
          //instance with the name of the subdir
            File f = new File(directory,
            subdirs[i]);
            deleteDir(f);
            log("Deleted " + f);
        }
    }
}
```

Files

The following class demonstrates several different methods of the java.io.File class. These include getting the size of a file, renaming a file, and moving a file to a different directory. It also gives you some utility methods to do things you might commonly need to do, such as get the file size in bytes.

CommonFileTasks.java

```
package net.javagarage.demo.io.files;

import java.io.File;
import java.io.IOException;
import java.text.DecimalFormat;
import java.text.NumberFormat;

/**<p>
 * This class demonstrates a number of
 * typical things you need to do when working
 * with files.
 * </p>
 * @author eben hewitt
 * @see File, NumberFormat
 **/
public class CommonFileTasks {

private static final String ps = File.separator;

//enum new in 5.0 classname
//the values will be automatically populated in order
//starting with 0 if you don't specify them

public enum Size {

BYTES, KILOBYTES, MEGABYTES;
}
```

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


String aFile = "C:\\mysql\\bin\\mysqld-nt.exe";


log(getFileSize(aFile, Size.BYTES));

log(getFileSize(aFile, Size.KILOBYTES));

log(getFileSize(aFile, Size.MEGABYTES));


String bFile = "C:\\test.txt";


createFile(bFile);

fileExists(bFile);

renameFile(bFile, "C:\\test2.txt");

deleteFile(bFile);

moveFile("C:\\test2.txt", "C:\\backups\\");

}


/**

 * Creates a file if it doesn't already exist.

 * @param fileName Name of the file you want to create

 */

public static void createFile(String fileName){

try{

File file = new File(fileName);


//creates file

boolean success = file.createNewFile();

if (success) {

log("Created new File");

} else {

log("File existed already. Not re-created.");

}

} catch (IOException e) {

log(e.getMessage());

}

}


/**

 * Renames a file. File will not be renamed if a file
 * of the new name already exists in that directory.
 * @param oldFile File to be renamed.
 * @param newFile New name for the file.
 */
public static void renameFile(String oldFile,
        String newFile){
boolean success = new File(oldFile).renameTo(new
File(newFile));
if (success) {
log("File " + oldFile + " renamed to " + newFile);
} else {
log("File " + oldFile + " could not be renamed.");
}
}

/**
 * Moves a file to a different directory. Note that
 * the default behavior is to fail if the directory
 * to which you are moving the file does not already
 * exist. Remember
 * that creating a new File <i>object</i> does not
 * create a new file <i>on disk</i>.
 * <p>
 * So this creates the directory if it doesn't exist.
 * @param fileName Name of the file to be moved.
 * @param destinationDir New location and name
 * for the file.
 */
public static void moveFile(String fileName, String
destinationDir){

File f = new File(fileName);
File dir = new File(destinationDir);
if (! dir.exists()){
//creates any needed directories that don't exist
dir.mkdirs();
}

boolean success = f.renameTo(new File(
new File(destinationDir), f.getName()));
   if (success) {
   log("File " + fileName + " moved to " +
           destinationDir);
   } else {
log("File " + fileName + " could not be moved.");
   }
}

/**
 * Deletes a local file.
 * @param fileName File to delete
 */
public static void deleteFile(String fileName){
boolean success = (new File(fileName)).delete();
if (success) {
log(fileName + " deleted.");
} else{
log(fileName + " could not be deleted.");
}

}

/**
 * Tests whether the file or directory exists already.
 * @param fileName The file to check.
 */
public static void fileExists(String fileName){
boolean exists = (new File(fileName)).exists();
if (exists) {
log("File " + fileName + " exists.");
} else {
log("File " + fileName + " does not exist.");
}
}

/**
 * file.length() returns a long representing the
 * file size in bytes. This makes it a little more
 * convenient to work with
```

```
    convenient to work with.
 */
public static String getFileSize(String
          fileName, Size s){

File file = new File(fileName);

    switch (s) {
    default:
    case BYTES:
       return file.length() + " bytes";
    case KILOBYTES:
       return file.length() / 1024 + " KB";
case MEGABYTES:
    double size = new Double(file.length() /
                           1024).doubleValue() /1024;
//create a display format for the number
//make it have two decimal places
NumberFormat formatter = new DecimalFormat("#.##");
return formatter.format(size) + "MB";
}
    }

private static void log(String msg){
System.out.println("—>" + msg);
}

}
```

The output is like this:

```
—>2248704 bytes

—>2196 KB

—>2.14MB

—>Created new File

—>File C:\test.txt exists.

—>File C:\test.txt renamed to C:\test2.txt

—>C:\test.txt could not be deleted.

—>File C:\test2.txt could not be moved.
```

Useful Methods

The java.io.File class includes a number of methods that make it easy to work with files. For the examples that follow, assume I've got a file on my hard drive at C:\\backups\\test2.txt that we'll use to compare the results of the following methods.

Here they are:

- boolean delete() Deletes a file
- boolean exists() Returns whether the file denoted by this pathname exists
- File getAbsoluteFile() Gets the file name in absolute form: C:\backups\test2.txt
- File getCanonicalFile() Gets the file name in canonical form: C:\backups\test2.txt
- String getParent() Gets the pathname of the parent directory of this file: C:\backups
- boolean isDirectory() Whether the current file is a directory
- boolean isFile() Whether the current file is a file
- boolean isHidden() Whether the current file is marked hidden
- boolean renameTo(File newName) Renames the file on which the method is called to the newName
- boolean setLastModified(long time) Sets the date and time of when this file was last modified
- URI toURI() Makes a URI object of this file: file:/C:/backups/test2.txt
- URL toURL() Makes a URL object of this file: file:/C:/backups/test2.txt

Note that use of some of the preceding methods (in particular toURL() and getCanonical-File() require you to handle the checked exception IOException).

Traversing Directories and Files

This is a class you can easily transfer directly into your projects. It features a method for recursively visiting a directory tree and all of the files in it. There is a method called execute that simply writes out directory names in all caps and indents file names under the directory in which they live. You can easily modify that method to do whatever kind of processing you want (for instance, make it all read only, set the last modified date, whatever). Here it is.

TraverseDirs.java

```
/*
This class lists all of the directories and files
under a specified start directory.
The method "execute" here simply prints out info
about the dir. You could change the execute method
to do something more exciting.
*/
package net.javagarage.demo.io.dir;

import java.io.File;

public class TraverseDirs {

private static final String startDir =
"C:\\eclipse21\\workspace\\dev\\WEBINF\\
src\\net\\javagarage\\demo\\io";

public static void main(String [] arg){
//specify the dir to start in
//and list everything under it
listEachDirAndFile(new File(startDir));
}

  //iterate all files and directories under dir
    public static void listEachDirAndFile(File dir) {
      execute(dir);

      if (dir.isDirectory()) {
        String[] sub = dir.list();
        for (int i=0; i< sub.length; i++) {
          listEachDirAndFile(new File(dir, sub[i]));
        }
    }
  }
  //iterate only directories under dir (recursive call)
  public static void listDirs(File dir) {
    if (dir.isDirectory()) {
      execute(dir);

      String[] subDirs = dir.list();
      for (int i=0; i < subDirs.length; i++) {
        listDirs(new File(dir, subDirs[i]));
      }
    }
  }

  //iterate files in each dir (recursive call)
  public static void listFiles(File dir) {
    if (dir.isDirectory()) {
      String[] sub = dir.list();
      for (int i=0; i < sub.length; i++) {
        listFiles(new File(dir, sub[i]));
      }
    } else {
      execute(dir);
    }

  }

//this is just where you do whatever work
//you want to do with each file or dir
public static void execute(File f){
if (f.isFile()){
System.out.println("\t" + f.getName());
} else {
System.out.println("Dir: " + f.getName().toUpperCase());
}
}
}
```

For me, the output is like this:

```
Dir: IO

Dir: DIR

TraverseDirs.class

TraverseDirs.java

Dir: FILES

CommonDirectoryTasks.java

CommonFileTasks$Size.class

CommonFileTasks.class

CommonFileTasks.java

WriteBytes.java
```

Notice that the CommonFileTasks$Size.class represents the enum Size, and the IO directory is empty.

File IO: Reading and Writing Stream Data

File IO in Java can be very confusing at first. The reason is because there are a number of abstract classes, and a large number of IO classes that implement specific aspects of IO behavior or requirements, which results in the programmer wrapping two or often three different File class constructors in order to get the desired implementation.

The classes we refer to here are in the java.io package.

Input and Output Streams

Java's implementation of input and output (I/O) services is based on streams. Streams are used to read and write byte data. The data may come from a file, a network socket, a remote object, a serialized object, input at the command line, or somewhere else. Streams can be filtered, allowing limited encryption, serialization of object data, compression using the zip or other algorithm, translation, and so on.

InputStream and OutputStream are the two abstract parent classes that other, more specialized stream types subclass to do the actual work of reading and writing byte data.

InputStream

Depending on what the source of your bytes is, you use a different subclass. Dig: FileInputStream gets bytes from a file. Meant for reading raw bytes. For reading character data, try FileReader.

ByteArrayInputStream

Contains an internal buffer that keeps track of the next byte to be supplied.

PipedInputStream

Gets bytes from an object of type PipedOutputStream. This is intended for use in multithreaded programming, where data is read from a PipedInputStream object by one thread, and the data is written to the PipedOutputStream on another thread.

OutputStream

Depending on where you want to write your bytes, you use one of the following implementing subclasses:

FileOutputStream

Writes the bytes directly to a File object or a FileDescriptor. Use this one if you have byte data like images to write. It's not the best choice for character data—for that consider FileWriter. Note that some platforms may allow a file to be written to by only one stream at a time.

ByteArrayOutputStream

Implements an output stream that writes data to an array of bytes, allowing the buffer to automatically grow. Get the data written to this object using toString() or toByteArray().

PipedOutputStream

This is the sender of data in a piped relationship.

Characters and PrintStream

There are two reasons to refrain from using byte input and output streams for reading and writing character data.

The first is that Java uses Unicode to represent every character with 2 bytes. The purpose of doing so was to ease and encourage internationalization and localization with languages whose characters (such as Korean and Chinese) require 2 bytes. Many plain text files from which you'll read data in, however, use only one byte (8 bits) to represent each character in ASCII text.

The second is that when you work with files containing text, you need to be able to read and write line by line; basic stream I/O doesn't have the concept.

However, you can work with character data in streams by using the PrintStream class. PrintStream provides a number of print and println methods that mimic system calls that are familiar to programmers of C and C++. A chief benefit is that it automatically converts characters, which allows you to pass Strings, chars, and other primitives as arguments. Although it may sound foreign at this point, PrintStream is actually the IO implementation that you're most familiar with; System.out is a print stream, wired for standard output. System.err is a PrintStream as well.

An advantage to working with a PrintStream is that it can be set to automatically flush the buffer after a byte array is written, one of the print methods is called, or if a newline character ('\n') is written to the stream. This accounts for the behavior of System.in, which is implemented as an InputStream.

The unusual thing about working with PrintStream is that it does not throw IOException. Instead, a flag is set internally that can be checked using the checkError() method.

Note that when you use a PrintStream, it converts all characters to bytes, using the current platform's underlying character encoding. If you need to write characters instead of bytes, use a PrintWriter.

Readers and Writers

Java provides a separate set of parent classes for doing character I/O, as opposed to byte IO that is implemented with streams.

Reader is an abstract class, very similar to InputStream. You don't use a reader directly, but rather a different subclass, depending on the source of the characters you need to read.

Writer is also an abstract class, very similar to OutputStream, and its implementing subclasses are used for writing out character data.

Character data comes encoded as a certain character set representation. So let's take a quick detour here before getting into Readers and Writers.

Character Encoding

FileReader and FileWriter read and write double-byte characters. But many native file systems are based on single-byte characters. These streams will utilize the default character encoding scheme of the file system.

To get the default character encoding, check the following property:

```
System.getProperty("file.encoding")
```

You can specify an encoding other than the default. To do this, create an InputStreamReader on a FileInputStream (to read) or OutputStreamWriter on a FileOutputStream (to write) and specify the encoding you want to use. Let's do an example of how to read ISO-Latin-1 or UTF-8 Encoded Data.

```
//String enc = "8859_1"; //ISO-Latin-1

String enc = "UTF8"; //UTF 8


try {

BufferedReader in = new BufferedReader(

          new InputStreamReader(new

FileInputStream("myFile"), enc));

      String line = in.readLine();

   } catch (UnsupportedEncodingException e) {

//handle

   } catch (IOException e) {

//handle

   }
```

Note that the constructor for FileInputStream is overloaded to accept a String containing the charset name, or a charset object, or a CharsetDecoder object. After we construct the FileInputStream object, we pass it as an argument to the InputStreamReader, and then pass that to the BufferedReader constructor, and work with the buffered reader.

You can also call the getEncoding() method to determine what charset is being used by this stream.

Charset

A charset is a combination of a defined set of character codes, and a scheme for encoding the characters into the codes. As we have seen, you can use various charsets when working with file input/output.

Every JVM implementation must support the following charsets:

US-ASCII Seven-bit ASCII, the Basic Latin block of the Unicode character set

ISO-8859-1, ISO-LATIN-1

UTF-8 Eight-bit UCS Transformation Format

UTF-16BE Sixteen-bit UCS Transformation Format, big-endian byte order

UTF-16LE Sixteen-bit UCS Transformation Format, little-endian byte order

UTF-16 Sixteen-bit UCS Transformation Format, byte order identified by an optional byte-order mark

Reader

The implementing subclasses of reader that you might use are illuminated in the following sections.

BufferedReader

This guy is perhaps the most popular reader, and for good reason. He's fiendishly handsome, plays a surprisingly smoking clarinet, and isn't afraid to kegstand character data when pressed to. There are other reasons too. To begin with, BufferedReader provides access to each individual line in a file. The other readers don't. So if you're reading data from a log file, a csv file, or other such source, this is probably your best choice.

Also impressive is that the BufferedReader is buffered. That means that it provides very efficient reading of characters, arrays, and lines. You can specify the buffer size you prefer if you don't like the default, which is usually large enough.

Here is a typical use of BufferedReader:

```
BufferedReader input

   = new BufferedReader(new FileReader("myInputFile.txt "));
```

Here's what's happening. You construct a FileReader object, passing it the String name of the file you want to read. You pass this newly minted FileReader object to the BufferedReader constructor, and call it "input." You can then read the data in one of three ways.

Call input.readLine() to access a single line of character data. A line is considered to be terminated by either a line feed (\n), a carriage return (\r), or a carriage return followed immediately by a line feed. This method will return null after it reaches the end of the stream. So you can read every line of a file like this:

```
String line = "";

while ((line = in.readLine()) != null)

    //do something with line

}

//eof
```

The loop looks a little busy because we're doing two things: checking if we reached the end of the file, and then assigning the current line of character data to the "line" string variable so that we can do something with it inside the loop. That line will be set to null when the end of the file is reached.

Call input.read() to read a single character.

Call input.read(char[] cbuf, int off, int len) to read characters into a portion of an array. This method repeatedly invokes the read() method of the underlying stream until one of the following happens:

1. The number of characters specified in the len parameter have been read.
2. The underlying read() method returns −1, which indicates the end of file has been reached.
3. The ready() method returns false, indicating any further calls to read() would block.

CharArrayReader and StringReader

These classes are useful when the character data source is an array or String.

FileReader

Useful when the character data source is a File. Wrap it in a BufferedReader for freshest taste.

InputStreamReader

Use this class when the character source is an InputStream. It allows any input stream type to be used with characters instead of bytes. Here's the basic recipe for reading lines of characters from an input stream:

1. Create your input stream type based on the byte source.
2. Create InputStreamReader object using that input source.
3. Create BufferedReader object from that input stream reader.
4. Call readLine() method. Returns a non-terminated String.
5. Process yer data.
6. Repeat until readLine() returns null.
7. Take the day off and go to a yachting festival.

Writer

Writer subclasses work the same way as reader subclasses do.

BufferedWriter

Efficiently writes character text to an output stream using a buffer. The BufferedWriter provides a newLine() method, which uses the system-dependent character for a new line as defined by the system property line separator.

Because many Writers send their data directly to the underlying stream, it is a good idea to wrap a BufferedWriter class around a FileWriter or OutputStreamWriter, much as we did with BufferedReader.

```
try {
        BufferedWriter out = new BufferedWriter(new

FileWriter("outFile.txt"));
        out.write("some data");
        out.close();
    } catch (IOException e) {
       //handle
    }
```

In the preceding example, the file outFile.txt will be created if it does not already exist.

If you want to append character data to the end of a file that already exists, pass a Boolean true as a second argument to the FileWriter constructor, as in the following example:

```
BufferedWriter out = new BufferedWriter(new

FileWriter("outFile.txt", true));
```

Note that you can call the flush() method to flush the buffer should it be necessary to manually do so.

OutputStreamWriter

This class serves as a bridge between character and byte stream classes. Its purpose is to encode the characters written to this stream as bytes of the specified charset.

Each time you call the write() method, the charset encoder is invoked on the given characters, and are sent to a buffer. The data is then written to the underlying output stream.

PipedWriter

This class is the companion to PipedReader, mentioned previously. A typical constructor specifies the name of the PipedReader to synch to.

Note that while there is a no-arg constructor for this class, it must be connected to a PipedReader before it can be used. Do so using the connect(PipedReader pr) method. If the PipedWriter is already connected to a PipedReader, an IOException is thrown.

Let's look at some code that puts this to work.

Reading Text from a File

Here is a class that you can likely use with some frequency. It reads in a text file line by line and calls a processing method as it does so. For this example, the processing method simply sends all of the characters to uppercase and prints it to the console.

ReadingTextFromFile.java

```
package net.javagarage.demo.io.files;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

/**<p>
 * This code reads in a file line by line
 * with BufferedReader and processes each
 * line.
 * @author Eben Hewitt
 **/
public class ReadingTextFromFile {

public static void main(String[] args) {
read("C:\\readtest.txt");
}

//convenience method to encapsulate
//the work of reading data.
public static void read(String fileName) {
    try {
        BufferedReader in = new BufferedReader(
        new FileReader(fileName));
        String line;
        int count = 0;
        while ((line = in.readLine()) != null) {
          //our own method to do something
          handle(line);
          //count each line
          count++;
```

```
            }
            in.close();

            //show total at the end of file
            log("Lines read: " + count);

        } catch (IOException e) {
            log(e.getMessage());
        }
    }

    //does the work on every line as it is
    //read in by our read method
    private static void handle(String line){
        //just send characters to upper case
        //and print to System.out
        log(line.toUpperCase());
    }

    //convenience to save typing, keep focus
    private static void log(String msg){
        System.out.println("—>" + msg);
    }
    }
```

The output from the file is a result that has all characters in uppercase text, a ? string pointing to every line—whether is has data on it or not—and then the total number of lines read in, as in the following:

```
//blah blah blah
—>EXECUTING ACTIONS
—>
—>EXECUTING ACTION: INSTALLDRVRMGR
—> ARG1: '(NULL)'
—> ARG2: '(NULL)'
—> ARGS: ''
—>INSTALL DRIVER MANAGER SUCCEEDED
```

```
—> RETURN HR: 0X0

—>

—>Lines read: 31
```

Writing Text to a File

This class writes character data to a file. If the file does not exist, it is created. If it does exist and has data in it, any new data is appended to the end of the file.

WritingTextToFile.java

```
package net.javagarage.demo.io.files;


import java.io.BufferedWriter;

import java.io.FileWriter;

import java.io.IOException;

import java.io.UnsupportedEncodingException;

import java.util.Date;


/**<p>

 * This code reads in a file line by line

 * with BufferedReader and processes each

 * line.

 * @author Eben Hewitt

 **/

public class WritingTextToFile {


public static void main(String[] args) {

write("C:\\writetest.log");

log("All done.");

}


//convenience method to encapsulate

//the work of reading data.

public static void write(String fileName) {

try {

BufferedWriter out = new BufferedWriter(
new FileWriter(fileName, true));
out.write("GMT Date:");
out.newLine();
out.write(new Date().toGMTString());

out.close();

} catch (UnsupportedEncodingException uee) {
log(uee.getMessage());
} catch (IOException ioe) {
log(ioe.getMessage());
}
}

//convenience to save typing, keep focus
private static void log(String msg){
System.out.println("—>" + msg);
}
}
```

The result simply prints

```
—> All done.
```

And when you open the written file, it contains data similar to the following:

```
GMT Date:
27 Mar 2004 23:27:16 GMT
```

Reading and Writing Image Data

To find out how to read and write Image data, you can do two things. Check out the following classes in the 1.5.0 API.

The second thing is to look in the example code I've written in the toolbox section of the garage. The GaragePad app lets you draw freehand onto the canvas and then save the pixel data into an image.

Okay. That sounds kind of lame. So here is a sneak preview:

```
Image image = canvas.createImage(rect.width, rect.height);

Graphics g = image.getGraphics();

canvas.paint(g);

ImageIO.write((RenderedImage)image, "jpg", outFile);
```

This code uses the write method of the javax.imageio.ImageIO class to write out a new file with JPEG encoding, and it will be called whatever the value of outFile is. To do so, it creates a java.awt.Image object from a java.awt.Canvas object, which captures the drawn pixels.

Chapter 23. FRIDGE: GUACAMOLE

I think it was pretty hard, doing all of that work in the last topic. Where I come from, programmers can get hungry just like anybody else. And when that happens, I think there's nothing better than a relaxing break, with a cool side of fresh guacamole dip. Don't you agree?

Now. I'm going to divulge here my very own recipe for making guacamole. In my view, this one page right here is worth the entire price of the book. That's really what I think. I think you'll like it that much. Please try it out and tell me what you think. This recipe serves four people. Two, if you live in my family.

Most importantly, you have to choose good avocados. This can be fun. First, you don't want the green ones; those are not ripe enough. You also don't want the totally black ones that mush a little when you pick them up, because those are overripe and will have undesirable black dots inside. Choose avocados that are a mix of green and black on the outside, and that feel firm but also give a little when you squeeze them.

Now here we go:

Get a ceramic mixing bowl and empty into it the fruit of four avocados.

Mash the avocado with a big masher.

Cut a lime in quarters. Squeeze the juice of the lime wedges onto the avocado.

Take two tomatoes, still on the vine, and dice them. Make the sections large, though, like the size of a dime. Drop those into the bowl.

Get a big white onion and cut it in half. Save one half for something else later; we don't need it. Cut the other half into dime-sized pieces just like the tomato. Not lame little tiny pieces. Good sized pieces. I might call them chunks.

Generously add fresh ground salt and pepper. Stir all of this together.

Here is the killer. Dice a chile pepper. One of those long, thin dark green ones. They're called serranos. If you can't find that kind in your store, you might want to consider moving to the desert like me. We have both Java and serranos here.

Stir all this up together and serve with tortilla chips. It should be fluffy, fresh, cool, and perfectly delicious.

It's a meal in itself. It takes about 35 minutes to make. This is one of my favorite things in the world, probably.

Chapter 24. USING REGULAR EXPRESSIONS

DO OR DIE:

- Straight Cs
- Blow this taco stand the minute I graduate
- Stay sweet

Regular expressions are fiendish things. However, sometimes you need 'em. Luckily, the Java API introduced in 1.4 made working with them a good deal easier. In this topic, we will do a brief overview of the regex for the uninitiated (thank you sir; may I have another). Next, we'll see the facility that Java makes available for using them. Then we'll use them. Then we'll be done so we'll stop.

Purpose of Regex

If you're already Mr. Big Deal Regex Guy[1] and just want to know how Java does it, skip this section. Else, continue.

[1] I would like to take this opportunity to acknowledge our female readers within the programming community. Please note that throughout this book, for each reference to "x guy", where x refers to some technology, let "guy" refer to a member of either gender. I hope that this is accepted usage here, as such use colloquially seems generally accepted. For example, "Hey, are you guys going to lunch with us?" spoken cheerfully to one's male and female coworkers, or, "Uh, when did you guys get back from Reno—I thought you weren't gunna be home til Sunday," when referring directly to one's own parents.

A regular expression is a description of a textual pattern that enables string matching. You create a string of characters with special meaning and compile them into a regular expression pattern, and then use that pattern to find strings that match it.

The most common example of a regex type pattern is *, where the asterisk matches everything. In SQL, you might type %ant and because the % in SQL matches zero or more of any character, your expression will match "Fancy pants", "supplant", and so forth.

See Table 24-1 for a reference of metacharacter shortcuts you can use as you write regular expressions.

Regex Metacharacters

If You Want...	Use
Any digit 0 – 9	\d
Any non-digit	\D
Letters, numbers, and underscores	\w
Any character that isn't a letter, number, or underscore	\W
A whitespace character	\s
Any non-whitespace character	\S
Any one character	-
Allow nothing before this character	^
Allow nothing after this character	$
Zero or one character	?
Zero or more characters	*
One or more characters	+

The metacharacters are those that stand in for character types or are a placeholder for the number of characters they specify. You use metacharacters in combination with regular character sequences. Although that sounds straightforward enough, it can be tricky to use them in meaningful combinations of any complexity with characters.

Maybe you feel comfortable with regex now after that short introduction. That's really about all there is to metacharacters. A moment to learn, way too long to master.

As the great blues artist Willie Dixon sang, "I ain't supestitious, but a black cat crossed my trail...." So just in case you *do* have some reservations, let's take a look at some examples in Table 24-2.

Matching Regular Expressions with Character Sequences

This Regex	Matches These Strings
A*B	B,A,AAB, AAAB, etc.
A+B	AB,AAB,AAAAB, etc.
A?B	B or AB only
[XYZ]C	XC,YC,ZC only
[A-C]B	AB,BB or CB only

[2-4]D	2D,3D,4D only
Deal\s\d	Deal 8, Deal 999, but not Deal3 or DealA
(X|Z)Y	XY or ZY
(cat\s) { 2}	cat cat
(cat\s){ 1-3}	cat, cat cat, or cat cat cat

I hate saying this, but regular expressions are the subject of a number of complete books and it's probably best if I refer you to one of them at this point instead of going further with them outside the scope of their use in Java. Anyway, the preceding tables and the following sample code really should cover most of the situations you will need at first. Before you do anything crazy, check the API documentation for the java.util.Pattern class. It features a huge number of commonly used regular expression patterns, and if you're stuck, that might help you out. Which would be great. I have a really cool car.

Regex in Java

Ah, regex in Java. It's no big thing. Although Java can't do much to make our time-honored regular expressions any less obscure, they can make them easier to work with.

The relevant package here is java.util.regex. It contains two classes: Pattern and Matcher. Here's how it works:

1. Create a String containing your regular expression.
2. Compile that String into an instance of the Pattern class. You do this using the static `Pattern.compile(String regex)` method.
3. Create an instance of the Matcher class to match arbitrary character sequences against your pattern.

Those steps look like the following in code:

```
Pattern pattern = Pattern.compile("hi\s {1-3}");

Matcher matcher = p.matcher("hi hi hi");

boolean isMatch = m.matches(); //true
```

You can also do all of this in one step (very cool).

```
boolean isMatch = Pattern.matches("hi\s {1-3}", "hi hi hi");
```

It is easy to use the metacharacters in your regular expressions to determine if a String matches or the regex not. For example, say you need to do some validation on data entered by users in your application. You need to make sure that a U.S. zip code is entered, which means five digits. You can do it as follows:

```
/**
 * Returns whether or not the passed number is
 * exactly five digits, which the requirements state
 * the customer number is.
 * Matches: 01234 and 85558<br>
 * Does not match: three or 123ert or 999999 or 876
 * @param numToCheck the number you want to check.
 * @return true if the passed number is 5 digits,
 * false otherwise
 */
private boolean validateCustomerNumber(String
        numToCheck){
  return Pattern.matches("^\\d{5}$", numToCheck);
}
```

Pattern and Matcher

As mentioned, there are only two classes in the java.util.regex package.

A Pattern object is a compiled instance of a String representing a regex. The Pattern object can be used to create a Matcher object to match character sequences.

A Matcher object interprets patterns to match them with character sequences. You can create a Matcher directly or through invoking the Pattern class's `matcher()` method. There are three ways to match a pattern using Matcher.

1. The `matches()` method tries to match the complete input sequence against the pattern.
2. The `lookingAt()` methods tries to match the input sequence against the pattern, starting at the beginning.
3. The `find()` method scans the input sequence looking for the next subsequence that matches the pattern.

Regular expressions are commonly used to search through vast reams of digital text. They can also be used to check if a string passed in from a user matches a certain pattern. A good example of this is when you ask users to choose a password, and require that it be six characters and contain at least one number. User validation is another good example—we use it commonly to make sure that an e-mail address entered by a user is well formed.

However, there is another common use for regular expressions, and that is replacing text that matches a certain pattern.

Let's look at a complete example now that demonstrates use of both features: matching strings (and returning true if they match and false if they don't) and replacing character sequences found with different character sequences of our choice.

RegexDemo.java

```
package net.javagarage.demo.regex;

import java.util.regex.*;

/**

 * Demonstrates how to use the Java regular expressions

 * classes. You should be able to use

 * this code right out of the box, as it also serves

 * as a useful library that checks all of these things:

 * <ul>

 * <li>URL

 * <li>email address

 * <li>phone number

 * <li>dates

 * <li>numbers

 * <li>letters

 * <li>tags, such as <body>

 * <li>general characters, such as new lines,

 * whitespace, etc.

 * </ul>

 * <p>

 * All of these regular expressions must be escaped

 * because we are using them in Java. That means that

 * a regex that looks like this: <br>

 * <code>\s+</code>

 * <br>

 * we must write like this:<br>

 * <code>\\s+</code>

 * <br>

 * ...with all of those \ characters escaped.

 * <p>

 * By default, pattern matching is greedy, which

 * means that the matcher returns the

 * longest match possible.

 * <p>

 * These regexes should be just fine for general

 * business app use, but don't put this code in your

 * nuclear reactor or space shuttle (duh).

 *

 * @author eben hewitt
 * @see java.util.regex.Pattern,
java.util.regex.Matcher
 **/
public class RegexDemo {

//Constants for regular expressions for those
actions:

//Constant matches any number 0 through 9.
public static final String NUMERIC_EXP = "[0-9]";

//Constant matches any letter, regardless of case.
public static final String ALL_LETTERS = "[a-zA-Z]";

//POSIX for all letters and numbers. Works only for
        US-ASCII.
public static final String ALPHANUMERIC = "\\p{Alnum}";

//Useful for stripping tags out of HTML
public static final String TAG_EXP = "(<[^>]+>)";

//matches an email address
public static final String EMAIL_ADDRESS_EXP =
"(\\w[-._\\w]*\\w@\\w[-._\\w]*\\w\\.\\w{2,3})";

//matches a URL
public static final String URL_EXP =
"^http\\://[a-zA-Z0-9\\-\\.]+\\.
         [a-zA-Z]{2,3}(/\\S*)?$";

//matches 12/14/2003 and 1/5/03
public static final String DATE_EXP =
"^((0[1-9])|(1[0-2]))\\/([1-9]|(0[1-9])|
         ([1-2][0-9])|3[0-1])\\/[0-9]{1,4}$";

//phone number with area code
public static final String PHONE_EXP =
        "((\\(\\d{3}\\)
?)|(\\d{3}-))?\\d{3}-\\d{4}";

/**
 * Matches a line termination character sequence.
 * In this case, it is a character
 * or character pair from the set:
 * \n, \r,\r\n, \u0085,
 * \u2028, and \u2029.
 */
public static final String LINE_TERMINATION =
"(?m)$^|[\\r\\n]+\\z";

/**
 * Matches more than one whitespace character
 * in a row. For example: "hi there".
 */
public static final String DUPLICATE_WHITESPACE =
        "\\s+";

/**
 * typical characters
 */
public static final String TAB = "[\t]";
public static final String NEW_LINE = "\n";
public static final String CARRIAGE_RETURN = "\r";
public static final String BACKSLASH = "\\";

/**
 * Tells you if a certain String matches a certain
 * regex pattern. Use this method if you want to
 * define your own regex, and not use one provided
 * via the constants.

 * @param s String used as character sequence
 * @param p String regular expression you want to match
 * @return boolean indicates whether the pattern is
 * found in the String.
 * If the pattern is found in the string,
 * returns true.
 */
public static boolean regexMatcher(String input,
          String p) {
CharSequence inputStr = input;
String patternStr = p;
// Compile regular expression
Pattern pattern = Pattern.compile(patternStr);
// Replace all occurrences of pattern in input
Matcher matcher = pattern.matcher(inputStr);
return matcher.find();
}

/**
 * Finds all instances of <code>pattern</code> in
 * <i>string</i>, and replaces each with the
 * <code>replacer</code>. <br>
 * Examples:<ul>
 * <li>Input:regexReplacer("D99D",
 * RegexHelper.ALL_LETTERS, "x")
 * Result: x99x
 * <li>etc
 *
 * @param input String you want to clean up.
 * @param p String defining a regex pattern that you
 * want to match the passed String against.
 * @param replacer String containing the characters
 * you want in the final product
 * in place of the pattern string characters.
 * @return String Containing the same String but with
 * replaced instances.
 */
public static String regexReplacer(String input,
          String p, String replacer) {
    CharSequence inputStr = input;
    String patternStr = p;
    String replacementStr = replacer;
    // Compile regular expression
    Pattern pattern = Pattern.compile(patternStr);
    // Replace all occurrences of pattern in input
    Matcher matcher = pattern.matcher(inputStr);

    return matcher.replaceAll(replacementStr);
}

//shows the results of using the different
//regexes for matching
private static void demoMatchers(){
    print("MATCHING:");

    // This input is not alphanumeric
    print("Is %*& alphanumeric? " +
      regexMatcher("%*&", RegexDemo.ALPHANUMERIC));

    //good email: returns true
    print("Is VALID EMAIL: " +
    regexMatcher("dude@fake.com",
    RegexDemo.EMAIL_ADDRESS_EXP));

    //bad email: returns false
    print("Is VALID EMAIL: " +
    regexMatcher("not @ good",
    RegexDemo.EMAIL_ADDRESS_EXP));

    //good URL: returns true
    print("Is URL: " +
    regexMatcher("http://www.javagarage.net",
    RegexDemo.URL_EXP));

    //bad URL: returns false
    print("Is URL: " +
    regexMatcher("java.com", RegexDemo.URL_EXP));

    //check date 30/12/2003: false
    //no, it isn't localized. you can't have everything
    print("Is VALID DATE: " +
    regexMatcher("30/12/2003", RegexDemo.DATE_EXP));
    //check date
    print("Is VALID DATE: " +
    regexMatcher("5/12/2003", RegexDemo.DATE_EXP));

    //check a phone number
    print("Is VALID PHONE: " +
    regexMatcher("(212)555-1000",
RegexDemo.PHONE_EXP));
}



//shows replacing characters matched
private static void demoReplacers(){
    print("REPLACING:");
    //remove all HTML tags from this code
    //and replace with nothing
    print("Remove Tags: " +
    regexReplacer("<html><body><p>the
             text</p></body></html>",
    RegexDemo.TAG_EXP, ""));

    //remove extra whitespace
    print("Extra whitespace replace: " +
    regexReplacer("far away",
    RegexDemo.DUPLICATE_WHITESPACE, " "));

    //remove and replace new line characters
    print("Remove New Lines: " +
    regexReplacer("1\n2\n3", RegexDemo.NEW_LINE,
    " NEW LINE WAS HERE "));

    //replace tab characters (\t) with pipes (|)
    print("Tab replace: " +
    regexReplacer("Hello\tSweetheart",
               RegexDemo.TAB, "|"));
}

//just to save typing
private static void print(String s){
    System.out.println(s);
}

//run the show
public static void main(String[] a) {

    demoMatchers();
    demoReplacers();
}
}
```

The output of executing RegexDemo.java is

```
MATCHING:
Is %*& alphanumeric? false
Is VALID EMAIL: true
Is VALID EMAIL: false
Is URL: true
Is URL: false
Is VALID DATE: false
Is VALID DATE: false
Is VALID PHONE: true
REPLACING:
Remove Tags: the text
Extra whitespace replace: far away
Remove New Lines: 1 NEW LINE WAS HERE 2 NEW LINE WAS
HERE 3
Tab replace: Hello|Sweetheart
```

As with most of the code in this here *Garage* book, the idea is that the comments are inline, and I want to encourage you to read the code. So I put all the stuff I want to say about RegexDemo.java in the class itself. You can use this code in your environment. It will come in handy maybe.

You can also match and/or replace strings in files (obviously, I think), not just in strings you pass into the static methods of the preceding class. Want to try it? Okay!

Let's say that you've got a file at E:\\old.data, and you want to find all occurrences of the phrase "wood" and replace it with the character sequence "banana". You can do it with the file ReplacePatternFile.

The following is the content of old.data:

```
How much wood could a woodchuck chuck

if a woodchuck could chuck wood.
```

ReplacePatternInFile.java

```
package net.javagarage.demo.regex;


import java.io.*;

import java.util.regex.Matcher;

import java.util.regex.Pattern;


import javax.swing.JOptionPane;


/**

 * <p>

 * Demos how to open an arbitrary text file and

 * replace all occurrences of the given regex pattern

 * with some other character sequence. Neat!

 * <p>

 * And Dude said let John Lee Hooker accompany the

 * writing of this class. And thus it was done.

 * And Dude saw that it was good.

 *

 * @author eben hewitt

 */

public class ReplacePatternInFile {

    BufferedWriter writer;

    BufferedReader reader;


public static void main(String[] args) {

    ReplacePatternInFile work = new ReplacePatternInFile();
    work.replaceFile("E:\\old.data", "E:\\new.data");
    System.out.println("Replaced matches");
}

//this method reads in the file specified by the
first
//param, and writes it out to the file name specified
//by the second param
private void replaceFile(String fileIn,
                String fileOut) {
    File inFile = new File(fileIn);
    File outFile = new File(fileOut);

    try {
    //get an inputstream to read in the desired file
    FileInputStream inStream = new
FileInputStream(inFile);

    //get an output stream so we can write the new file
    FileOutputStream outStream = new
FileOutputStream(outFile);
    //the buffered reader performs efficient reading-in
    //of text from a source containing characters
    reader = new BufferedReader(
    new InputStreamReader(inStream));

    //this will write the new data to our second file
    writer = new BufferedWriter(
    new OutputStreamWriter(outStream));

    //this is the string we want to match for replacing
    Pattern p = Pattern.compile("wood");
    Matcher m = p.matcher("");

    String s = null;
    String result;

    //loop over each line of the original file,
    //and each time we meet an occurrence of p
    //the readLine method
    while ( (s = reader.readLine()) != null ) {
    //reset with a new input sequence
    m.reset(s);

    //replace the matches with this string
    result = m.replaceAll("banana");
    //write the data to the file
    writer.write(result);
    //put a new line character
    writer.newLine();
    }

    //clean up after yourself
    reader.close();
    writer.close();

    } catch (IOException ioe){
          JOptionPane.showMessageDialog(null,
ioe.getMessage());
          System.exit(1);
    }
    }
}
```

After executing the code, we have a new file called new.data. The contents of that file should look like this:

```
How much banana could a bananachuck chuck
if a bananachuck could chuck banana.
```

Well, that's it for this topic of regular expressions. That's enough about it for all I care. I just hope that it's all you care for too. At least for right now. Now I feel uncomfortable. Why since I was in sixth grade I don't know what to do with my hands. Maybe I should take up smoking. Hm. It is possible, after all.

And on that note, let's split.

Chapter 25. CREATING GUIS WITH SWING

DO OR DIE:

- Shake it.
- Please, don't break it.

Let GUI = Graphical User Interface;

Let Swing = JavaProgrammingLanguage.getProgrammingElement(GUI);

Swing.listImportantElements() {

//returns: TopLevelContainer (Frame)

//Intermediate Container (ContentPanel)

//LayoutManager(s)

//Atomic Elements (Components)

Frame.getDefinition() Main window that holds every other component of a Swing application. The frame, which is created as an object of type JFrame, contains controls for iconifying (minimizing) and maximizing the frame, and a little X for closing it.

Frame.getNote() Note that the JFrame is not the only kind of top-level container for Swing apps. JDialog is too, and so is JApplet (which is discussed in its own topic on Applets).

FRIDGE

The back of this book, Chapter 35, features a "Toolkit," which is a load of applications that are pretty good quality that feature popular kinds of functionality, such as displaying scrollable areas, transforming XML and displaying it in a user interface, and connecting to a socket at the click of a menu button. Making GUIs with Java using AWT and Swing is a gigantic, really gigantic topic. The books that cover Swing alone are around 1,400 pages long. This topic tries to show you around the landscape a little, and then points you to the other places in the book where you can see this stuff implemented live. There is simply not room to go into Swing in depth here. But as we look at the landscape, you can get a feel for how things work, and it should be enough to get you started.

Anatomy of a Swing App: Stuff You Typically Need to Do in Swing

Starting the App: InvokeLater()

First, the main method calls a static method from the javax.swing.SwingUtilities class called invokeLater(). This method is suited for starting the application, as its purpose is to update the GUI asynchronously on the event dispatching thread.

You can call this method from any thread if you want the event dispatching thread to run some code. If used in this way, it would be invoked like so:

```
Runnable updateGUI = new Runnable() {

   public void run() {

      new MyGUI();

   }

};

SwingUtilities.invokeLater(updateGUI);
```

Here, we just create a new object of a type called MyGUI, which probably extends JFrame or holds a JFrame member variable, sets up the properties of the frame such as how big it should be and what kinds of components should be in it, and then shows itself.

The invokeLater method causes the Runnable to have its run method called in the dispatch thread of the AWT Event Queue. It won't happen until after all pending events are processed. That thread, the AWT Event Queue, stores events in a queue, and executes them asynchronously and sequentially (in the order stored).

Creating the Frame

Now let's look at how to make a main application window for storing buttons and text fields and other user controls. You do this using a JFrame, which you can give a title, as in the following example:

```
JFrame myFrame = new JFrame("My Application");

//now call methods on myFrame to add controls, set

//the layout, make it visible, and so forth
```

Note that if you call the static method JFrame.setDefaultLookAnd FeelDecorated(true), this must be invoked before you call the JFrame's constructor, or it will have no effect. After you have a frame, there are a few maintenance things you need to set up. First, call frame.setDefaultClose Operation(JFrame.EXIT_ON_CLOSE) to ensure that when you click the X to close the window it will actually close.

Next, you need to call the pack() method, which fits all of the components into the available space according to their preferred size.

Finally, you call setVisible(true) in order to display the frame and its components.

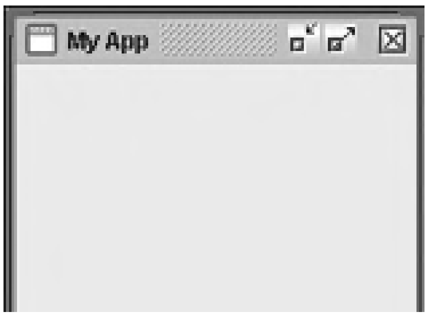
Frame.runExample():

This code demonstrates creating, sizing, and showing an empty frame, similar to the Windows Form you get when you start Visual Studio.

FrameDemo.java

```
package net.javagarage.demo.ui;

/**<p>

 * Very simple frame that does nothing.

 * The purpose is to give you something to

 * start with each time you make a new Swing app.

 * </p>

 * @author Eben Hewitt

 **/

import java.awt.*;

import javax.swing.*;


public class FrameDemo {


     JFrame frame;

    //simply creates the frame, adds an empty

    //label to it so that it has some size,

    //and shows the frame

    private static void launch() {


    /* Change the way the frame looks and feels.

     * decorate it with this command, instead of

     * having the system do it. you must call this

     * method BEFORE creating the JFrame.

     */

    JFrame.setDefaultLookAndFeelDecorated(true);


   //create window, and put a title

   //in top-left corner

   frame = new JFrame("My App");
   //make the app stop when window is closed

frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

   JLabel emptyLabel = new JLabel("");
   emptyLabel.setPreferredSize(new Dimension(200, 200));
   frame.getContentPane().add(emptyLabel,
                           BorderLayout.CENTER);

   //cause the window to be sized in such a way that
   //it snugly fits the preferred size and layouts of
   //the components in it. this is a method of
   //the java.awt.Window class
   frame.pack();
   //show it
   frame.setVisible(true);
  }

public static void main(String...args) {

SwingUtilities.invokeLater(new Runnable() {
   public void run() {
       launch();
   }
   });
}
}
//end FrameDemo.java
```

Figure 25-1. The basic frame will shut down the JVM when closed, and can be resized.



ContentPanel.getDefinition() The frame contains a content panel. The main content pane, which you get a reference to via frame.getRootContent Pane() is the workspace onto which you place buttons, text fields, and other elements with which the user directly interacts.

ContentPanel.getNote() The content panel is also called a pane, like a window pane. In the main, a pane can be plain. It is simply a plane.

There are different kinds of panes, however, including the JScrollPane, which automatically adds scrollbars, and the JTabbedPane. Both of these we make good use of in examples here.

Adding User Controls

Components.listImportantElements() {

//returns:

- JButton A button that shows text and can do something when you click it.
- JLabel An element used generally for holding textual messages.
- JTextField Like an HTML <input>, this component allows the user to enter text.
- JEditorPane This lets the user enter text or display text-based files.
- JScrollPane A container that scrolls. Use it by creating content in a different panel and then adding that panel to the scroll pane.
- JTabbedPane A container that holds tabbed panes.

/*

Let's talk about a few different, important Swing components in turn. They're just objects, so after you are familiar with their general behavior, you can create them, call setX() methods on them to specify their particulars, add an event handler if necessary, and add them to the content pane.

*/

}

User Controls.stepInto()

The components we discuss in this section are commonly used controls that users need in applications. This is by no means an exhaustive look at the controls Swing makes available. It's just what you need to get started. Anyway, after you are used to working with a few of the controls, it is an easy step to read the API for what other kinds of components you get for free.

JLabel

This is an area. Just a plain area, corresponding to a label in Visual Basic. You can put text on it, which is its usual purpose. Or an image, which is probably a good idea to put there. Labels are just regular folk. Create them like this:

```
JLabel label = new JLabel("You must conform.");
```

The text passed to the constructor will show up on the label, and will be left-justified and centered vertically.

To create a label containing an image and text, you can follow this example:

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Container;
import java.awt.Dimension;
import java.awt.Font;

import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

public class LabelTest {
    public static void main(String[] args) {
        new LabelTest();
    }

    public LabelTest(){
        JFrame frame = new JFrame("Label Test");
    Container mainPanel = frame.getContentPane();

    mainPanel.add(getTitlePanel("My Label test
         text.", "/images/some.gif"));

    frame.pack();
    frame.setVisible(true);
  }

  private JPanel getTitlePanel(String title,
         String imagePath){
    JPanel titlePanel = new JPanel(new
BorderLayout());

      //use the Dimension class to make a 2-D
      //size of width, height
    titlePanel.setPreferredSize(
         new Dimension(300, 100));

      //make it white instead of default gray
    titlePanel.setBackground(Color.WHITE);

      //the image
    JLabel lblImage = new JLabel();
      //get an image from the path specified
    ImageIcon ico = new ImageIcon(imagePath);
      //add the image to the label
    lblImage.setIcon(ico);

      //the text
    JLabel lblTitle = new JLabel();
      //set the font for the text
    lblTitle.setFont(new Font("SansSerif",
          Font.BOLD, 18));
      //add the text to the label
    lblTitle.setText(title);

    //add the image to the panel
    titlePanel.add(lblImage, BorderLayout.WEST);
```

```
        //add the text to the panel

        titlePanel.add(lblTitle, BorderLayout.CENTER);


        return titlePanel;

    }

}
```

We will cover layouts, which are used above, later in this chapter. Otherwise, the code should be self explanatory. If the image is not found, then it simply won't appear.

JTextField

This control corresponds to an input control in HTML. Users enter text here, or you can cause it to have a default value when it is displayed by calling its setText(String s) method. Then, you can call the getText() method to retrieve its String value.

JButton

The JButton represents a clickable button that can have a text label, an image label, or both. The constructors follow:

- JButton() Creates a button with no set text or icon.
- JButton(String text) Creates a button with text label
- JButton(Icon icon) Creates a button with an icon label.
- JButton(String text, Icon icon) Creates a button with both text and icon.
- JButton(Action a) Creates a button where properties are taken from the Action supplied.

Check out the following ButtonDemo class to see a JButton with a text label that has several properties set, and which performs an action when you click it.

ButtonDemo.java

```
package net.javagarage.demo.ui;

/**<p>

 * Demonstrates how to use a text field, a button,

 * and a label. When you enter text and click the

 * button, the label's text changes to your value.

 *

 * This example uses the default layout, which

 * is FlowLayout. We don't have to do anything

 * to make that work, and components will just
 * stack up after each other.
 * @author Eben Hewitt
 **/
import javax.swing.AbstractButton;
import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JFrame;
import javax.swing.JTextField;

import java.awt.Color;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyEvent;

public class ButtonDemo extends JPanel
implements ActionListener {

protected JButton button;
protected JTextField txtField;
protected JLabel label;

public ButtonDemo() {
//creates the field to enter text
txtField = new JTextField();
//make the background yellow
txtField.setBackground(Color.YELLOW);
//will disappear if we don't do this
txtField.setColumns(25);
//creates the button object
button = new JButton("Change text");

//places text in the button
```

```
button.setVerticalTextPosition(AbstractButton.CENTER);

//this is left for locales that read left to right
button.setHorizontalTextPosition(AbstractButton.LEADING);

//change some frivolous things just to show
button.setBackground(new Color(80,80,80));
button.setForeground(Color.WHITE);
/*
 * it is sometimes desirable to allow the user to
 * type with the keyboard to fire an event, as
 * opposed to only allowing clicking. to do this, set
 * the mnemonic using the setMnemonic method.
 *
 * The arg to this method is a char, indicating the
 * character that will fire the event, often in
 * combination with the ALT key.
 *
 * it should be one of the characters in the label,
 * in which case it will appear underlined.
 *
 * See the KeyEvent API for pre-defined keys.
 * IE, we could use VirtualKey C,
 * which means ALT + C character.
 */

button.setMnemonic(KeyEvent.VK_C);
//inherited from AbstractButton
//this is what the event listener will pick up
button.setActionCommand("changeText");

//this is what makes the button do something
//when you click it. without registering
//and action listener, nothing will happen.
button.addActionListener(this);

//create the label whose text we'll change
label = new JLabel();
```

```
label.setText("This is the default text");



//Add the components to the container in order.
//since we're using FlowLayout, we don't need
//to do anything else
add(txtField);
add(button);
add(label);
}

//this is what happens when the button is
//clicked (or when ALT C is keyed)
public void actionPerformed(ActionEvent e) {
//we set the button's ActionCommand to this
if ("changeText".equals(e.getActionCommand())) {
//make the label value be whatever
//the user typed
label.setText(txtField.getText());
}
}

public static void main(String[] args) {
//the event-dispatching thread will show the app
//This method of running Swing apps is recommended
//by Sun
javax.swing.SwingUtilities.invokeLater(new Runnable() {
public void run() {
launch();
}
});
}

//sets up the look and feel, creates the frame,
//lines up the objects, and shows them.
```

```
private static void launch() {

JFrame.setDefaultLookAndFeelDecorated(true);


//Create and set up the window.

JFrame frame = new JFrame("Button Demo");

frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);


//Create and set up the content pane.

ButtonDemo newContentPane = new ButtonDemo();

newContentPane.setOpaque(true);

          //content panes must be opaque

frame.setContentPane(newContentPane);


//Display the window.

frame.pack();

frame.setVisible(true);

}

}

//end ButtonDemo
```

Spacing and Aligning Components

Spacers

There is a class that inherits from JComponent called Box. This is a lightweight class that uses BoxLayout as its Layout Manager. You can use this class to get at the convenience methods it makes available. These methods allow you to add invisible components to the layout that help you fit, position, and size the regions of your layout. Specifically, they help you do that by creating certain relationships between your visible control components.

We won't go into them in much depth. Suffice to say, here they are, here is what they're for, and go try them out for yourself to get the feel for them. This is the point in Swing programming that it becomes a little like learning to drive: eventually, you just need to feel when to shift gears and how quickly to let up on the clutch.

Glue

Glue represents space of malleable size in a layout. Adding glue between two components tells the runtime to take excess space in the layout and smoosh it between the components.

```
container.add(buttonOne);

container.add(Box.createHorizontalGlue());

container.add(buttonTwo);
```

You can also createVerticalGlue();.

Area

A rigid area is an invisible region that is always of the specified size. You use it to create a certain amount of space between two components that you don't want to change, or perhaps to push text up toward the top of your window by placing a rigid area below it.

An example follows:

```
static Component Box.createRigidArea(Dimension d)
```

Dimension takes two ints specifying the height and width.

```
container.add(buttonOne);

container.add(Box.createRigidArea(new

Dimension(10,0)));

container.add(buttonTwo);
```

Strut

A strut is an invisible component of fixed length. It is used to force space between two components. The strut has no width at all, except there is excess space surrounding it, in which case it will fill up the space in the manner of any component. There are two kinds of struts: vertical and horizontal.

Create a strut using these methods:

```
static Component createVerticalStrut(int height)

static Component createHorizontalStrut(int width)
```

Here is an example of how to use a strut to put space between two buttons:

```
container.add(buttonOne);

container.add(Box.createVeticalStrut(15));

container.add(buttonTwo);
```

You'll notice in this case, the strut acts like the rigid area. However, it is recommended that you use rigid areas instead of struts, because struts can have unlimited length, which could cause undesirable results in nested components.

PREV < Day Day Up > NEXT

JEditorPane

We create a JEditorPane in the blogger application in the Toolkit section. I refer you there to see in detail how that is used. But there is one question that is often asked when figuring out how to use JEditorPanes. And that's how to handle hyperlink clicks.

You do need to do something in fact. If you use a JEditorPane to display a simple HTML page, which is one of its main purposes, nothing will happen when you click on the links. Unless you create an object of a class that implements the HyperlinkListener interface, as shown in the following example:

```
try {
    String url = "http://www.javagarage.net";
    JEditorPane editorPane = new JEditorPane(url);
        //don't let user try to edit content!
    editorPane.setEditable(false);
    editorPane.addHyperlinkListener(new LinkListener());
} catch (IOException e) {
      //handle...
}

class LinkListener implements HyperlinkListener {
   public void hyperlinkUpdate(HyperlinkEvent event) {
      if (event.getEventType() ==
              HyperlinkEvent.EventType.ACTIVATED)
{
         JEditorPane pane =
               (JEditorPane)event.getSource();
         try {
            // Show the new page in the editor pane.
            pane.setPage(event.getURL());
         } catch (IOException e) {
            //handle..
         }
      }
   }
}
```

JTabbedPane

This component provides a way to access an arbitrary set of panels. It capitalizes on CardLayout shown earlier, and offers small tabs atop your JPanels to offer the user an easy way to switch around. This is used frequently in office productivity applications to let the user specify preferences and so forth. It is handy and surprisingly easy to use.

TabbedPaneDemo.java

```
package net.javagarage.demo.ui;

/**<p>

 * Shows a JTabbedPane, which allows you to

 * choose between pages with a tab, like

 * paper manila folders.

 *

 * </p>

 * @author Eben Hewitt

 **/


import javax.swing.JTabbedPane;

import javax.swing.ImageIcon;

import javax.swing.JLabel;

import javax.swing.JPanel;

import javax.swing.JFrame;

import javax.swing.JComponent;

import java.awt.BorderLayout;

import java.awt.Dimension;

import java.awt.GridLayout;

import javax.swing.SwingUtilities;

import java.net.MalformedURLException;

import java.net.URL;


public class TabbedPaneDemo extends JPanel {

static URL imageURL;


public TabbedPaneDemo() {

//remember that the call to super()
```

```
//MUST be first call in constructor

super(new GridLayout(1, 1));

//create the tabbed pane

JTabbedPane tabbedPane = new JTabbedPane();


/*remember that the present working directory

will be above your classes directory.

so if this class is in

C:\\projects\garage\net\javagarage\etc...

Put the image you want in C:\\projects\garage.

*/


String imagePath = System.getProperty("user.dir") +

"\\kitty.gif";

//get the image that will prefix each tab

ImageIcon icon = new ImageIcon(imagePath);


//create a new panel using our custom

//utility class

JComponent noodlePanel = new TabPanel("prissy

one").getPanel();

//add a tab to the panel

tabbedPane.addTab("Noodlehead", icon, noodlePanel);


JComponent doodlePanel = new

TabPanel("princess").getPanel();

tabbedPane.addTab("Doodlehead", icon, doodlePanel);


JComponent mrPanel = new TabPanel("the boy

kitty").getPanel();

tabbedPane.addTab("Mr. Apache Tomcat", icon, mrPanel);

//add the tabbed pane

add(tabbedPane);


//scrolling tabs makes small navigation arrows for

//use when all tabs do not fit on the screen
```

```
tabbedPane.setTabLayoutPolicy(JTabbedPane.SCROLL_TAB_
LAYOUT);
}


//Makes the icon of the image file we provide, to
//include on each tab. If no file is specified or
//found, null is returned, and no icon is included.
protected static ImageIcon createIcon(String path) {
try {
imageURL = new URL(path);
return new ImageIcon(imageURL);
} catch (MalformedURLException me){
System.err.println("This is not an acceptable URL: " +
path + " \n" + me.getMessage());
return null;
}
}

/**
 * Put the application on the event-handling thread
 * and start it up.
 */
private static void launch() {
//must be called first, and static-ly
JFrame.setDefaultLookAndFeelDecorated(true);

//Create and set up the window.
JFrame frame = new JFrame("Tabbed Pane App");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

//create a new instance of this class and
//make the contents visible
JComponent newContentPane = new TabbedPaneDemo();
newContentPane.setOpaque(true);
frame.getContentPane().add(newContentPane,
BorderLayout.CENTER);
//show it
frame.pack();
frame.setVisible(true);

}


public static void main(String[] args) {

//run the app as recommended

SwingUtilities.invokeLater(new Runnable() {

public void run() {

launch();

}

});

}

}


/**

 * Class to encapsulate the business of creating panels

 * that hold the content of the application (in this

 * case just a line of text). The tabs tack on to the

 * top of these panels.

 */

class TabPanel {

JPanel panel;

JLabel filler;

public TabPanel(String text){

panel = new JPanel(false);

filler = new JLabel(text);

filler.setHorizontalAlignment(JLabel.CENTER);

panel.setPreferredSize(new Dimension(250,100));

panel.add(filler);

}


public JComponent getPanel() {

return panel;

}

}


//end TabbedPaneDemo



} //end components.stepInto()
```

LayoutManagers Overview

LayoutManager.getDefinition(): //returns: Layout managers provide different metaphors for organizing the atomic components within a content pane. Layout Managers include FlowLayout, BorderLayout, BoxLayout, CardLayout, GridLayout, and GridBagLayout. You can create your own layout managers, or use a null layout manager, which requires you to position each component using absolute pixel values. FlowLayout is used as the default layout manager if you do not specify a particular one you'd like to use.

LayoutManager.stepInto() { We will step into layout managers for a moment here so that we can forget about them for the rest of the topic. They are the underlying arranger of the space, so it is appropriate to start with them. Just don't get too bogged down by them. I'll try to keep it peppy.

If you have used HTML to make a Web page, using layout managers is a bit like using tables and CSS to position different elements on your page. Layout managers dictate (usually) the size and position of components within the UI.

You set the layout of a panel using the setLayout() method, as in the following:

```
//get the main content pane of your frame:

Container pane = frame.getContentPane();


//make this pane to use border layout

pane.setLayout(new BorderLayout());


//put a component in a region

pane.add(new JtextField(10), BorderLayout.EAST);
```

Let's look at each layout manager. To do so, we will create a totally generic frame using the BasicFrame class so we don't have to retype a bunch of code for each example.

BasicFrame.java

```
package net.javagarage.demo.swing.layouts;


import javax.swing.JFrame;

/** <p>

Just for use in demo-ing different Layout

managers. Implements the singleton pattern.
```

```
**/
public class BasicFrame extends JFrame {

private static BasicFrame frame;

private BasicFrame() {
super("My Frame");

}

public static BasicFrame getInstance() {
if (frame == null)
frame = new BasicFrame();
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
return frame;

}
}
```

Switch (LAYOUT_MANAGERS) {

Now we look at different ways to organize controls on your panels.

Case: FLOW LAYOUT :

This is the simplest to use, and is used in many Swing examples in this book. In part, that's because you don't have to do much to make it go, and it is used by default by Jpanels. It arranges components in a flow—that is, by adding them in a certain direction. The effect is similar to typing into a word processor: as you add new words, they just get tacked onto the end. FlowLayout has three constructors.

1. FlowLayout()

   Nuff said.

   There are two component orientations possible for a Flow Layout: left to right and right to left. They are

   ```
   ComponentOrientation.LEFT_TO_RIGHT

   ComponentOrientation.RIGHT_TO_LEFT
   ```

   You specify which you want to use like this:

   ```
   contentPane.setComponentOrientation(

   ComponentOrientation.RIGHT_TO_LEFT);
   ```

2. FlowLayout(int alignment)

   Three alignments can specify the orientation in the FlowLayout constructor. They are

   ```
   FlowLayout.LEADING (components should be left-aligned)

   FlowLayout.CENTER

   FlowLayout.TRAILING (components should be right-aligned)
   ```

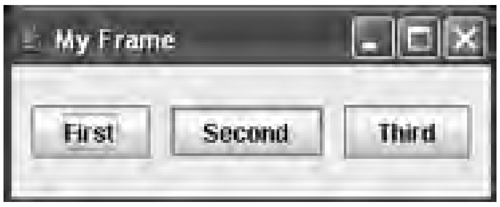
3. public FlowLayout(int alignment, int horizontalGap, int verticalGap)

   The horizontal and vertical gap parameters allow you to specify the number of pixels to put between each component. The default gap is 5.

FlowLayoutExample.java

```
package net.javagarage.demo.swing.layouts;

import java.awt.Container;
import java.awt.FlowLayout;
import javax.swing.JButton;

/**<p>
 * Shows how to use a FlowLayout
 * with an alignment and horizontal and
 * vertical gaps
 * between each component.
 * </p>
 * @author Eben Hewitt
 **/
public class FlowLayoutExample {
public static void main(String[] args) {
BasicFrame frame = BasicFrame.getInstance();
Container pane = frame.getContentPane();

int hGap = 10;
int vGap = 20;
pane.setLayout(new FlowLayout(FlowLayout.RIGHT,
          hGap, vGap));

pane.add(new JButton("First"));
pane.add(new JButton("Second"));
pane.add(new JButton("Third"));

frame.pack();
frame.setVisible(true);
}
}
```

Figure 25.2. A flow layout with the mouse hovering over the second button.



Case: BORDER LAYOUT :

Border layout resizes and arranges the components within it to fit in five different regions: PAGE_START, LINE_START, CENTER, LINE_END, and PAGE_END. If you don't specify a constant indicating the region onto which you want to place a component, it will be added to the center.

The placement of the regions looks like this:

PAGE_START

LINE_START CENTER LINE_END

PAGE_END

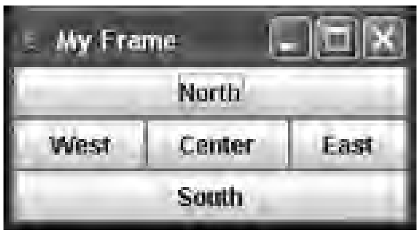
Components in a Border layout are layed out according to their preferred size. That means that filling each region with a JButton will allow each component to fill up the space, but adding JLabels to each region will not, as the label's preferred size is just large enough to hold its data.

BorderLayoutExample.java

```
package net.javagarage.demo.swing.layouts;


import java.awt.BorderLayout;

import java.awt.Container;

import javax.swing.JButton;

import javax.swing.JLabel;


/**

 * Demonstrate using border layout.

 * @author Eben Hewitt

 * @see BasicFrame

 **/

public class BorderLayoutExample {
```

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

BasicFrame frame = BasicFrame.getInstance();

Container pane = frame.getContentPane();

pane.setLayout(new BorderLayout());
pane.add(new JButton("Center"), BorderLayout.CENTER);
pane.add(new JButton("North"), BorderLayout.NORTH);
pane.add(new JButton("South"), BorderLayout.SOUTH);
pane.add(new JButton("East"), BorderLayout.EAST);
pane.add(new JButton("West"), BorderLayout.WEST);

frame.pack();
frame.setVisible(true);
}
}//end BorderLayoutExample
```

Figure 25.3. The border layout.



Case: BOX LAYOUT :

This layout manager arranges components from one of two directions: horizontally or vertically. The components in each region will not wrap, so if you resize the window, the components will stay arranged as you specified.

The BoxLayout class features only one constructor.

```
BoxLayout(Container cont, int axis);
```

You create a box layout specifying the orientation (axis) you want to use. It dictates along what kind of line your components will be layed out. You have four choices.

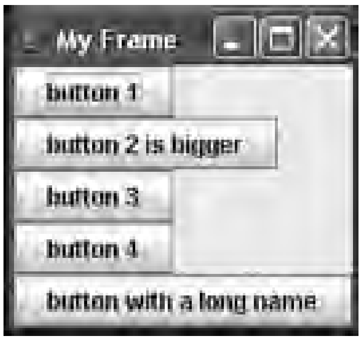
1. X_AXIS: Add components horizontally, from left to right.
2. Y_AXIS: Add components vertically, from top to bottom.
3. LINE_AXIS: Components should be layed out according to the value of the container's ComponentOrientation property. That is, if the container's ComponentOrientation is horizontal, components are layed out horizontally.
4. PAGE_AXIS: Components should be layed out according to the value of the container's ComponentOrientation property. If the container's orientation is horizontal, components are layed out vertically; otherwise, they are layed out horizontally, and components are layed out left to right. For vertical orientations, components are always layed out from top to bottom.

The first argument to the constructor specifies the container that you want to stick the layout to.

BoxLayoutExample.java

```
package net.javagarage.demo.swing.layouts;

import java.awt.Container;
import javax.swing.BoxLayout;
import javax.swing.JButton;

//demos a simple box layout
public class BoxLayoutExample {

public static void main(String[] args) {
doSimple();
}

static void doSimple(){
BasicFrame frame = BasicFrame.getInstance();
Container pane = frame.getContentPane();

//when you add components, they will be
//added vertically, from top to bottom
pane.setLayout(new BoxLayout(pane,
BoxLayout.Y_AXIS));

pane.add(new JButton("button 1"));
pane.add(new JButton("button 2 is bigger"));
pane.add(new JButton("button 3"));

pane.add(new JButton("button 4"));

pane.add(new JButton("button with a long name"));


frame.pack();

frame.setVisible(true);

}

} //end of simple box layout
```

Figure 25.4. A simple box layout.



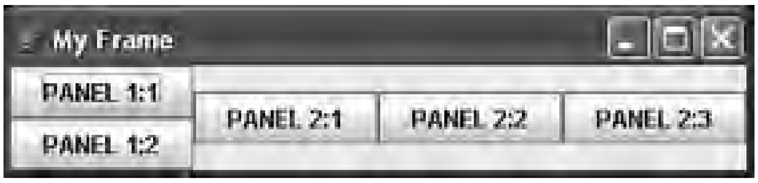
Note that you can nest box layouts to achieve more sophisticated effects, and exert greater control over how your components are arranged. What follows is an example of how to do just that sort of thing.

NestedBoxLayout.java

```
package net.javagarage.demo.swing.layouts;


import java.awt.Container;

import javax.swing.BoxLayout;

import javax.swing.JButton;

import javax.swing.JPanel;


// demos nested box layouts

public class NestedBoxLayouts {
```

```
public static void main(String[] args) {
BasicFrame frame = BasicFrame.getInstance();
Container pane = frame.getContentPane();

//X == Horizontal
pane.setLayout(new BoxLayout(pane,
BoxLayout.X_AXIS));

//we're going to add two panels to the
//content pane
JPanel p1 = new JPanel();
JPanel p2 = new JPanel();

//so when we add the panels, they will go side
//by side (horizontally)
pane.add(p1);
pane.add(p2);

//now make each panel have its OWN box
//layout panel one (the left panel)
//will arrange its components vertically (Y)
p1.setLayout(new BoxLayout(p1, BoxLayout.Y_AXIS));

//panel 2, the right panel, will arrange
//horizontally, from left to right
p2.setLayout(new BoxLayout(p2, BoxLayout.X_AXIS));
//add two buttons to the left panel
//notate buttons like this:
//Panel Num : Button Num
p1.add(new JButton("PANEL 1:1"));
p1.add(new JButton("PANEL 1:2"));

//add three buttons to the right panel
p2.add(new JButton("PANEL 2:1"));
p2.add(new JButton("PANEL 2:2"));
p2.add(new JButton("PANEL 2:3"));

frame.pack();
frame.setVisible(true);

}

} //end NestedBoxLayout
```

Figure 25.5. One box layout nested within another allows more sophistication.



Note that we have nested layouts of the same kind here to demonstrate how to do it so you get the control you need. But you don't have to nest layouts of the same type. You can nest a grid layout inside a box layout inside a flow layout if you want. It is common to nest layouts to get the functionality, flexibility, and control required.

Case: CARD LAYOUT:

A card layout treats each component in the container as a card on a stack. Just as with a deck of cards, only one card is visible at a time. As with much in Swing programming, the order of invocation matters; the first panel added to the layout is what is displayed when the frame is first shown.

The CardLayout class offers a set of methods to flip through the cards sequentially, or to show a particular card to show a panel based on its String name. This is useful in working with weezards (step-by-step panels the user must go through). Note that Swing also gives you for free the tabbed pane, which is covered next with an illustrative example in case you need that kind of specific functionality. In fact, using a tabbed pane is easier than using card layout, because a JTabbedPane implements its own layout.

Following are the steps to make this layout go:

```
//create a panel to hold all the cards

JPanel cards;

final static String PANEL_ONE = "My first panel";

final static String PANEL_TWO = "My second panel";


//...

//now make each card, which is just a JPanel

JPanel card1 = new JPanel();

JPanel card2 = new JPanel();


//...


//initialize the panel that holds the cards
```

```
cards = new JPanel(new CardLayout());


//they get String names here

cards.add(card1, PANEL_ONE);

cards.add(card2, PANEL_TWO);


//Now to show a card, you get the layout from the

cards JPanel

CardLayout layout = (CardLayout)(cards.getLayout());


//call the show() method and pass it the String name

layout.show(cards, PANEL_TWO);
```

That's all you need to do. The following methods of the CardLayout class let you choose panes:

```
void first(Container)

void next(Container)

void previous(Container)

void last(Container)

void show(Container, String)
```

Again, unless you're doing a wizard, it's generally easier to use a JTabbedPane to get the same effect.

Case: GRID LAYOUT:

Grid layout presents components within a grid. Each cell in a grid is the same size. You place one component in each cell. Components within a grid cell take up all of the available space in the cell.

There are three constructors available for creating a grid layout.

- public GridLayout() Creates a grid with a single row, and one column for each component.
- public GridLayout(int rows, int cols) Creates a grid with the specified number of rows and columns. Sometimes.
- public GridLayout(int rows, int cols, int hGap, int vGap) Creates a grid with the specified number of rows and columns (sometimes), and adds space between the components.

Note that you can also set the number of rows and columns using the methods setRows() and setColumns(). However, the number of columns you specify is ignored completely if the value you specify is non-zero. Whether you use setColumns() or pass them into the constructor. In this case, the number of columns is determined by the number of components you have over the number of rows. Huh?

If I create a grid layout and create six components with five rows, I have more components than rows. So the layout will spill over, creating two columns. It then will distribute the components as evenly as possible over each column—ignoring my request to set a certain number of columns. I wind up with two columns of five rows each, with components in each of the first three rows of both columns, and empty cells at locations (4,0), (4,1), (5,0), (5,1).

These methods will throw an IllegalArgumentException if both the rows and columns are set to 0.

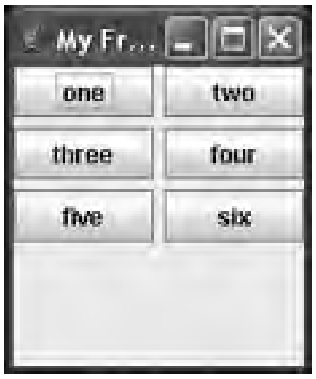
Here is an example.

GridLayoutExample.java

```
package net.javagarage.demo.swing.layouts;


import java.awt.Container;

import java.awt.GridLayout;


import javax.swing.JButton;


//demo how

public class GridLayoutExample {

static int rows, cols, hGap, vGap;


public static void main(String[] args) {

BasicFrame frame = BasicFrame.getInstance();

Container pane = frame.getContentPane();

//let's have 5s—all around!

rows = cols = hGap = vGap = 5;


//create layout: notice how my pleas

//for certain number of columns is viciously

//ignored, laughed at even.

pane.setLayout(new GridLayout(rows, cols, hGap, vGap));


//add components

pane.add(new JButton(" one "));

pane.add(new JButton(" two "));

pane.add(new JButton(" three "));

pane.add(new JButton(" four "));

pane.add(new JButton(" five "));

pane.add(new JButton(" six "));
//we have space left over for two more

//in each column because we have 10 cells:

//5 rows * 2 columns


frame.pack();

frame.setVisible(true);

}

}

//end of Grid Layout example
```

Notice that if you resize the window, the layout does not shift. You still have the number of columns the layout manager created.

Figure 25.6. The grid layout.



Case: GRID BAG LAYOUT:

FRIDGE

You have to be careful in keeping track of your grid. If you add components into the same cell, they can overwrite each other in the display, causing one to be in front of the other. This is not only undesirable in itself, but can cause strange behavior. For example, if you resize the window, you might find the higher z-indexed component flipping in front of the other.

GridBagLayout (a.k.a. GBL) can be very tricky to work with. It is far more complex to create, manage, keep track of, and modify than other layout types. Things sort of come down to two choices in Swing apps: nest a number of other layout types within each other as necessary, or use GridBagLayout.

Now that we're scared, all a GBL does is allow the placement of components in a grid of rows and columns (so far so good, it's just like Grid Layout, we can do this, okay, what's next?…), *allowing components to span multiple rows and/or columns* (nnnnnooooooooooo!!!!!!!).

Not all columns have to have the same width, and not all rows have to have the same height.

This is actually a lot like nested HTML tables, isn't it? It is enough like them that we'll be able to manage it. If you can write an HTML page that nests tables within tables to control your layout at a granular level, you are a lot of the way home with GBL. The problem turns out to be not so much the GBL itself, but the fact that the layout gets the size of the cells in the grid from the preferred size of the components within it. For that reason, you will likely have to do some experimenting as you work with GBL, because this changes across components. For example, a button will fill up all of the available space, whereas a label will only take what it needs bare minimum.

Constraints

The GBL specifies the size and position of each component using constraints. Constraints are objects of type GridBagConstraint. You don't need a new GridBagConstraint for each component; you can reuse a constraint if it's convenient.

When you create a constraint, you can set the following fields: gridx and gridy.

Specify the row and column at the upper left of the component, with coordinates (0,0) representing the upper-left column. X represents the horizontal plane and Y the vertical plane. You can also use the default value GridBagConstraints.RELATIVE to indicate that the component should be placed immediately to the right of or immediately below the component that was added to the container just before this component was added.

- Gridwidth Specify the number of columns in the display. Default is 1.
- Gridheight Specify the number of rows in the display. Default is 1.
- Fill Used to indicate the manner in which to resize the component when the component's display area is larger than the component's requested size.

  Possible values:

  FRIDGE

  GridBagLayout allows a component to span multiple rows only when the component is in the column farthest to the left, or the component has positive gridx and gridy values.

  NONE Default.

  HORIZONTAL The component will entirely fill its display area horizontally. The height will not change.

  VERTICAL The component will entirely fill its display area vertically. The width will not change.

  BOTH The component will fill its entire display area.

- ipadx, ipady Specifies how much padding to add to the minimum size of the component. Default is 0.
- Insets Specifies the minimum amount of space between the component and the edges of its display area (internal padding). The value is specified as an object of type Insets. By default, components have no external padding.
- Anchor Use to determine where to place your component when the component is smaller than its display area. Valid values are CENTER (default), PAGE_START, PAGE_END, LINE_START, LINE_END, FIRST_LINE_START, FIRST_LINE_END, LAST_LINE_END, and LAST_LINE_START.
- weightx, weighty Use the values to specify how to distribute space among columns (weightx) and among rows (weighty) during resizing.

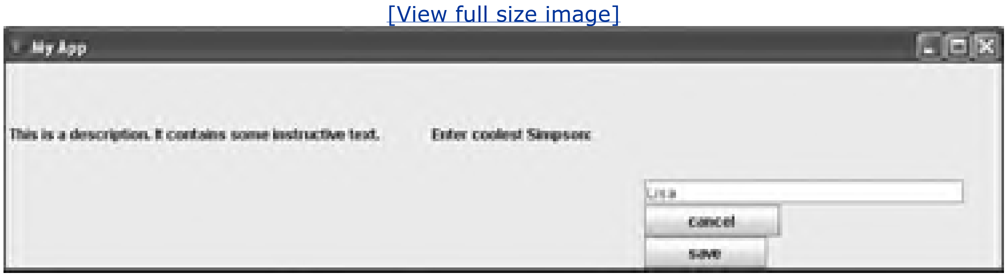
GridBagLayoutExample.java

```
package net.javagarage.demo.swing.layouts;

import java.awt.Container;
import java.awt.Dimension;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;

import javax.swing.Box;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;

//you could do this to save typing:
//import static javax.swing.GridBagConstraints.*;
//i don't here in order to be explicit

//demos a grid bag layout
public class GridBagLayoutExample {

public static void main(String[] args) {
doSimple();
}

static void doSimple(){
//BasicFrame frame = BasicFrame.getInstance();
JFrame frame = new JFrame("My App");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
Container pane = frame.getContentPane();

//create the layout and constraints
GridBagLayout layout = new GridBagLayout();
GridBagConstraints constraints = new GridBagConstraints();
frame.setLayout(layout);

//create all of the controls we will add
JLabel lblDesc = new JLabel(" This is a description. " +
"It contains some instructive text. ");

JLabel lblEnter = new JLabel("Enter coolest Simpson: ");

JTextField txtField1 = new JTextField(20);

JButton btnSave = new JButton("save");

JButton btnCancel = new JButton("cancel");


//now create new constraints for each component

//before adding it to the pane


//constrain Description Label

//gives height to the whole area

constraints.ipady = 50;

//puts space around the label

constraints.ipadx = 20;


layout.setConstraints(lblDesc, constraints);

pane.add(lblDesc);


//constrain Enter label

constraints.ipady = GridBagConstraints.NONE;

layout.setConstraints(lblEnter, constraints);

pane.add(lblEnter);


//constrain Text Field

constraints.gridwidth = GridBagConstraints.CENTER;

constraints.anchor =

GridBagConstraints.LAST_LINE_START;

layout.setConstraints(txtField1, constraints);


pane.add(txtField1);


//constrain Cancel Button

constraints.gridx = 2;

//add constraints to Cancel button

layout.setConstraints(btnCancel, constraints);

//add Cancel button to the pane
```

```
pane.add(btnCancel);

pane.add(Box.createRigidArea(new Dimension(30,0)));

//constrain Save button
constraints.gridx = 3;
constraints.gridy = 3;

layout.setConstraints(btnSave, constraints);
pane.add(btnSave);

frame.setSize(frame.getPreferredSize());

frame.pack();
frame.setVisible(true);
}
}
```

Figure 25.7. The grid bag layout in action.

[View full size image]



} //end LayoutManagers.stepInto();

Handling Action Events

Examples in the Toolkit, such as the GaragePad doodler, show how to capture mouse motion events. There are also examples in the doodler application and the text editor that show how to handle action events from menu items when they're clicked.

In Swing, you create user controls for the user to interact with. When the user does something to a control—for example, clicks a button—that action fires an event. You create listeners that do something when that event is fired, and then attach them to the user controls. Actions can optionally hold information about tool tip text or icons, though this is rare.

Specifically, to handle an event you override the actionPerformed() method.

```
public void actionPerformed(ActionEvent evt) {

        // do something

}
```

There are a couple of different ways to do this. Please see the blog on Inner Classes earlier in the book for a discussion and demonstration of implementing action listeners. Please see the Toolkit in the back of this book for numerous, complete, working examples of handling action events.

Creating Menus

It is easy in Swing to create menus. First, you create a menu bar to hold each menu. A menu is the little area that drops down out of the menu bar.

```
//create the menu bar

JMenuBar menuBar = new JMenuBar();


// Create a menu

JMenu menu = new JMenu("Tools");

menuBar.add(menu);


// Create a menu item

JMenuItem hacksaw = new JMenuItem("Hacksaw");

hacksaw.addActionListener(someActionListener);

menu.add(hacksaw);


// Install the menu bar in the frame

frame.setJMenuBar(menuBar);
```

The main thing to note here is that clicking menu items creates an action event that you need to handle with an ActionListener implementation. There is a good example of using menus in the Garage Text Editor Toolkit example.

A Note About Mixing Swing and AWT Components

Swing components are lightweight. AWT components, now out of fashion, are heavyweight. That means that it is a bad idea to mix the older AWT components with Swing components. The reason is that you will get unpredictable behavior. More specifically, you will find that the AWT components will always force themselves in front of the sweet little lightweight Swing components. You could have serious trouble displaying a Canvas, for example, in tandem with a JPanel. Just something to beware of.

The designers of the Swing libraries learned a lot from working with AWT, and you should be able to find the components you need in the newer libraries without having to mix.

Disclaimer: The most popular book on Swing is 2 volumes and nearly 3,000 pages. We don't have room for much more. We are really touching the surface here, but this is enough to get started. As this book shows, you can make a number of interactive and GUI programs without going too far into layout managers, and after you have handled one action, you've handled them all.

Chapter 26. BLOG ENTRY: SOFTWARE DEVELOPMENT BLACK MARKET

DO OR DIE:

- Star Date 8675309

There is not one thing called a software development department. Or the "Programming Division of IS." Or whatever it is called at your company. Each department is very different whether it employs 3 programmers, or 30, or 3,000. It's different yet again whether you help sell books for an e-commerce site, make garage doors or thermostats work, help create a program that gets burned onto CDs and shrink-wrapped and sold in Wal-Marts throughout the world, or make, automate, and integrate business applications for a government entity. But in each of these very different places seethes a very real and very scary possibility—the possibility that your team is slowly bled for resources (people, money, tools, access, and so on). We have all felt it. That's bad for us, and we have all talked it to death because it's a drag. What we don't talk about as much is what happens to the organization when it starves us like that. What happens should be no surprise. The same thing happens in the world when people are starved of resources, money, tools, and access to things they need. Think about books that have been banned. Or the prohibition on alcohol in the 1920s. Or drugs of all kinds. These things don't cease to circulate because they've been outlawed. They still thrive. But they thrive in a black market. Which is a very dangerous place. Maybe you've never experienced this developer's black market. Maybe the reason that you've stayed at your same company for 10 years (ha) is that they lavish you with time, money, resources, tools, people, and access. But I rather doubt it.

A black market is where people illicitly acquire and distribute goods and services.

Do you work in a developer's black market? Ask yourself the following. Have you ever run down the hall to the offices of your friends down in networking and asked for write permissions on a directory somewhere on the network? You just need this quick fix, just this once, then your app will work and you can all go to lunch, you tell her. You explain that there is not a security policy in place but this would do the trick and your manager is on a conference call and you can't move forward until that directory is available for your app to write to. This workplace has a black market.

If you call someone in your organization that works in a related department and get him to do a favor for you to help make your application work, you have a black market. Your organization's software is dependent on the personal good will that its workers generate toward others, or, if you are not an optimist, how easy it is for you to manipulate or coerce your coworkers.

You shouldn't have to cut deals to get a project done. If you have to cut deals to get your project done, it is because your organization is asking for something that they can't afford or don't want to pay for.

In my view, you're going to either pay now, or pay later. And when you pay later, the cost is always, without fail, far, far more. This is true for testing and commenting and debugging too.

If a company lets its departments run on the black market, it is going to pay later.

If your department's product depends on a black market, you must help stop it immediately. Determine what you need to get your job done. Not what they want to hear. What you really need. The real hours. The real cost of 14 copies of that IDE. The real number of programmers. These things we are used to asking for. We do this a lot. It's easy to put them into MS Project and everyone can print it out and think they have a plan. Typically, this is the culprit right here.

Just like no one in their right minds would think that a significant set of ideas can be satisfactorily represented in a whiz-bang PowerPoint presentation, don't think for a second that the fundamental coherence of a plan can be represented in MS Project.

I am not bashing Microsoft here. Those applications are great for what they do. I am saying that they don't do what we hope they would: they don't give us a plan anymore than Word writes the great American novel simply because it has a (flawed) grammar checker and lets you type.

Do not mindlessly assent to working on incoherent projects. You will become miserable. Ultimately you will quit and try to sell your family on how exciting Fresno really can be since they have a new programmer position waiting for you there.

Impress upon your managers the importance, the necessity, of a long-term, coherent plan.

Do not make one-off apps in the dark. Consider long-term architecture. Make modular applications that allow you to maximize the re-use of code. This kind of thing takes planning, yes. But it also takes knowledge of what you're going to do next, so that you can see what you might want to re-use.

Ensure that changes or architecture you want to introduce will have a visible, positive impact on customers. If it doesn't help your customers, you have a very hard case to make.

Yes, business changes. Yes, the business managers cannot predict where the market will go. They don't have a crystal ball. No one is asking them to predict the future. Many times, what project you do next depends on whether you get a certain grant, or whether a certain bid is successful. Obviously.

If you are working on business applications or creating a product that might participate in a suite of products, there are things you need to know. You need support from people with power regarding your plans. The people in power don't know this. Let them know it.

It is impossible to build a service-oriented architecture in an organization without a long-term plan and the knowledge, input, and support of the business people who make decisions about your overall long-term goals. It is likely that they do not know or care what a service oriented architecture is. Show them the benefit, or the necessity, and tell them you need some information. Because you can't string apps together with tape.

You need to consider a framework, and cross-cutting features such as authentication, authorization, customization, interoperability, platform support and reliance, and so on. And if your job is to write code, you almost certainly don't have the answers to these questions on your own.

We in IT departments are in a tough spot. It is polite for us to suggest that we only exist to support the larger goals of our government, or our basketball shoe-making company, or even our software development company. This is obviously, trivially true. It is not the whole story. Business people do not think of us as a real department, such as Finance or Human Resources. They think of us as a meta-department. An expense. As if Human Resources or Finance or the Police Department is not an expense.

If you work on business apps or infrastructure, you are not simply in product development. You're serving two masters, each of which only knows the other by its sordid reputation. And that politesse can actually be a disservice to the business. Demand to participate in the goals. Because IT is different: It is a separate entity and yet a meta-entity that touches everyone every day.

Demand that the stake holders in an application are present and communicative. Software projects fail when organizations think that it is only the job of the software developers to make software. Stakeholders must participate. Management must provide resources and get behind projects. Customers must use the software.

The IT department must understand the long-term goals of an operation, and make a long-term IT plan to support those goals. Often we go this far. But then we start working on our own and a black market springs up. Go a step further. Go back to the business people and get their buy and support of your plan. Route out the black market.

Chapter 27. DATES AND TIMES

TO DO LIST:

- Dry cleaning
- Pick up the girl from soccer
- Put Java Dates and Times to good use

Here we go, into the land of dates and times. Dates are not the sorts of things I think are fun. I find dates cruel, punishing, and difficult to wield in different languages. It's surprising. I would think that dates would be fairly straightforward—after all, we've had dates for a long time. Our various calendars are more or less elegant, but their implementation and use in programming languages often seems to fall short of the mark. They are complicated, messy things.

Everybody knows what dates are and I wager that few readers are interested in knowing how and why they work the way they do in Java. I wager that you want a few utility methods that will just help you solve your current problem, no questions asked. If I'm wrong, the Java API makes terrific, suspense-filled reading.

Dates

In this topic, I will just present a few classes that do a lot of different things with dates. You can see what kinds of classes, methods, and fields are available to you, and hopefully you can just copy and paste methods here with little modification into your own projects.

Dates.java

```
package net.dates;

import java.util.*;
import java.text.*;

/**
This class contains a number of useful methods
that make it convenient to do common operations
involving dates. It uses java.util.Calendar and
java.util.GregorianCalendar to do so.
<p>
Hopefully you can paste this
directly into your own utilities library.
<p>
Just modify the classes to accept calendar objects
and return the int values. I didn't do this here
so that it would make the code easier to read.
<p>
See the Java API documentation for Calendar and
GregorianCalendar for many examples of usage.
*/

public class Dates {

Calendar calendar = Calendar.getInstance();


public static void main(String...ar) {
    numberOfDaysInMonth();

    dayOfWeek();

    getDateDifference();

}




//find out the number of days in a month of a given year

public static void numberOfDaysInMonth() {


//make calendar of the month

//you could pass this in as a parameter

    Calendar c = new GregorianCalendar(2004,

            Calendar.JULY, 1);


    //get number of days in this month

    int days = c.getActualMaximum(Calendar.DAY_OF_MONTH);


log(days); //31


    //month in a leap year

    c = new GregorianCalendar(2004,

            Calendar.FEBRUARY, 1);

    days = c.getActualMaximum(Calendar.DAY_OF_MONTH);


log(days);// 29


}


//gets the number of years difference from

//some past date and now

public static void getDateDifference() {


    Calendar now = Calendar.getInstance();


//the date we want to compare to now: Shakespeare is born

    Calendar compareDate =
new GregorianCalendar(1564, Calendar.APRIL, 23);

    //get difference based on year
    int yearDifference = now.get(Calendar.YEAR) -
compareDate.get(Calendar.YEAR);

log(yearDifference);//440 years ago

}



//The day-of-week is an integer:
//1 (Sunday), through 7 (Saturday)
public static void dayOfWeek() {

    Calendar anv = new GregorianCalendar(2004,
                   Calendar.MARCH, 27);
    int dayOfWeek = anv.get(Calendar.DAY_OF_WEEK); // 7

log (dayOfWeek);

}

public String getDate() {
        return getMonthInt() + "/" + getDayOfMonth() +
                "/" + getYear();
    }
/**
     * Useful for returning dates in a style
     * presentable to end users.
     * Returns date in this format:
     * Thursday, October 10, 2002
     * </p>
     */
    public String getLongStyleDate() {
        String myDate =
               getDay() + ", " + getMonth() + " "
               + getDayOfMonth() + ", " + getYear();
        return myDate;
    }

    public String getSlashDate() {
        String myDate = getMonthInt() + "/"
               + getDayOfMonth() + "/" + getYear();
        return myDate;
    }

        ///returns the year
    public int getYear() {
        return calendar.get(Calendar.YEAR);
    }
        ///gets the month as a String
public String getMonth() {
        int mo = getMonthInt();
        String[] months = { "January",
                 "February", "March",
                 "April", "May", "June",
                 "July", "August", "September",
                 "October", "November", "December" };
        if (mo > 12)
            return "?";

        return months[mo - 1];
    }
        //gets month as int on scale of 1 - 12
        //it is 0 - 11 by default
public int getMonthInt() {
    return calendar.get(Calendar.MONTH) + 1;
    }

public int getDayOfMonth() {
        return calendar.get(Calendar.DAY_OF_MONTH);
    }
        //gets the day of week as a string
public String getDay() {
          int d = getDayOfWeek();
          String[] days = {"Sunday", "Monday",
                   "Tuesday", "Wednesday",
               "Thursday", "Friday", "Saturday"};

          if (d > 7)
              return "error";

      return days[d - 1];
}

public int getDayOfWeek() {
          return calendar.get(Calendar.DAY_OF_WEEK);
}

public int getEra() {
          return calendar.get(Calendar.ERA);
}

public String getUSTimeZone() {
          String[] zones = {"Hawaii", "Alaskan", "Pacific",
                        "Arizona", "Mountain", "Central",
                        "Eastern"};
          return zones[10 + getZoneOffset()];
      }
public int getZoneOffset() {
       return calendar.get(Calendar.ZONE_OFFSET)/
                            (60*60*1000);
}

public int getDSTOffset() {
          return calendar.get
                    (Calendar.DST_OFFSET)/(60*60*1000);
}

public int getAMPM() {
          return calendar.get(Calendar.AM_PM);
}

public String getTime() {
          return getHour() + ":" + getMinute() + ":" +
                 getSecond();
```

```
}

public int getHour() {
     return calendar.get(Calendar.HOUR_OF_DAY);
}

public int getMinute() {
     return calendar.get(Calendar.MINUTE);
}

public int getSecond() {
     return calendar.get(Calendar.SECOND);
}

public static String getDateAsString(Date dt) {
     return getDateAsString(dt, DateFormat.SHORT);
}

public static String getDateAsString(Date dt, int
                      format) {
     if( dt == null )
         return "";

     return DateFormat.getDateInstance(format)
                    .format(dt);
}

public String getDateTime() {
     return getDate() + " " + getTime();
}

public static java.util.Date getDate(String str)
                               throws ParseException {
     return
     DateFormat.getDateInstance(DateFormat.SHORT).
                parse(str);
}

    /**
```

```
     * <p>Useful for getting the date in a format
     * useable by JDBC.
     * Returns current date in this format:
     * 2004-03-21 for March 21st, 2004
     * </p>
     */
public static java.sql.Date getCurrentJDBCDate(){
     java.sql.Date date = new java.sql.Date(
               (new java.util.Date()).getTime());
     return date;
  }

public static java.sql.Date
getJDBCDate(java.util.Date dt) {
     if( dt == null ) {
        return null;
     }
     return new java.sql.Date(dt.getTime());
}

     //will print out anything we pass it
     //just a convenience to save typing
private static <T> void log(T msg) {
     System.out.println(msg);
}

}
```

Notice that the preceding class has a number of methods that make it convenient to use dates in various situations: to print to the screen in a user-friendly way, to pass to an SQL database (most of these methods store dates in a manner different than Java does), and for representing dates in different formats appropriate for different occasions.

Time

The following class contains methods for formatting times in ways you are likely to want to do.

Times.java

```
package net.dates;

import java.util.*;

import java.text.*;


public class Times {


 Calendar calendar = Calendar.getInstance();


    public static void main(String...a) {

    log(get24HourTime() );

    log(getAMPMTime() );

    log( getSimpleTime() );

  }


/**

    * Returns time in this format:

    * 14.36.33

    */

   public static String get24HourTime() {

   Date d = new Date();

   Format formatter = new SimpleDateFormat("HH.mm.ss");

   return formatter.format(d);

   }

   /**

    * Returns time in this format:

    * 01:12:53 AM

    */

   public static String getAMPMTime() {
```

```
   Format formatter = new SimpleDateFormat("hh:mm:ss a");

   return formatter.format(new Date());

   }


   public static String getSimpleTime() {

      Format formatter = new SimpleDateFormat("h:mm a");

         return formatter.format(new Date());

   }


public int getHour() {

      return calendar.get(Calendar.HOUR_OF_DAY);

   }


   public int getMinute() {

      return calendar.get(Calendar.MINUTE);

   }


   public int getSecond() {

      return calendar.get(Calendar.SECOND);

   }


   public String getTime() {

      return getHour() + ":" + getMinute() + ":" +

            getSecond();

   }


      //will print out anything we pass it

      //just a convenience

   private static <T> void log(T msg) {

      System.out.println(msg);

   }

}
```

Running this class outputs something like this, depending on what time it is:

```
18.25.04

06:25:04 PM

6:25 PM
```

This class not only offers some utility methods that you can put to use in your own applications, but it makes use of the Format class and its subclasses in the java.text package (such as SimpleDateFormat). I recommend checking out the API documentation for these classes to find out about more usage options.

Chapter 28. USING TIMER TASKS

DO OR DIE:

- Start a task to run repeatedly at a certain interval
- Start a task after a specified delay
- Start a task to run at a specific time

This topic elucidates the use of tasks, made available to us via the java.util.TimerTask API. This is a wonderfully practical and easy-to-use API.

Using TimerTask to Do Stuff at Intervals

If you've ever used a UNIX cron job, you're used to needing to do stuff at timed intervals. Say you want to gzip the log files and move them in an archive every night at 3 a.m. Or maybe you need to start a process that will go out and check a server for some kind of update, and you want to check it every hour.

I once wrote an application that received user signups, and wrote them to a file. There were a lot of signups every day, but there wasn't a need to act upon them immediately. So the client asked that every hour we would do some junk to the data and then transfer it over to their AS/400 via FTP. Then they would do some junk to the data over there, and my app would check every couple of hours to see if they had written out the new file, and if they had, fetch the whole thing. So, anyway, if you need to do something repeatedly, or start doing it after a delay, java.util.TimerTask is the ticket.

We'll look at both ways in the upcoming pages.

Scheduling a Task to Start After a Delay

There are a few different ways to use the TimerTask to schedule and repeat jobs. You can specify to delay the start of a task using a millisecond delay, or by specifying a date.

DelayTask.java

```
package net.javagarage.tasks;

import java.util.Timer;
import java.util.TimerTask;

/**
 * Uses java.util.Timer schedules the task to start
 * running three seconds after it's instantiated.
 * @author eben hewitt
 */
public class DelayTask {
//hold it in a member so everybody can get to it
protected Timer timer;

//public constructor accepts the
//number of milliseconds to delay before running
//the task
public DelayTask(int msDelay){
timer = new Timer();
//pass the schedule method the task that you want
//to start, and the delay
timer.schedule(new PrintOutTask(), msDelay);
}

//using inner class to get to the timer member
//make it extend TimerTask
class PrintOutTask extends TimerTask{
//we must implement the run() method
public void run(){
//do the work here...
System.out.println("Task is doing some work...");
//terminates and discards this timer
timer.cancel();
System.out.println("All done.");
}
}

public static void main(String[] args) {
System.out.println("Scheduling task...");
new DelayTask(3000);
System.out.println("The task has been scheduled.");
}
}
```

The output initially is

```
Scheduling task...
The task has been scheduled.
```

And then after a three second delay, the following is printed:

```
Task is doing some work...

All done.
```

So the main thing that you need to do to schedule a task is as follows:

- Create a class that extends TimerTask.
- In that class, implement the inherited run() method, and do your work in there.
- Create an instance of the Timer class. This starts a new thread.
- Create an instance of the class that extends TimerTask (here, called PrintOutTask, because his job is to print out statements).
- Schedule the task to start.
- When you need to stop the task, call the timer's cancel() method.

You can schedule a task to begin after a delay, but you can also schedule it to start at a certain moment in time. Let's try that now.

Scheduling a Task to Start at a Certain Moment in Time

In the following code, we do essentially the same thing that we did earlier, but this time we specify exactly when we want it to run. You do this by using a different constructor of the TimerTask class.

StartAtTimeTask.java

```
package net.javagarage.tasks;


import java.util.Calendar;

import java.util.Date;

import java.util.Timer;

import java.util.TimerTask;


/**

 * <p>

 * Starts a task at a particular date/time.

 * <p>

 * Note that if the scheduled execution time is

 * in the past, the task will start immediately.
 * @author eben hewitt
 */
public class StartAtTimeTask {

public static void main(String[] args) {
int delay = 2000; //2 seconds
int period = 5000; //5 seconds

//Get the Date corresponding to 3:58:00 pm today.
  Calendar calendar = Calendar.getInstance();
  calendar.set(Calendar.HOUR_OF_DAY, 15);
  calendar.set(Calendar.MINUTE, 58);
  calendar.set(Calendar.SECOND, 0);

Date scheduledDate = calendar.getTime();

System.out.println("Task scheduled for " +
           scheduledDate);

Timer timer = new Timer();
timer.schedule(new TimerTask() {
public void run(){
System.out.println("The task ran at " + new Date());

}
}, scheduledDate);

}
}
```

The output shows that the task ran at a different time than the task was scheduled for, because that time was in the past. Otherwise, it would have printed 15:58:00 as expected.

```
Task scheduled for Fri Jan 02 15:58:00 GMT-07:00 2004
The task ran at Fri Jan 02 15:59:08 GMT-07:00 2004
```

Note that the program will continue running after the job is scheduled, and will keep running. We don't call the cancel() method. That's because normally the reason you schedule a task is so that the scheduling thread can be alive to know when it is supposed to do its work.

Scheduling a Task that Runs Repeatedly at a Given Interval

As with my FTP application mentioned previously, you often need to check for a status or get an updated file or post some new data at a given interval, say every hour. The following code shows how to do this.

RepeatingTask.java

```
package net.javagarage.tasks;

import java.util.Timer;

import java.util.TimerTask;


/**

 * <p>

 * Demonstrate how to use a Timer that

 * causes some code to execute repeatedly

 * at a given interval.

 * @author eben hewitt

 */

public class RepeatingTask {


public static void main(String[] args) {


int delay = 2000; //delay for 2 seconds

int interval = 60000; //repeat every minute


Timer timer = new Timer();


timer.scheduleAtFixedRate(new TimerTask() {

public void run() {

// This is where you do the work

System.out.println("Current date and time is: " + new
java.util.Date());

} //end run method of our anonymous TimerTask

}, delay, interval); //end call to scheduler


}

}
```

When I run it, it outputs the following, and would keep outputting forever until the VM is shut down or something more colossal happens like the world blows up, which has the effect of shutting down the VM.

```
Current date and time is: Fri Jan 02 14:43:09 GMT-07:00 2004

Current date and time is: Fri Jan 02 14:44:09 GMT-07:00 2004

Current date and time is: Fri Jan 02 14:45:09 GMT-07:00 2004
```

Here are all of the overloaded Timer methods that are available:

```
schedule(TimerTask task, long delay, long period)

schedule(TimerTask task, Date firstTime, long delay)

schedule(TimerTask task, Date time)

schedule(TimerTask task, long delay, long period)

scheduleAtFixedRate(TimerTask task, long delay,

    long period)

scheduleAtFixedRate(TimerTask task, Date firstTime,

    long period)

cancel()

purge()
```

Stopping a Timer Thread

As we have seen, timer tasks will run forever if you don't shut down the thread. So there are a few different ways we can stop a timer task that's running.

1. Unplug the machine.
2. Call System.exit(0), which shuts down all of the threads in an app.
3. Create the timer as a daemon thread. Doing so will automatically stop the thread once there are only daemon threads left executing, and not user threads. See the "Creating a TimerTask as a Daemon" section later.
4. After all of the tasks have finished executing, remove all of the references to the Timer object. This is not a great way to do this if you need specific control over the cessation of this task, because when it is garbage collected, the thread terminates. But you aren't guaranteed of knowing when exactly that will be.
5. Call the Timer object's cancel() method. That method terminates the timer and throws away any tasks that are currently scheduled. After you cancel a timer in this manner, no more tasks may ever be scheduled on it.

Now that we see the different ways to do it, let's look at an example that offers us the most control, which is a good thing: calling the timer.cancel() method.

StoppableTask.java

```
package net.javagarage.tasks;


import java.util.Timer;

import java.util.TimerTask;

/**

 * <p>

 * Shows that we can extend TimerTask and just

 * override the run method there to do the work.

 * The benefit of this over doing it the other

 * way is that anonymous inner classes don't have

 * access to the non-final variables of the enclosing

 * class, so we can't call timer.cancel and such.

 * Here we can call timer.cancel(), which is what

 * we're trying to do.

 *

 * @author eben hewitt

 */

public class StoppableTask {

int delay = 0; //no delay

int interval = 2000; //run every two seconds
```

```
Timer timer;

//constructor
public StoppableTask() {

//make thread a daemon thread
timer = new Timer();

System.out.println("Starting task");

timer.schedule(new SuperTask(), delay, interval);
}

public static void main(String[] args) {

new StoppableTask();

}

// this inner class guy does whatever the
//task's work is
class SuperTask extends TimerTask {

private int counter = 1;

public void run() {

if (counter <= 3 ){
//This is where you do the work, just like Thread
System.out.println("This is me, the task, doing my
    work " + " time number " + counter + "!");
counter++;
} else {
System.out.println("All done.");
//put this thread to rest
timer.cancel();
}

}
} //end SuperTask class

} //end StoppableTask class
```

Here is the output if everything falls into place:

```
Starting task
This is me, the task, doing my work time number 1!
This is me, the task, doing my work time number 2!
This is me, the task, doing my work time number 3!
All done.
```

I suggest you buy as many blues albums as you can. Start with Robert Johnson, Skip Jones, and Muddy Waters, and work your way through Blind Willie McTell, JB Lenoir, Koko Taylor, Willie Dixon, and Keb Mo'.

Creating a TimerTask as a Daemon

This is easy and fun and it has a purpose. My favorite kind of thing. If you need to run a task, and then stop it after it has run, you need to stop it manually as we have seen. Except in one case: If you create the timer as a daemon, your task's thread will stop automagically when it realizes that there are only daemon threads left (and no user-invoked threads).

Here's how we do it:

```
new Timer(true);
```

I told you it was easy.

Why would you want to schedule a thread as a daemon thread? Well, think of a background process. If you are going to need this task to schedule repeated maintenance of the app, that you need to do throughout the application's life, this is a good choice. Think "garbage collector."

Creating a Timer for a Swing GUI Application

These timers are terrific, but they are meant for executing the kinds of tasks we mentioned—performing maintenance or doing some kind of transfer or other programmatic function. There is a different, related timer class for use in Swing applications. Don't start to panic because now there's even more about this when you didn't really care that much to begin with you just wanted a little timer and now you have to read all this junk but don't worry it will be okay. The functionality that is provided by the Swing timer and the java.util.Timer class are pretty much the same. There are a couple of differences.

- The Swing timer is useful if you are building a Swing app and don't need a bunch of timers. The util Timer class is scalable up to thousands of tasks, whereas you don't want to use the Swing timer for more than oh 25 or so.
- The main difference in terms of implementation is that Swing timers can schedule tasks to run on your GUI's event thread, because they all share that thread. Although you can subclass java.util.TimerTask and call the EventQueue.invokeLater() method to replicate this functionality, it is built in with the Swing timer.

Here's how you do it:

1. Create an object of type javax.swing.Timer.
2. Register an action listener on it.
3. Start the timer using the start() method.

The following code creates a timer and starts it up. After the elapsed delay, an action event is fired every delay (here, every two seconds). The second argument to the Timer constructor represents the guy who will do the work—an ActionListener.

```
int delay = 2000; // two seconds

 ActionListener worker = new ActionListener() {

       public void actionPerformed(ActionEvent evt) {

//do some task

       }

   };

 new Timer(delay, worker).start();
```

This event firing business will continue until someone puts it out of its misery by calling stop(). On the other hand, if you want it to execute only once, you can call the timer's setReapeats(false); method.

Chapter 29. APPLETS

DO OR DIE:

- Check out how to make applets
- Check out how to view applets
- Check out some gossip about applets

Back in 1995, everyone loved applets. Now, everyone hates them. Some computer science courses still start their Java courses with applets. Although that made sense 10 years ago, it doesn't make sense now, and, alas, the applet is relegated to little more than a glorified footnote here.

In this topic, we will look at how to write and use Java applets. Applets are small applications that run on the client. The early success of Java was due in large part to a successful demonstration of an applet running on a client machine via an applet container in a Web page.

There is some debate at this time regarding the usefulness of applets. They can be cumbersome to download, and the browser wars have made it difficult to deploy mission-critical applets and be sure that your users are seamlessly able to use them.

There are many very attractive alternatives to applets. Although these alternatives don't necessarily provide all of the functionality or all of the security that applets do, they often prove good enough for developers—and with many of these competing technologies offering easier development and deployment, applets have had a hard row to hoe of late.

Not-Applet Technologies

Some of these not-applet technologies include Flash rich user interfaces, Java WebStart, and of course Active X.

Rich user interfaces and client applications can be built now using Flash. The most recent version of Flash brings Action Scripting to the forefront, and features improved database access and XML processing over previous versions. Although Flash requires a browser plug-in too, and is arguably not easier to write than an applet is, Flash applications on the client are becoming more popular and useful, especially when integrated with the server-side Flash Gateway for Macromedia JRun or CFMX.

You can also execute ActiveX controls if you are in a Microsoft environment. These are commonly used to support such business as the display of Crystal Reports that are dynamically generated and printed to the browser.

A very interesting technology for executing full-blown programs on the client is called Java WebStart. It comes with Java 1.5 and is similar to applet technology, except that it doesn't require a browser to run in. You can write complete, full-blown Swing applications that can gain access to the local file system and do more than what is allowed in applets, and the apps will execute as if they are local. This technology bridges both worlds: the ease of maintenance and distribution that the Web affords, and the sophistication that desktop apps afford.

Applets did start the excitement surrounding Java, then moved out of the limelight as Java moved to the server. Although servlets had been available for a couple of years, they are tedious to write, and are poor choices for pages that need to write out to the browser for reasons we will see later. JavaServer Pages were introduced in 1999 and quickly garnered a lot of the interest previously reserved for applets. In an interesting twist, there is a significant trend by now for moving complex application functionality back onto the client. Java WebStart is one answer to that cry for client-side power. We'll have to wait for the *More Java Garage* book to get more of the low down on WebStart though.

Applet Basics

Applets are regular Java classes that extend Applet. Because of this, you can do just about anything that Java allows you to do from within an applet that runs in its own space on the client. There are, however, key restrictions. For example, you are not allowed to open a socket connection from inside your applet, except one back to the same server from which the applet came. Additionally, you cannot access the local file system. Beyond constraints such as these, applets are very powerful.

There are a few steps to getting an applet running on a client.

- Write a Java class that extends the java.applet.Applet class or javax.swing.JApplet class and imports the packages you need.
- Write an HTML page that calls the applet using the HTML <object> tag or <applet> tag.

Using the HTML <object> tag, you specify the size and location of the applet on the page, as well as the class file to be used. The browser then loads the plug-in, which contains its own Java Virtual Machine.

In the beginning, you could only view an applet by viewing the Web page in Sun's HotJava browser. This browser is currently in version 3.0 and is still available for free download. However, it is not a full-featured browser, it is slow, and it has little support for the kinds of developments in the language (such as CSS) that users have come to expect.

FRIDGE

You should already have the plug-in installed. It comes with the JRE. If for some reason you don't, you need to download it from www.java.com to execute the examples in this section. If you do have it installed, you can open the plug-in outside a browser and review the options it makes available to you. On Windows machines, you access it by double-clicking the icon in the Control Panel. You can also try opening Internet Explorer and opening the Tools menu. You should see "Sun Java Console" there.

Applets moved into the big time when they were adopted by Internet Explorer and Netscape browsers. Using a virtual machine embedded in the browser itself, applets could be embedded in a Web page using the HTML <applet> tag. As of HTML 4.0, the W3C has deprecated this tag in favor of <object>, which is more general. Note that older browsers may not recognize the <object> tag. Browser manufacturers have not updated their Java Virtual Machines for a version later than Java 1.1. This poses obvious difficulties to developers interested in deploying applets with functionality defined in later versions of Java.

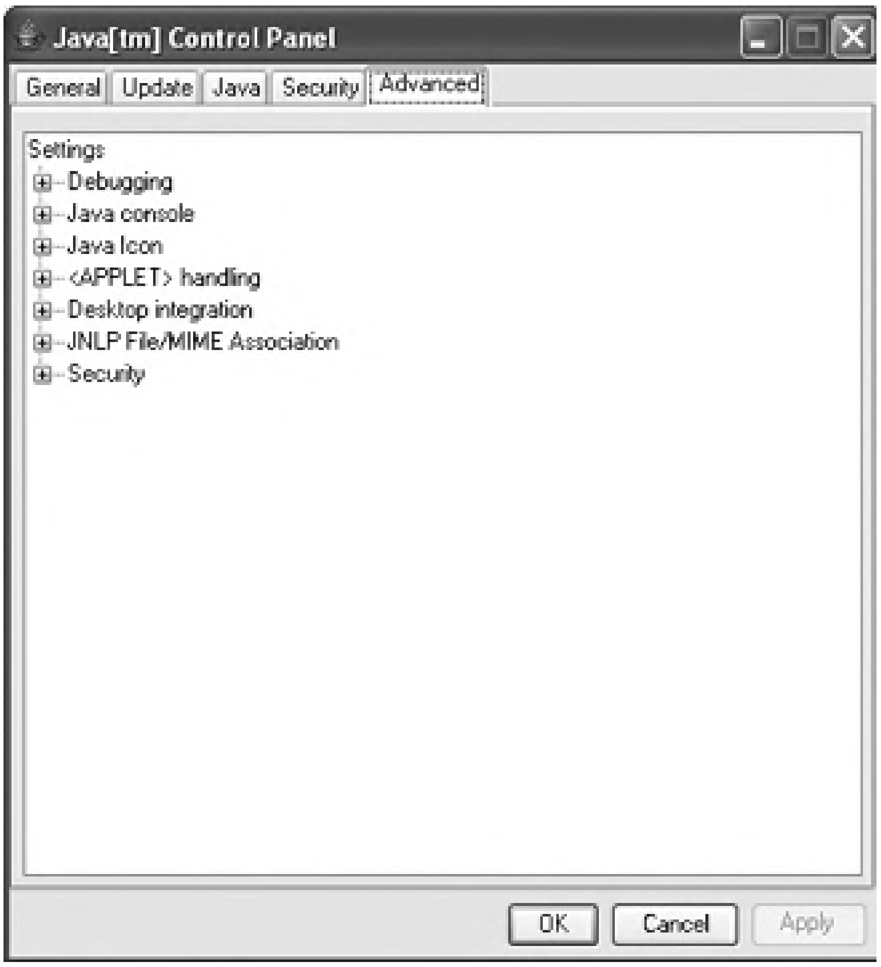
So, Sun devised a plan to propagate use of applets without being at the mercy of Netscape and Microsoft to keep up-to-date. This plan became the Java plug-in. You are likely familiar with this plug-in if you have used the <cfform> controls <cftree>, <cfgrid>, and so forth in ColdFusion 5 or later. This 5MB download allows both Internet Explorer and Netscape to execute applets using this runtime, which should always be up-to-date. Another benefit to the plug-in is added user control over the execution environment. The plug-in allows users to switch between different versions of Java Virtual Machines, for instance.

Eventually, Sun determined that limiting the plug-in to working with the newer <object> tag was dissuading developers from using applets. As of now, the fate of applets is admittedly up in the air.

If you run an applet, you should see the small Java coffee cup logo in your system tray. This indicates that an instance of the Java applet plug-in is running. You can right-click on this icon and choose Open Console to view information regarding your running applet. If the applet has errors, they will show up here, which is good to know for debugging.

The other option of interest in the Java plug-in is the Control Panel, as shown in Figure 29.1. Click Open Control Panel to bring it up.

Figure 29.1. The Java applet control panel offers a way to tinker with network settings and automatic updates.



Settings
- Debugging
- Java console
- Java Icon
- <APPLET> handling
- Desktop integration
- JNLP File/MIME Association
- Security

Here you can view your security settings and fine tune how applets are run in your browser.

There are basically two options for writing applets. You can extend java.applet.Applet. You can also choose to write applets using Swing components, which are sophisticated GUI elements located in the javax.swing package. To write applets using Swing components, you instead extend javax.swing.JApplet.

Differences Between Extending JApplet and Applet

You can use either one. Because JApplet actually extends Applet, much of the functionality is identical. There are a few differences, however.

With JApplet, you have access to the content pane, which can be called using getContentPane(). If you have a content pane of your own (such as a panel) that you want to replace it with, you can call setContentPane(). When you add components to your applet, you add them to the content pane, not the frame.

FRIDGE

Dude. Make sure you write an HTML page that can display your applet. That is the purpose of applets—to be called from within HTML. Just because it is called appletviewer doesn't mean you pass it the name of the applet class. Pass it the name of the HTML file that references the applet class. I know I just said it, but boy does that happen a lot.

Also, you get the Metal look and feel by default. If you are on Windows, for example, and want to make your applet look like Windows, you need to manually switch to Native look and feel.

BorderLayout, not FlowLayout, is the default layout.

Lastly, you use the paintComponent() method (not paint()) to render your text and graphics.

Viewing Applets

There are two ways to view an applet you write: via Web page and with the appletviewer utility that ships with the JDK. Let's write a quick applet, view it in with the appletviewer tool, and then view it in a browser. Note that both ways of viewing an applet require an HTML page. The only way you can get out of writing the HTML page for testing purposes is with an IDE. For example, if you are using Eclipse (free from www.eclipse.org) to write your Java code. Write your applet class, and then click Run... > Java Applet. Works like a champ. Of course, you need to write the HTML once you go to deploy….

Writing an Applet

Let's just write an applet and try to view it and see what comes out of the woodwork, shall we?

SimpleApplet.java

```
package net.javagarage.applets;

import java.awt.Color;
import java.awt.Container;
import java.awt.FlowLayout;

import javax.swing.JApplet;
import javax.swing.JButton;
import javax.swing.JLabel;

/**
 * <p>
 * Example of very simple JApplet.
 * @author eben hewitt
 */
public class SimpleApplet extends JApplet {

public void init() {
Container content = getContentPane();
content.setBackground(Color.PINK);
content.setLayout(new FlowLayout());
content.add(new JLabel("Hello, world"));
content.add(new JButton("Poke me"));
  }

}
```

This applet doesn't do much at all. You see a label, you see a button, you press the button, nothin' happens. Put whatever code you want inside there. My purpose is to show you that you need to extend JApplet or Applet, and then write some stuff in the init() method body.

Now that the writing is out of the way, let's view that sucker. If you're using Eclipse or another IDE, you know what to do—cheat. If you want to rack up a little extra karma, you can write the HTML file now and view it with the appletviewer tool that ships with the JDK. Here's a file we can use.

ShowSimpleApplet.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>
<head>
<title>Applet Test</title>
</head>

<body>

<applet align="center"
    code="net.javagarage.applets.SimpleApplet.class"
    width="200" height="100">
</applet>


</body>
</html>
```

Notice a couple of things here. First, we use the .class file extension. Second, we use the fully qualified class name, which means including package names.

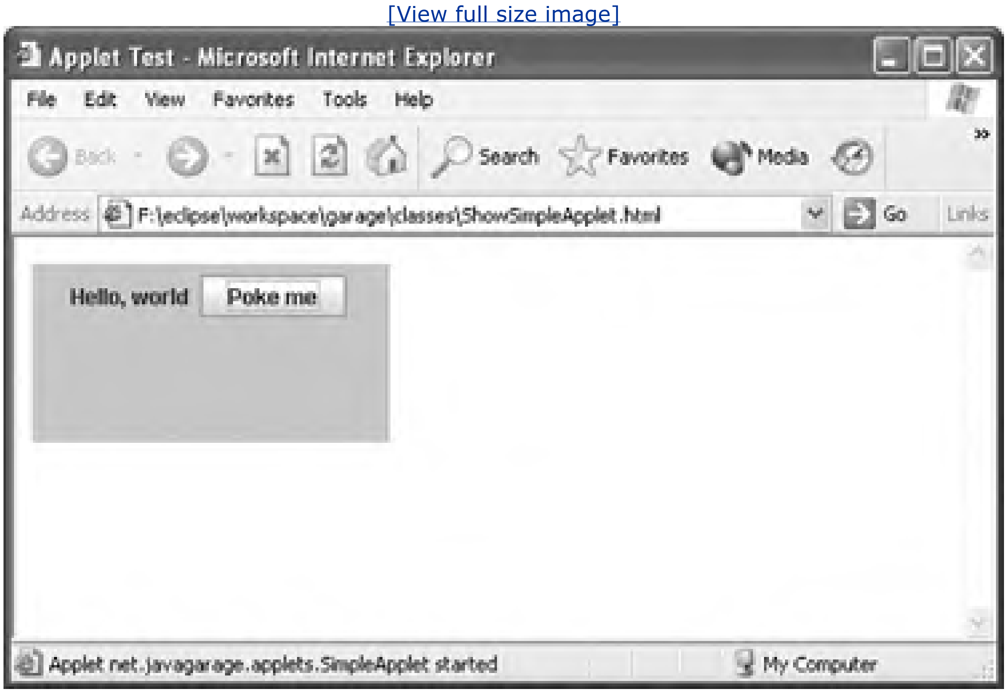
Now, you can just open your browser and point it to that HTML page to view the applet. Or, you can be fancy and use the JDK appletviewer tool at a terminal window, like this:

```
>appletviewer
F:\\eclipse\\workspace\\garage\\net\\javagarage\\applets\\Show
SimpleApplet.html
```

This is the full path to the file on my Windows XP machine. Note that you need not include the drive. Adjust the command for the location of your HTML file. This should launch the viewer program and display the applet, as shown in Figure 29.2.

Figure 29.2. Our little applet is all grown up and out in the world.

[View full size image]


Applet Test - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Search
Favorites
Media
Address
F:\eclipse\workspace\garage\classes\ShowSimpleApplet.html
Go
Links
Hello, world
Poke me
Applet net.javagarage.applets.SimpleApplet started
My Computer

If something went wrong and your HTML page just shows a gray box with a little red X in the upper-left corner, either there is a fatal error of some kind or (most likely) your applet cannot be found. You can check the path to the file and try again (make sure you included the fully qualified name of the applet class, including packages)—OR, you can right-click on the applet and select Open Java Console. This will show you the stack trace leading up to the problem. It is very likely that one of the following occurred:

1. You have specified the path to the applet incorrectly.
2. You have left off the .class extension from the name of the applet class in the HTML.
3. You have placed the HTML file in a place where it can't find the applet class: remember that the HTML should be at the root of your classes directory (the one that has "net" in it)—that's where java executes from.

Notice what I had to do here. If you look in the address bar, you'll see that the HTML page is referenced from within the classes directory. That's because that's where the class is. If you make a new file in your editor, it is likely to be sitting in the source directory—not the classes directory. Don't tell me that you have your source code in the same directory as your classes—that would be punishable by something pretty bad. A whole week without playing Road Blaster or Gauntlet, for example.

Applet Life Cycle Methods

There are several methods that allow you a little control over what happens during the life of an applet. Here they are in brief.

```
public void init() {

 // Called once by the browser when it starts the applet
}
public void start() {
   //called when the page that contains
      //the applet is made visible
}
public void stop() {
   //called when the page that contains the applet
      //is made invisible
}
public void destroy() {
  //called when the browser destroys the applet
}
public void paint(Graphics g) {
   //called when the applet is repainted
}
```

A Few Applet Tricks

Here are a few neat things that you can do within your applet, including some that might be downright necessary.

Showing a Message in the Browser's Status Bar

The status bar of the browser typically will display messages in the bottom-left corner. This feature is so retro mid-90s, it is impossible to resist.

```
applet.showStatus("All your base are belong to us. Ha ha ha");
```

Nuff said.

Making the Browser Visit a URL

You can redirect the browser using the showDocument() method, like this:

```
try {
     URL url = new URL(getDocumentBase(),
     "http://www.example.com/some.jsp");
     applet.getAppletContext().showDocument(url);
   } catch (MalformedURLException e) {
//deal
   }
```

Passing Parameters into Applets

You can pass a parameter into an applet from the HTML page that displays the applet. You do so using the applet.getParameter(String s) method. Then, you pass the parameter in via a special HTML tag called <param>.

Here is the code to get the parameter:

```
public void init() {
    String parameterName = "stockPrice";
    String s = applet.getParameter(parameterName);
}
```

This code will return null in the event that no parameter named "stockPrice" can be found.

Here is the HTML required to pass it in:

```
<applet code=MyApplet width=200 height=200>
   <param name=stockPrice value="$212.95">
</applet>
```

Note that if you are using a servlet container or some other app server like ColdFusion or ASP.NET or PHP, you can set the value of the parameter dynamically—that is, at runtime.

Loading Image Files in Applets

You can use image files inside your applets.

```
Image image;
public void init() {
   //get the image
   image = getImage(getDocumentBase(),
          "http://www.example.com/some.gif");
}
public void paint(Graphics g) {
   //draw image in upper-left corner
   g.drawImage(image, 0, 0, this);
}
```

Playing Audio Files in Applets

You can cause your applet to play audio files that are available via URL as well.

```
public void init() {
        // Load audio clip
        AudioClip ac = getAudioClip(getDocumentBase(),
"http://www.example.com/some.au");
//play the audio
        ac.play();

        //stop audio
        ac.stop();

        //play audio continuously
        ac.loop();
    }
```

With these easy methods, you can supply buttons to your applet that respond just like stereo controls.

We have admittedly skimmed over applets here (sort of). That's because they simply do not hold the attractiveness they once did. Server-side apps are more powerful in many ways. There is Java WebStart (which allows you to run Web-distributed programs locally with automatic updates), and applets have proven slow and difficult to manage. But they have a certain important place in Javaland, so they should be discussed. With what we've covered here, you certainly have enough to get started. The real appeal of applets of course is in the applications that you write and distribute as applets. Much of what we cover throughout the rest of this book could be created as applets knowing what we have gone over.

If you are eager to see some pretty sophisticated applets, try visiting www.java.com or http://java.sun.com/products/plugin/1.4.2/demos/plugin/applets.html. Sun has some very cool examples here. They probably will put a 5.0 page up, but it isn't there as of this writing, so check back for it.

Chapter 30. FRIDGE: BIG DADDY FLAPJACKS

In an alarming change of plans, I'd like to have breakfast now. I'm not sure what kinds of questions inspire you. Maybe questions about the size of the universe, or how life is started, or the salary you'll get writing Java applications. Probably my favorite question is, "What's for breakfast?"

Let's just take a little break before going back to that hard application writing work. I'd like to have breakfast. If you would care to join me, we're having pancakes. And why not? I think everybody loves pancakes. Unless you are a vegan; then maybe you wouldn't like pancakes at all, and you will have to drink Diet Mountain Dew instead for your breakfast. Though I could be wrong about that.

Here is the recipe, which I really think you'll like. I'm telling you: try these. They're easy to make, and they really taste delicious. If you don't like to cook, maybe you can just chat with your sweetheart while you mix them up. That's what I do.

Ingredients:

1 egg

1 cup buttermilk

2 tablespoons vegetable oil

1 cup flour

1 tablespoon sugar

1 teaspoon baking powder

1/2 teaspoon baking soda

2 good pinches of salt

Grease your griddle. I just spray Pam on it for one second. Then start your griddle over a hot flame so it's ready when your batter is ready. In a mixing bowl, beat the egg. Then add remaining ingredients, stirring completely after adding each new ingredient.

Here's the secret to this recipe: let the batter sit for five minutes before you pour it. Then, when you're ready, pour batter onto grill using maybe a 1/4 cup measuring cup. Flip the flapjacks when they start to bubble on top.

This recipe makes eight good size Big Daddy Flapjacks. Enjoy immediately with real maple syrup and a hot cup of java. Now you'll be well nourished and ready for another exciting day of programming.

Chapter 31. USING SYSTEM AND RUNTIME

DO OR DIE:

- Great lamb chop ideas—just in time for the holidays!
- Take advantage of insider tips and tricks for making millions playing the market
- Investigate the Runtime class
- Get the static methods of the System class to do some work for a change
- See what the Java Runtime class can do

There is a class called System. It represents the underlying host for your application. You are doubtless familiar with it from your endless calls to System.out.println(). Well, friend, it does more. A lot more. And the Runtime class represents the Java Virtual Machine to your application. These two classes can be used in different ways to improve usability, fine tune your apps, improve performance, and make you the life of every party. Let's have a little look-see.

Using the System

The System class cannot be instantiated. But you can use its fields to get a hold of the standard out, standard in, and standard error IO streams. These streams represent how to print to the console, how to receive the input the user enters at the console, and the venue for printing error data.

The System class features a number of utility methods that you can use to perform miscellaneous tasks. Let's look at the methods, in order of sexiness.

Printing Data to the User

We have seen this a number of times already. I include it here for pure pedantry (and because there is actually some stuff that we might find useful). To print to the standard output stream, which is the console the user can view when executing a Java application, write thusly:

```
System.out.print("Some string");
```

The sister of the print method, println(), prints the text and as an added bonus appends a system-specific newline character. Now, you do not have to pass a String into the print() or println() method. These methods are both overloaded to accept a String, each of the primitive types, a char array, and an Object. This is where it gets interesting.

When an Object is passed to the method, its toString() method is automatically invoked. That means we can do this:

```
Object obj = new Object();

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

Here is the result:

```
java.lang.Object@119c082
```

The implementation of toString() in Object prints a String equal to this call:

```
getClass().getName() + '@' +

Integer.toHexString(hashCode())
```

It is recommended, however, that you always override toString() in your objects. Remember that every Object is ultimately a descendant of java.lang.Object, but they are the immediate descendants of all other superclasses. So if you don't override it, you'll get either the same useless information as shown in the preceding when someone—or someone you love—calls your Object's toString() method, explicitly or implicitly.

Determining Runtime Speed

The first method we'll look at is currentTimeMillis().

```
long System.currentTimeMillis()
```

This returns the current time in milliseconds, as follows:

```
Time: 1068434679594
```

This number, a primitive long, is the total number of milliseconds between midnight, January 1, 1970, and the current time. Although it may seem humorous in a department meeting to tell your boss that you'd be happy to meet him in 894,957 milliseconds to discuss your promotion, you might be the only one laughing. There is a very common use for such information: You can use it to figure out how long it is taking your code to run.

Say you've got a complex operation and you're concerned that it's taking too long. You can figure that out with some fancy tools that cost thousands of dollars, or you can trust old reliable: currentSystemMillis().

To help us flush out the bottlenecks in our application, we can get the time before and after chunks of suspect code, and subtract to determine total execution time. The code in SystemDemos.java shows this.

SystemDemos.java

```
public class SystemDemos {

public void testCurrentTimeMillis(){

    //some poorly written code that takes a while

    System.out.println("Bad Execution time: " + doBadCode());
    System.out.println("Improve Execution time: " +
        doImproveCode());
}

private long doBadCode(){
    long before = System.currentTimeMillis();
    for (int i = 0; i < 100000; i++){
        String s = new String("Object " + i);
    }
    long after = System.currentTimeMillis();
    return after - before;
}

private long doImproveCode(){
    long before = System.currentTimeMillis();
    String s = "";
    for (int i = 0; i < 50000; i++){
        s = "Object " + i;
    }
    long after = System.currentTimeMillis();
    return after - before;
}

public static void main(String[] args) {
    SystemDemos sys = new SystemDemos();
    sys.testCurrentTimeMillis();
}
}
```

Here is our result:

```
Bad Execution time: 200
Improve Execution time: 60
```

Note that this way of measuring execution time is not precise enough for applications that require serious performance. There is time taken to perform the method call that is not accounted for, and threading issues that are not accounted for, as well as random system events, such as spikes occasioned by other apps running. This method is good enough to get a rough outline. For more serious stuff, you can get a profiler to precisely measure performance.

In SystemDemos.java, we inspect two different methods: one that constructs strings in the naughty way (String s = new String("x"));, and another that constructs them the happy way. In each method, we create 100,000 different String objects and check how long it takes. There are only two differences between the methods. As you can see, redeclaring the String object in each of 100,000 iterations of the loop is an expensive proposition. It takes more than three times as long to achieve the exact same result. Those milliseconds could be precious in a complex application, and this is a simple, quick way to get a rough idea of whether a bit of code is going to end up a problem child.

Exiting an Application

You can announce your departure from an application using the exit(int status) method. This will shut down the currently running Java Virtual Machine by calling the exit method in the Runtime class, making the following two calls equivalent:

```
System.exit(0);

//or...

Runtime.getRuntime().exit(0);
```

By convention, an exit value of 0 indicates normal system exit, whereas any other number (commonly -1) indicates a deleterious state.

Forcing the Garbage Collector to Run

You can't.

Suggesting Kindly That the Garbage Collector Run

This you *can* do. The garbage collector runs in a separate thread from the thread of execution, reclaiming space here and there when it finds objects that can no longer be referenced, or other unreachable code.

The garbage collector has been around the block. She's seen objects come and go, every day throughout the years. And she's not about to take any lip from some upstart crow telling her how to manage her memory.

To suggest that the garbage collector run, which does not guarantee that it runs, try tempting her with this:

```
System.gc();
```

Note that, like other methods in the System class, this is the effective equivalent of calling

```
Runtime.getRuntime().gc();
```

This means that the Java Virtual Machine will have made its "best effort to reclaim space" (API documentation). But if the VM is busy, its best effort may not leave anyone thinking, "Whoa, that was really a good effort." In 99 cases out of 100, however, the call to gc() can be helpful, assuming that you need the memory back immediately, because the garbage collector will pick up method local variables very soon after the method returns.

Here is an example that demonstrates.

```
private void testGC(){

System.out.println("Free mem kb before bad code: " +

Runtime.getRuntime().freeMemory() / 1024);

for (int i = 0; i < 100000; i++){

String s = new String("Object " + i);

}

System.out.println("Free mem kb AFTER bad code: " +
Runtime.getRuntime().freeMemory() / 1024);

System.gc();

System.out.println("Free mem kb after GC call: " +

Runtime.getRuntime().freeMemory() / 1024);

}
```

And the result:

```
Free mem kb before bad code: 1864

Free mem kb AFTER bad code: 1493

Free mem kb after GC call: 1898
```

Notice that the garbage collector sees that all of these String objects are, as my friend John from Texas used to say, "a big show about nothing," and knows it can remove them. The call to the garbage collector freed up half a megabyte of memory. Running this program with the call to System.gc() commented out yields this result:

```
Free mem kb before bad code: 1864

Free mem kb AFTER bad code: 1493

Free mem kb after GC call: 1493
```

Executing an External Application in Java

There are many times that you might like to launch an application that is running on the user's machine. For example, an external Web browser could be used to display help pages. There is a way to do this that is as easy as running it yourself from the command line. All you have to know is its name on the system. To make your Java application open Windows Explorer, write

```
try {

Runtime.getRuntime().exec("explorer");

} catch(IOException ioe){

System.out.println(ioe.getMessage());

}
```

You have to catch or rethrow the IOException that this call can occasion. Note that we just pass in the system name, not including the file extension.

Passing Arguments to an External Application

Not only can we start an external application, we can pass arguments to it. Instead of allowing Windows Explorer to choose where to open, let's make it bend to our will by passing in the name of the directory in which we would like it to open: the Eclipse IDE workspace directory. We need to make two changes.

```
try {

String[] arg = {"F:\\eclipse\\workspace"};

Runtime.getRuntime().exec("explorer", arg);

} ...//catch statements
```

The first thing we've done is we've created an array of Strings with one value, which represents the arguments to our external app. Second, we called the overloaded exec() method that accepts a String array of arguments (sound like any method you're familiar with?), and passed that sucker in.

Java will run the application if it can. Here, we start Windows Explorer, which will open in the Eclipse workspace directory.

This functionality is sometimes employed to start a browser and display a Web page in it. Note that you don't need to know the path to the program you want to launch. If you can type it in a terminal and make it run, it will work here. This functionality is very similar to doing this in Windows: Click Start > Run. Type cmd to get a prompt. Enter explorer F:\eclipse\workspace (or whatever path you want to open). You're good to go.

Interacting with the User

There are a few different ways to interact with the user from within an application. For desktop software, you use classes in the Swing packages javax.swing. There are servlets and JavaServer Pages that allow you to interact from a Web server. However, there are many occasions on which you just want something simpler. It can be difficult to set up Swing applications with sophisticated layouts, events, and handlers. There are many times too when a console application is not only fine, but is the best choice for interacting with the user. In these cases, you turn to the System class.

Reading User Input from the Console with System.in

When you want to print something out to the console, you use the print() or println() methods of the System.out object. To read user input from the console, get the static System.in object, which is of type InputStream. As with the OutputStream provided by the out object, this stream opens automatically when your program starts. So all you have to do is read from the stream. Sounds straightforward enough, right?

Well, not really.

We know that dealing with IO is often a little tricky, just because there are so many ways to do it. What type of field is InputStream? It is abstract, representing the superclass of all classes that accept data as byte arrays. That means that we need to wrap it in a class that defines a method for accepting bytes of input as characters. However, we're talking about a fairly costly operation: using a byte array reader to read data, then converting it to characters with an InputStream reader, returning the converted text constantly. What we need is a way to do all of that, but make a little temporary storage area, a buffer, for the character data. The most efficient way to do that is to wrap our InputStreamReader with a BufferedReader. So the call that gets data from System.in looks like this:

```
BufferedReader in
  = new BufferedReader(new InputStreamReader(System.in))
```

Now we have a reference that we can use to read each line of user input. To do that, we use a loop that checks for the end of the input, like this:

```
while((line = in.readLine()) != null) {
//strip spaces out just to show an operation happening
line = line.replaceAll(" ", "");
}
```

If you are going to do this sort of thing (use runtime.exec()),you really should read the next section after the Calculator code, called "Thread Issues with Runtime.exec()."

Note that if you want to compile the preceding code in a class, you will need to catch the exceptions thrown by the reader classes.

Toolkit: A Simple Calculator

This program demonstrates how you might write a simple calculator in Java. It incorporates a number of the programming constructs we've covered so far in this book, and it could be usefully incorporated into other applications. The main thing it does that is of interest here is that it uses the standard input and shows how to read user-entered data in a complete, working app.

It demonstrates using File I/O (BufferedReader and InputStreamReader), using regular expressions, and exception handling.

```
package net.javagarage.apps.calculator;


import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.util.regex.Matcher;

import java.util.regex.Pattern;


/**<p>

 * Demos a working calculator, usable from the

 * console. The advantage is that it shows, in

 * a small program, how to fit different pieces

 * together.

 * <p>

 * More specifically, it shows how to read in

 * data the user types on the console, and how

 * to parse strings in a real-world way.

 * <p>

 * Limitations include the fact that a legal

 * expression is constituted by two operands

 * and one operator; i.e., we can't do this:

 * 4 * (5 + 6)

 * </p>

 * @author eben hewitt

 * @see BufferedReader, Float, String, Pattern

 **/

public class Calculator {
```

```
private String readInput() {
    String line = "";

    try {
    //get the input
    BufferedReader in = new BufferedReader(
            new InputStreamReader(System.in));

    while((line = in.readLine()) != null) {
        //strip spaces out and do calculation
        line = "result: " +
                calculate(line.replaceAll(" ", ""));
    }
    } catch (IOException e) {
        System.err.println(e.getMessage());
    } catch (NumberFormatException nfe){
        System.out.println("Error->" +
                nfe.getMessage());
        System.out.println("Enter an expression: ");
        //recursive call. Otherwise the program stops.
        readInput();
    }
    return line;
}

private float calculate(String line){
    //these will hold the two operands
    float first, second;
    String op = "";
    int opAt = 0;
    //build regex to parse the input
    //because * is an operator in regex, we
    //need to escape it.

    Pattern p = Pattern.compile("[+/-[*]]");
    Matcher m = p.matcher(line);
    //when you find an arithmetic operator in the
```

```
        //line, get its position
        if (m.find()){
            opAt = m.start();
        } else {
            //user didn't enter a valid expression
            System.out.println("usage-> Enter a valid
                      expression");
            return Float.NaN;
        }
        //get the part of the string before the operator
        //that is our first operand
        first = Float.parseFloat(line.substring(0, opAt));
        //find the occurrence of one of the operator
        //symbols
        op = line.substring(opAt, opAt+1);
        //get the number between the operator and the end
        //of the string; that's our other operand
        second = Float.parseFloat(line.substring(opAt+1,
        line.length()));
        if (op.equals("*")){
            System.out.println(first * second);
        }
        if (op.equals("+")){
            System.out.println(first + second);
        }
        if (op.equals("-")){
            System.out.println(first - second);
        }
        if (op.equals("/")){
            System.out.println(first / second);
        }

        return Float.NaN;
    }

    /**
     * You can hit CTRL + C to cancel the program
     * and start over.
```

```
 * @param args
 */
public static void main(String[] args) {
    System.out.println("Enter an expression: ");
    new Calculator().readInput();
}
}
```

Result

Here is what a sample execution of the program looks like:

```
Enter an expression:
655+99807
100462.0
65 * 876
56940.0
510 / 70
7.285714
34 - 9876
-9842.0
fake input
usage-> Enter a valid expression
```

The limitations of this program are that it doesn't handle nested expressions and it doesn't have a control for the user to stop the program; it will just run until you manually shut it down.

Thread Issues with Runtime.exec()

Stuff doesn't always come out as we plan. Sometimes things go wrong in our code. But sometimes, we just haven't done as much as we could have to plan for contingencies.

The previous examples of using the System.in stream work fine. If you need to access System.err during execution of your program, though, you have two streams coming in, and only one place to read them. You have tried to be a good programmer, understanding that sometimes bad things happen to good programs, and handle them. But this presents a special kind of difficulty. Here is the common problem.

Your program tries to read from the standard input, and then tries to read from the standard error stream. This doesn't seem like what you want, but how else are you going to do it, as your while loop processes? You can't read from both standard in and standard error at the same time, right? (You can, and should). Because your process could instantly try to write to standard error (if something goes wrong in your program before anything is written to standard in). If that happens, your program could block the on in.readLine(), waiting for the process to write its data. However, back at the ranch, the program is trying to write to standard error. It might be expecting you to read standard err and free up the buffer. But you don't do this because you are waiting for the process to write to standard error. You can see where this is headed. Deadlock.

This is when threads are really called for. Look at your program and honestly answer this question: "How many things am I trying to do here?" If the answer is more than one, ask yourself this: "Do any of these tasks require waiting for event?" If the answer is yes, that guy is a good candidate for a thread. Doing things in one thread can be more complex obviously, but often prevents your application from crashing after many pending and unresolved issues are piled up higher and higher without chance at resolution. The best-case scenario here is usually very poor application performance.

To achieve this smoother functionality, we need to do a little work of our own. Specifically, we need two classes: one to represent threads for standard input and standard error, and another to be the app that places the call to runtime.exec(). First, the mama app.

MultithreadedSystem

```
package net.javagarage.sys;

/**
 * <p>
 * Demonstrates use of threads to handle standard
 * error stream interruptions of input in order
 * to avoid deadlock.
 *
 * @see Thread, ThreadedStreamReader
 * @author eben hewitt
 */
public class MultithreadedSystem {

public static void main(String[] args) {

MultithreadedSystem m = new MultithreadedSystem();

try {
//m.doWork(new String[]{"explorer", "dude"});

//try using a program that uses the stdin, like this,
//remember that each command must be separate
m.doWork(new String[] {"cmd", "/c", "start", "java",
"net.javagarage.apps.calculator.Calculator"});
} catch (Exception e){
```

```
e.printStackTrace();

}


System.out.println("All done");

}


/**

 * This is where you would do whatever work with the

 * standard input that you want.

 * @param String[] command The name of the program

 * that you want to execute, including arguments to

 * the program.

 * @throws Exception

 */

public void doWork(String[] command) throws Exception {


//this will hold the number returned by the spawned app

int status;


//use buffers to stand in for error and output streams

StringBuffer err = new StringBuffer();

StringBuffer out = new StringBuffer();


//start the external program

Process process = Runtime.getRuntime().exec(command);


//create thread that reads the input stream for this process

ThreadedStreamReader stdOutThread =

new ThreadedStreamReader(process.getInputStream(), out);


//create thread that reads this process'

//standard error stream

ThreadedStreamReader stdErrThread =

new ThreadedStreamReader(process.getErrorStream(), err);


//start the threads

stdOutThread.start();

stdErrThread.start();
```

```
//this method causes the current thread to wait until
//the process object (the running external app)
//is terminated.
status = process.waitFor();

//read anything still in buffers.
//join() waits for the threads to die.
stdOutThread.join();
stdErrThread.join();

//everything is okay
if (status == 0) {
System.out.println(command[0] + " ran without errors.");

//you can print the values of the out
//and err here if you want
//if the result is not 0, the external app
//threw an error
//note that this is by convention only, though most
//programs will follow it
} else {
System.out.println(command[0] + " returned an error status: "
+ status);
//you can print the values of the out and
//err here if you want
}
}
}
```

The second class will extend thread, and we'll direct the output using it.

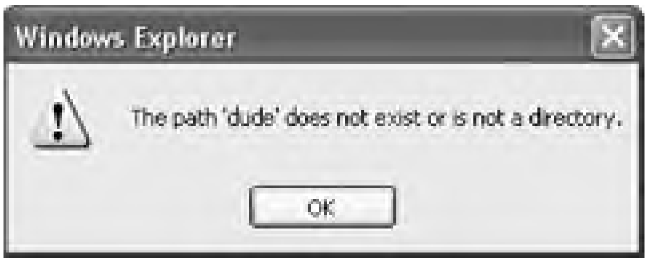
ThreadedStreamReader

```
package net.javagarage.sys;


import java.io.InputStreamReader;

import java.io.InputStream;


/**

 * <p>

 * File: ThreadedStreamReader

 * Purpose: To ensure access to standard error

 * System.err when reading a buffered stream with

 * System.in.

 * <p>

 * Note that typically in your programs you want to

 * implement the Runnable interface to do

 * multithreading, but here we are only ever going to

 * use this class for this thread-specific purpose,

 * so it is okay. It is preferable to implement

 * Runnable in general, because then you are free to

 * extend a different class (and Java does not allow

 * multiple inheritance).

 *

 * @author eben hewitt

 */

public class ThreadedStreamReader extends Thread {


private StringBuffer outBuffer;

private InputStreamReader inputStream;


//constructor accepts an input *(

//for instance, and

/**

 * @param InputStream either standard err or standard in

 * @param StringBuffer

 */

public ThreadedStreamReader(InputStream in, StringBuffer out){
```

```
outBuffer = out;

inputStream = new InputStreamReader(in);

}


//override run() to do the thread's work

public void run() {

int data;

try {

//hold character data in the buffer until we get to the end

while((data = inputStream.read()) != -1)

outBuffer.append((char)data);


} catch (Exception e) {

//tack the error message onto the end of the data stream

outBuffer.append("\nError reading data:" + e.getMessage());

}

}

}
```

Executing this application with arguments of explorer and dude gets us the error shown in Figure 31.1.

Figure 31.1. Windows Explorer starts up but cannot find this directory and returns an error status, which we capture.



Windows Explorer

The path 'dude' does not exist or is not a directory.

Our application then prints the following and exits:

```
Process explorer returned an error status: 1

All done
```

Of course, executing that program doesn't give us access to these streams, so it isn't entirely useful, but it proves in the simplest possible manner that our program does work. Which is not a bad practice.

Now, if you have the Calculator program compiled, you should be able to call it just as I do here. If it won't run, perhaps because you get an IOException: Cannot create process, error=2 or something like that, make sure that your path is set up correctly. You can check if your system can execute the Java command by opening a command prompt and typing java. If usage information prints out, you're in business. So go ahead and run it.

If everything goes as planned, our call to the Java calculator program should look like this after we're finished using it.

```
cmd ran without errors.

All done
```

I love it when a plan comes together.

Now let's look at another useful example of getting the runtime to execute something.

Toolkit: Getting the MAC Address from Your NIC

To show that there are useful things that are made possible with this business,, and to do something that's possibly useful, let's write a program that works on both Linux and Windows and calls programs that ship with each of those operating systems. Let's get the MAC address off of our network cards.

MACAddress.java

```
package net.javagarage.misc;


import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.util.ArrayList;


/**<p>

 * Uses system tools in windows or linux to

 * get the user's Media Access Control addresses

 * and print them out.

 * <p>

 * Demonstrates practical use of runtime.exec().

 * On Linux, you must be root to run it.

 * <p>

 * One shortcoming of this program is that if you
 * encounter any errors in the process, they aren't
 * registered.
 **/
public class MACAddress {
//if true, the messages passed into the log() method
//will print; if false, they are ignored
static boolean debug = true;

/**
 * Determine what OS we are on, and call a different
 * program to do our work depending on the result.
 * @return
 */
public static ArrayList getMACAddresses(){
//what regex matches the String representation
//of the address returned?
String matcher = "";

//what program is used on this OS to get the MAC address?
String program = "";

//how many places does this address String require
//us to move back?
int shiftPos = 14;

//different operating systems have different programs
//that get info about the network hardware
String OS = System.getProperty("os.name");

if (OS.startsWith("Windows")){
/* ipconfig returns something like this:
 * 00-08-74-F5-20-CC
 */
matcher = ".*-..-..-..-..-.*";
program = "ipconfig.exe /all";
```

```
}
if (OS.startsWith("Linux")){
/*ifconfig returns something like this:
"HWaddr 00:50:FC:8F:36:1A"
the lastIndexOf method will set the substring
to start 14 spaces back from the last :, which is
the beginning of the address
*/
shiftPos = -39;
matcher = ".*:..:..:..:..:.*";
program = "/sbin/ifconfig";
}
//sorry no macintosh
return getMACAddresses(matcher, program, shiftPos);
}

/**
 * Overloaded method uses the program that
 * ships with the current OS and parses the string
 * that it returns.
 * @return String The address of each
 * device found on the system.
 */
private static ArrayList getMACAddresses(String
matcher,
String program, int shiftPos){
String line = "";
Process proc;
ArrayList macAddresses = new ArrayList(2);
try {
//run the windows program that prints the MAC address
proc = Runtime.getRuntime().exec(program);
BufferedReader in = new BufferedReader(
new InputStreamReader(proc.getInputStream()));
while((line = in.readLine()) != null) {
//the matches method determines if a given
//String matches the passed regular expression
if (line.matches(matcher)) {

int pos = line.lastIndexOf("-")-shiftPos;

//add the address to the list

macAddresses.add(line.substring(pos,

line.length()));

}

}

} catch (IOException e) {

log("Error using ipconfig: " + e.getMessage());

proc = null;

System.exit(-1);

}

return macAddresses;

}


/**

 * Print out the info we parsed. These are separated

 * methods because you might want to do something

 * other than just print it out.

 * @param devices

 */

public static void printAddresses(ArrayList devices){

int numberDevices = devices.size();

System.out.println("Devices found: " + numberDevices);

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

System.out.println("Device " + i + ": " +

devices.get(i));

}

}


//convenience method for printing messages to stdout

private static void log(String msg){

if (debug){

System.out.println("—>" + msg);

}

}

//start program

public static void main(String[] args) {

printAddresses(getMACAddresses());
```

```
System.exit(0);

}


} //eof
```

The result is shown here:

```
Devices found: 2

Device 0: 00-20-A6-4C-93-40

Device 1: 00-04-76-4D-D6-B5
```

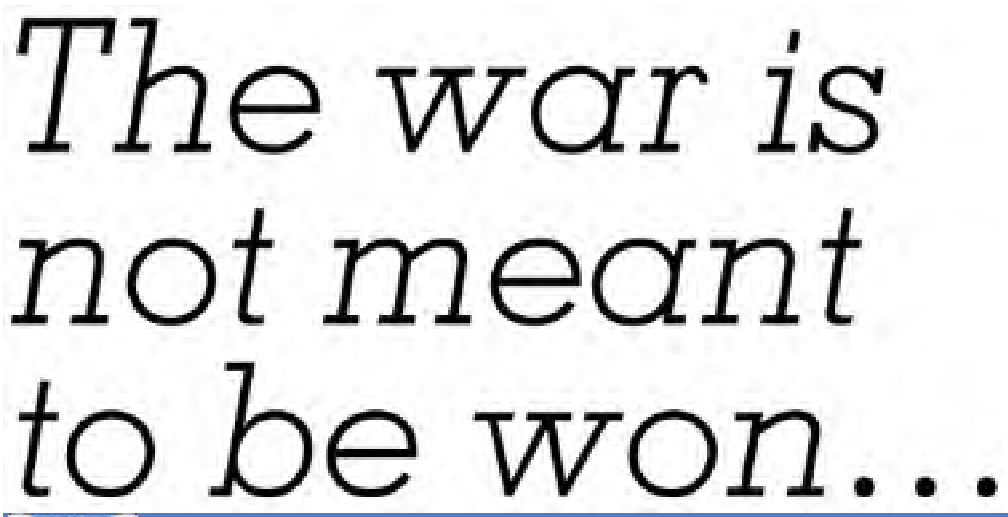
My laptop has one standard Ethernet adapter (device 1) and one wireless 802.11 card (device 0). Note that you can toggle the debug boolean to print out information regarding the current state of the program.

Determine Number of Processors on the Current Machine

This ability could come in handy if, say, you had some software product you were licensing per processor. You could use this method to help ensure user compliance:

```
Runtime.getRuntime().availableProcessors();

//returns an int.

//In my case, 1.
```

Read *1984* again.



Determine When Your Application Is About to Exit

What do you do when you want to shut down your Windows box? You click on Start. Likewise, when your application is shutting down, it doesn't stop at all. It rather starts something new. The application starts any registered shutdown threads. Only when they complete their work will the application exit. This is the case for normal termination—for instance, when you call this code:

```
System.exit(0);
```

Passing the 0 int to the exit method means, by convention, that everything is hunky dory in your application, and that you just want to end it. Typically, a non-zero number (such as -1), as you might imagine, indicates a non-normal termination.

Abnormal termination is typically unexpected, but there are times when the chances are greater that something is going to run amok in your app. Like when you pass control over to something else, such as a native library or a network resource. The user can shut down the JVM abruptly by typing CTRL + C.

If there is some work that you would like to do but the application is about to shut down, you can register a shutdown thread, known as a hook, to do your dirty work. Maybe this dirty work is like, "write a message out to a log, log the user out, make sure he isn't working with any shared document keys, or send an e-mail." Some would argue that these things are perhaps too much work to do in a shutdown hook.

For example, we could write an entry to a log file when a user starts an application, and then call the shutdown hook to write the time when he shuts it down. This could give us possibly useful metrics on the order of, "How long did Sally do any work before goofing off on the Internet for 4.5 hours?" Here's how you do it.

ShutdownHook.java

```
package net.javagarage.sys;

/**

 * <p>

 * Demos the usefulness of the

 * Runtime.getRuntime().addShutdownHook()

 * method.

 * @author eben hewitt

 */

public class ShutdownHook {


private void init(){

//register a new shutdown thread

   Runtime.getRuntime().addShutdownHook(new Thread() {
```

```
  //
  public void run() {
  //do work here
System.out.println("In shutdown hook");
  }
  });
}

private void doSomething(){
System.out.println("App started...");
}

public static void main(String[] args) {
ShutdownHook hook = new ShutdownHook();
hook.init();
hook.doSomething();
try {
System.out.println("Sleeping...");
Thread.sleep(5000);
} catch (InterruptedException ie){
System.out.println(ie.getMessage());
}
}
}
```

Here are a few final considerations regarding shutdown hooks:

- It is not guaranteed that the application will run your shutdown hooks. If the user unplugs the machine, for instance, they won't run. Also, if the process is terminated externally—via a kill process command, for instance—your hooks are not guaranteed to run.
- Do little work in these hooks. The user will expect that your app will shut down cleanly and quickly when he requests it; it is good not to test his patience.
- Code carefully in this method. This is obviously not a time to play fast and loose with things. The app is shutting down, and the JVM is shutting down; make sure your code here is thread safe.

That's enough about that. I'm going to watch some of the women's college volleyball championships. Oregon State is just demolishing Cal.

System Properties

There are certain things that the JVM always knows about, that you might like to know about, too. These things consist of information about the user's working environment. That's why they are called system properties.

Commonly Used System Properties

Below are listed some of the more commonly referenced system properties. For example, your application might want to perform some specific behavior if the underlying platform is Windows, or you might want to open a JFileChooser dialog that allows the user to save a file, and have it present the user's home directory as its starting location.

user.dir	User's current working directory
user.home	User's home directory
user.name	Account name for currently logged in user
file.separator	File separator (for example, "/")
line.separator	Character representing a line separator on this system
path.separator	Path separator (for example, ":" on Linux, ";" on Windows)
os.arch	Operating system architecture
os.name	Operating system name
os.version	Operating system version
java.class.path	Java classpath
java.class.version	Java class version number
java.home	Directory where Java is installed
java.vendor	String indicating the vendor of this JVM
java.vendor.url	Java vendor URL
java.version	Java version number

Java provides an easy and convenient way to access these properties and to set them, which we'll see how to do now.

PropertiesDemo.java

```
package net.javagarage.demo.properties;

import java.util.Enumeration;
import java.util.Properties;

/** <p>
 * Shows how to get all of the system properties.
 * Note that your methods should really do one thing,
 * not multiple things. Here we violate that for demo
 * purposes; that is, we both get the property AND
 * print it in the same method. Typically, you don't
 * do this.
 *
 * @author eben hewitt
 **/
 public class PropertiesDemo {

 public static void printSystemProperties(){
 //get the system properties
 Properties props = System.getProperties();

 //returns an enumeration
 Enumeration propNames = props.propertyNames();
 while(propNames.hasMoreElements()) {
 //get current property
 //cast to String type, since return type is Object
 String propName = (String)propNames.nextElement();

 //get the value of current property
 String propValue = (String)props.get(propName);
 System.out.println(propName + " : " + propValue);
 }
}

public static void main(String[] args) {

//print only the value of the name specified
System.out.println("Operating System: " +
System.getProperty("os.name"));

//probably not a smart idea (changing system
//properties with brute force), but...
System.setProperty("os.name", "Solaris 8");
System.out.println(System.getProperty("os.name"));

//you can make a new property too, which is cool
System.setProperty("javagarage.coolnesslevel", "supercool");

System.out.println(

System.getProperty("javagarage.coolnesslevel"));


//print all of them

printSystemProperties();

}

}
```

There are a few things going on here. The first is that we can reference system properties by name and get a String value back. The second is that we cannot only get the values of pre-defined properties, but we can create arbitrary new ones, and set them to values of our choosing. We can also overwrite the values of system properties, such as we do with the os.name property. (Not brilliant programming.) Last, we see that we can get an enumeration of all of the properties, and loop over it to print out each value.

There are a couple of things to keep in mind about these properties. The first is access. A desktop application has full access to all of these properties. An applet, which is typically deployed via a browser, does not. There is a subset of the properties that are available, but for security reasons, many are omitted. When your applet tries to reference a property it is not allowed to get at, an exception is thrown indicating the attempted security breach.

The second note is more about good programming practice. You can store values in properties that your application uses, such as preferences for fonts or a welcome message, or other user setting preferences. And although you will see code floating around that uses properties in this way, it is no longer a good idea. A few years ago, this was not only an acceptable use of properties, but one of their intended uses. Which is why you can write to them (duh). However, in recent versions of the Java language, more functional APIs have emerged that address this need in more specific and robust ways. These include the Preferences API, which is used to good effect in the Toolkit Garage Pad example, and the XML APIs, which allow you to store data in a hierarchical format. See, the System Properties stuff is all flat, all on the same plane. It can't be organized. Although the Preferences API requires a little more seductive talk to get it going, it allows you to do stuff like attach properties to individual classes, which is well worth the little bit of extra programming investment.

Also, if you do choose to put a value into the Properties Hashtable, you should be aware that it does not check if the value you add is actually a String, even though that is all they are supposed to store. So use them judiciously.

FRIDGE

Note that you can also use these guys for debugging. In particular, notice the java.class.path and java.library.path properties—you might find that your application requires some functionality provided by a JAR file that is not actually on your classpath, or that it is the incorrect version, or like that. Also, classes in the java.io package will interpret any relative path names as starting with the value of System.getProperty("user.dir"); so it might help you find that file that doesn't seem to be where you thought it would be.

Chapter 32. USING THE JAVA DEVELOPMENT TOOLS

The Java SDK contains a number of tools in addition to the commonly used java interpreter and javac compiler. This topic gives you an overview of all of the tools in the SDK, and details the most commonly used of those.

Using Common SDK Tools

appletviewer

The Java Applet Viewer command allows you to execute applets outside of the context of a Web browser. To view an applet, pass the URL of the HTML document with the embedded applet to the appletviewer program as an argument. If the document does not contain any applets, the program does nothing. The program will run each applet it finds in a separate window.

appletviewer recognizes applets embedded using the `<object>`, `<embed>`, and `<applet>` tags.

The following options are available:

- `-debug` Starts the appletviewer using the Java debug program jdb, allowing you to debug applets.
- `-encoding encodingName` Specifies the input HTML file encoding that should be used when reading the URL.
- `-Jjavaoption` Passes the specified `javaoption` as an argument to the Java interpreter.

Usage: `appletviewer [options] url file`

jar

The JAR tool combines multiple files into a single compressed file called a Java ARchive. It uses the same compression algorithm as Zip files, which means that you can open JAR files using any standard Zip utility. Its primary purpose is to facilitate ease of distribution when managing complete applications. For example, you can bundle your entire application into a single JAR for distribution, and others can use it by simply dropping it into their classpath. Alternatively, a JAR file can be set to be self-executing, so that the freestanding application it contains can be run by double-clicking. Applets have made good use of JAR files, because they facilitate all of the classes in an application being downloaded at one time, which saves considerable HTTP traffic, and speeds execution. In addition, JAR files can be signed so that users can determine the application's author.

If you use UNIX, you will find it easy to begin working with the `jar` command, as its syntax is nearly identical to TAR.

Options

The following options are available for the `jar` command:

- `c` Creates a new archive in a file named `f`, if `f` is also specified.
- `u` Updates an existing JAR file when `f` is specified.

  For example,

  jar uf myApp.jar myClass.class updates the JAR file named myApp.jar by adding to it the class named myClass.class.
- `x` Extracts the files and directories from JAR file `f` if `f` is specified, or the standard input if `f` is omitted. If input files are specified, only those files will be extracted; otherwise, all of the files in the directory will be extracted.
- `t` Lists the table of contents in JAR file `f` if `f` is specified, or the standard input if `f` is omitted. If input files are specified, only those files and directories will be listed; otherwise, all of the files and directories will be listed.
- `i` Generates index information for the specified JAR file and its dependents.

  For example,

```
jar i myApp.jar
```

generates an index called INDEX.LIST in the specified JAR file.

- f Specifies the name of the JAR file on which you want to perform an operation, such as create or update. The option f and the name of the JAR file must both be present if either one of them is present. Omitting f and JAR file accepts a JAR file from standard input (when you specify the x and t commands) or sends the JAR file to standard output (when using the c and u commands).
- v Generates verbose output. That is, it tells you a lot about what it is doing as it does it.
- 0 That's a zero, not the capital letter o. It tells the JAR application not to use compression, but to simply store the files.
- M If you specify the c or u commands, M prevents the creation of a Manifest file.
- m Includes name : value pairs that are in the Manifest file located in META-INF/MANIFEST. MF. A name: value pair is added unless one already exists with the same name, in which case its value is updated. The m and f options must appear in the same order that Manifest and JAR file appear when typing the command.
- -C dir Changes to the directory specified in dir temporarily during execution of the jar command while processing the following filesToAdd argument.

  ```
  jar uf myApp.jar -C classes . -C bin myClass.class
  ```

  The preceding command updates myApp.jar by adding to it all of the files of any type in the classes directory and does not create a directory named classes in the jar. It then temporarily changes to the original directory (.) before changing to the bin directory to add the single class myClass.class to the JAR file.
- -Joption Passes option to the underlying Java runtime, so possible values for option here are the same as the options available to be passed to the java launcher itself.

Typical Examples

The following is how you frequently will use the jar tool:

```
% jar cf myApp.jar *.class
```

This example takes all of the files in the current directory with a .class extension and bundles them into a resulting JAR file called myApp.jar. A Manifest entry is automatically created. A Manifest file stores meta-information regarding an application in the form of name : value pairs.

For executing a JAR file, see the next section.

java

The java tool launches Java applications. It starts a Java Runtime Environment, loads a specified class, and invokes that class' main method. You should use the class' fully qualified name. You may pass arguments to the application; any arguments that are not options passed.

Typical Examples

The following is how you frequently will use the java tool.

```
% java -cp Main.class
```

The java tool will look for the startup class and other necessary classes in three locations: the bootstrap classpath, the installed extensions, and the user classpath.

- -client Selects the Java HotSpot Client Virtual Machine.
- -server Selects the Java HotSpot Server Virtual Machine.
- -classpath The user classpath.
- -cp. Alternative method of specifying the -classpath option. Specifies a list of directories, JAR archives, and Zip archives to search for class files. Class path entries are separated by colons (:) on UNIX, Solaris and Linux, and a semi-colon(;) in Windows. Specifying -classpath or -cp overrides any setting of the CLASSPATH environment variable that might already exist. If neither -classpath nor -cp is used and CLASSPATH is not set, the user class path consists of the current directory (.).
- -Dproperty=value Sets a system property value.
- -d32 -d64 Specifies whether the program is to be run in a 32-bit or 64-bit environment. If neither -d32 nor -d64 is specified, the default is to run in a 32-bit environment, though this will likely change once 64-bit processors become more prevalent.
- -enableassertions[:<package name>"..." | :<class name>] -ea[:<package name>"..." | :<class name>] Enables assertions.
- -disableassertions[:<package name>"..." | :<class name>] -da[:<package name>"..." | :<class name>] Disables assertions (clever…).
- -enablesystemassertions –esa Enables assertions in all system classes.
- -disablesystemassertions –dsa Disables assertions in all system classes.
- -jar Executes an application encapsulated in a JAR file. The first argument to the call is the name of the JAR file containing the application. The manifest of the JAR file must contain a line like this: Main-Class: classnameContainingMainMethod. Using this option, the JAR file becomes the source of all user classes, and the runtime ignores other user class path settings.
- -verbose -verbose:class Displays verbose information regarding each loaded class.
- -verbose:gc Displays information regarding garbage collection events.
- -verbose:jni Displays information about Java Native Interface method execution.
- -version Displays information regarding version information. After the version info is displayed, the program exits.
- -showversion Displays information regarding version information. After the version info is displayed, the program

continues.

- -? –help Displays the usage information in this section and exits.
- -X Displays information regarding non-standard options and then exits. Non-standard options are not presented here.

Here is how to execute a program called "runme.jar" that has been bundled as a JAR:

```
java –jar runme.jar SomeArgument
```

It is assumed that you have created a manifest file for this JAR indicating which class in the application contains the main method.

Discovering Other SDK Tools

There are a number of other tools in the SDK that we won't cover in detail here, because they are often used only in advanced enterprise applications, or else are intended for dealing with fairly specific issues that we don't address here. It is a good idea at least to know about what tools are readily available for your use should your application call for them.

General Tools

These tools are not readily classifiable under one umbrella label, but they come with the kit and you might find them useful.

- javah Generates C programming language headers and stubs. You use it to write native methods.
- extcheck Detects JAR conflicts.
- jdb The Java debugger. Typically, you will want to use an interactive debugger available in an IDE, which is easier.
- javap Java class file disassembler.

Remote Method Invocation Tools

The RMI tools assist in creating applications that allow for interaction of components over the network.

- rmic Generates stubs and skeletons for remote objects. As of Java 5.0, RMI stubs are automatically generated for you.
- rmiregistry The remote object registry service.
- rmid RMI activation service daemon.
- serialver Returns class serialVerUID.

Java IDL and RMI-IIOP Tools

These tools assist in creation of apps that need to interact with CORBA/IIOP and OMG-standard IDL.

- tnameserv Provides access to the naming service.
- idlj Generates Java source code files that map an IDL interface and enable a Java application to use CORBA functionality.
- orbd Provides a way for clients to find and invoke persistent objects on a server in the CORBA environment.
- servertool Provides a convenient interface for registering, unregistering, starting, and shutting down a server.

Internationalization Tools

These help you create applications that are available to different locales. A locale customizes how data is presented and formatted for specific places in the world.

- native2 ascii Converts text to Unicode Latin-1 character set.

Security Tools

The tools in this set assist in setting security policies for your applications.

- keytool Manages keystores and certificates.
- jarsigner Generates and verifies JAR signatures.
- policytool A GUI tool for managing policy files.

Kerberos Tools

The following tools are specific to Kerberos, and if you're using Solaris 8, you have equivalent functionality provided by that OE.

- kinit Tool for obtaining Kerberos v5 tickets.
- klist Command-line tool to list entries in credential cache and key tab.
- ktab Tool to help user manage entries in the key table.

Java Plug-In Tool

This utility is used with the Java plug-in.

- unregbean Unregisters a package JavaBeans component over Active X.

Chapter 33. FAQ

THE LOWDOWN:

- Fast answers to common questions

This section contains a number of Frequently Asked Questions. It is meant to serve as a quick reference guide for when you're working and you need to find out how to do a variety of common tasks quickly.

Setting the CLASSPATH

Sometimes, your programs will reference external libraries that are custom libraries of your own, or a third-party library, or a package in the Java 2 Enterprise Edition. Use the `set` command to set the `classpath` environment variable. The `classpath` tells the Java Virtual Machine where to find necessary class libraries. These libraries can be directories, .jar files (most commonly), .class files, or even .zip files. The path to a .jar or .zip file ends with the file name; the path to a .class file ends with the directory name.

In Windows

First, open a command prompt. Then type

```
set CLASSPATH=.;C:\java\MyClasses;C:\java\My.jar
```

Note that setting the `classpath` in this manner will only sustain the variable for the current command prompt session. Closing the prompt and opening a new one resets the variable.

In Linux/UNIX

In the bash shell you can type the following to set the classpath:

```
export CLASSPATH=/home/someapp/MyClasses.jar:

$CLASSPATH
```

Setting JAVA_HOME in Windows

The environment variable JAVA_HOME sets the location of your Java SDK so it can be accessed from various locations.

On Windows XP

1. Navigate to Start > Control Panel > System.
2. Choose the Advanced tab.
3. Click Environment Variables.
4. In the System Variables pane, click New.
5. For Variable Name, type JAVA_HOME.
6. For Variable Value, type C:\j2sdk1.5.0 for a default installation of the SDK. You may need to adjust your value based on your installation.

On Linux/UNIX

1. Type the following code at the shell (you may need to change the location of the Java home directory for your system):

```
$ export JAVA_HOME=/home/usr/jdk1.5
```

2. To set the JAVA_HOME into the PATH, type the following:

```
$ export PATH=$JAVA_HOME/bin:$PATH
```

Setting the PATH in Windows

The PATH environment variable tells the system where to find the java, javac, jar, and other executables. This enables you to compile Java classes, run your programs, make .jar files, and so forth, from any directory on your computer.

1. Navigate to Start > Control Panel > System.
2. Choose the Advanced tab.
3. Click Environment Variables.
4. In the System Variables pane, double-click the PATH variable. Scroll to the end of the list of variables, and add the following text (for default installation of SDK 1.5.0):

```
C:\j2sdk1.5.0\bin
```

Checking Current Java Version

It is not uncommon to have many different JREs running on the same machine at the same time. To check the current default version:

1. Open a command prompt.
2. Type `java –version`.

You will see output similar to this:

```
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0-b91)

Java HotSpot(TM) Client VM (build 1.5.0-b91, mixed mode)
```

< Day Day Up >

Compiling and Running Programs

1. Open command prompt or shell
2. Change directories to directory where the .java file you want to compile is stored by typing cd.
3. To compile: javac myClass.java
4. To run: java myClass

< Day Day Up >

Primitive Data Types

byte	- 8 bits
short	- 16 bits
int	- 32 bits
long	- 64 bits
float	- 32 bits
double	- 64 bits
boolean	- True or false
char	- 16 bits, Unicode

Declaring and Initializing Variables

Primitives

```
boolean hasRoadRage = true;

char size = 'm';

int age = 11;

long userID = 42L;
```

Arrays

Arrays can be created and populated as shown in the following. Make an array that has two cells for holding int values. The two cells are then populated with the values 11 and 12, respectively.

```
int[] numbers = new int[2];

numbers[0] = 11;

numbers[1] = 12;
```

For the preceding array, calling array.length returns 2.

In the next example, an int array is created with three cells and populated at the same time.

```
int[] numbers = {1,2,3};
```

Arrays can also be populated in loops. Here, we create a new object to be stored in each cell of an array.

```
Employee[] employees = new Employee[5];

for (int x = 0; x < employees.length; x++) {

employees[x] = new Employee();

}...
```

Objects

Create a new object of type Button like this (assuming you have imported the javax.swing.Button class):

```
//reference type of Button, variable name is

//saveButton, and we use the constructor that accepts

//a String label. here we set the label's value to

//'save' right away.


Button saveButton = new Button("save");
```

Class Definition

You can have multiple class files in the same source (.java) file, but only one of them may be declared public.

```
public class MyClass {

    // class definition here

}


class AnotherClass {}
```

Class Modifiers

You can add class modifiers to the declaration that changes the definition of the class.

public

This class modifier indicates that you want it to be available to classes other than just those in the package in which it is declared. That is, you'll let anybody use it. If your class does not include the public modifier, it can only be used by other classes within the same package.

final

To make your class so it cannot ever be subclassed, you include the final modifier in the class declaration. This example is from the standard API.

```
public final class String {...}
```

abstract

Declare an abstract class to indicate that its implementation is not complete (that you didn't write code to do the work in all of the methods). Only abstract classes may have abstract methods. For example, the following is *illegal*:

```
public class MyClass {

public abstract void someMethod();

}
```

In the preceding example, you must either implement the method or declare the class abstract and implement the method in a subclass.

strictfp

This modifier indicates that your class will use the strict IEEE 754 standard for floating point (fp) variables, whether or not the methods or variables themselves are declared strictfp.

Declaring Methods

This example declares a method called setQuantity() that accepts a variable called qty of type int. Note that this is only something you can do in an abstract class or interface—declare it without implementing it.

```
public void setQuantity(int qty);
```

This example declares a method called getQuantity() that returns a variable of type int.

```
public int getQuantity();
```

Methods declared without an access modifier will have default access.

```
void setQuantity(int qty);
```

Calling Methods

Methods are called using dot notation against the object that defines the method. Your code has to account for the return type that the method declares. For example, if getQty returns an int, you call the method to set a variable of type int to the return value. Say that the getQty method returns an int with a value of 5:

```
int theQty = inventory.getQty();

//theQty now has a value of 5
```

You can also call a method directly to get the value without having to set its value into a variable. This is useful in constructors in particular: Say that we have an employee class that defines a constructor that accepts an int value for an ID and a String for a name, and a Manager class with a similar constructor:

```
Employee employee = new Employee(999, "Kevin Bacon");

Manager manager = new Manager(employee.getID(),

employee.getName());
```

You may also have a class that defines static methods—that is, methods that do not require an instance of the class to operate. An example from the API follows:

```
Collections.sort(myArrayList);
```

Notice that we call the sort method directly on the Collections class. The Collections class happens to consist exclusively of static methods that work with collections data.

Overloading Methods

Overloading a method means defining multiple methods in the same class, each with the same name, each of which accepts a different argument list.

```
public void calculateSquareRoot(int i);

public void calculateSquareRoot(double d);
```

The preceding calculateSquareRoot() method is said to be overloaded.

Overriding Methods

Overriding a method means defining a method in a subclass that has the same signature as the method of the same signature in the superclass. For example, the String class overrides the equals() method defined in its superclass Object. The following class definitions show this in action:

```
public class Test {

public void printQuestions(){

System.out.println("Multiple Choice...");

}


public static void main(String[] args) {

Test test = new EssayTest();

test.printQuestions();

}

}


class EssayTest extends Test {

public void printQuestions(){

System.out.println("Essay...");

}

}
```

Executing this file prints "Essay" to the standard out.

Reading User Input from the Standard In

Sometimes, you want to accept user data at runtime from the console. The following code example demonstrates how to read user input from the standard in (System.in).

```
//do imports and such

public static void main(String[] args) {

    System.out.println("Please say something ");

    try {

        BufferedReader is = new BufferedReader(

                new InputStreamReader(System.in));
        String input;

        while((input = is.readLine())!= null){
            //consider doing something even more
            //exciting here with the input
            System.out.println("You wrote: " + input);
          is.close();
          System.exit(0);
        }
      } catch(IOException ioe){
        System.err.println("Exception: " +
                                  ioe.getMessage());
      }
}
```

Package Declaration

A package is a logical and physical grouping of Java classes and interfaces, whose purpose is to minimize name conflicts. When using the package statement, it must be the first noncomment line in your source file. A package is declared using the package keyword as shown here:

```
package net.javagarage.utils;

// class definition here...
```

Import Declaration

The import command allows you to reference a class in the imported package only by its class name instead of having to type the fully qualified name every time you reference it in your class. For instance, java.lang.String is the fully qualified name for the String class. The String class is contained in the java.lang package. Because java.lang is always imported automatically, we are free to refer to a String without prefixing it with its package name.

To import all of the classes in a package, use a wildcard.

```
import java.sql.*;

// now you can refer to any class in the

// java.sql package by class name only in your code:


class myClass {

try {

    //look, ma! no package!

    Statement stmt = con.createStatement();

    //....

}...
```

Note that the import statement should be the second noncomment statement in your class file, following a package declaration.

You can import the static methods of a class using the new static import feature of Java 5.0:

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

Inheritance

The following example shows how to create a subclass using the extends keyword. In this example, Vehicle is the superclass, and Car is the subclass that inherits all of Vehicle's behavior. Car then will define behavior specific only to Cars.

```
class Vehicle {}

class Car extends Vehicle {}
```

Defining and Implementing an Interface

An interface is a set of requirements to which classes must conform if they want to use the service provided by the interface. To implement an interface, first declare that your class will implement the given interface; second, define the methods in the interface.

java.util.Collection has an interface that defines, among other methods, an add() method. This ensures that any implementation of java.util.Collection will support its methods.

An interface is defined as follows:

```
public interface Collection {

public abstract boolean add(A o)

...

}
```

An interface is implemented as follows:

```
public class myClass implements Collection {

//code here. can write, for instance, add() since

//add() is a method of java.util.Collection, and

//this class implements that interface. Eventually

//all methods in Collection must be implemented...

...

}
```

Exceptions

Throwing an exception means that you don't want to deal with it: You will let the caller handle it (or pass the buck too). Declare that your method throws an exception using the throws keyword, like this:

```
public void someMethod() throws Exception {
    if(somethingBad) {
    throw new Exception("Something bad happened: ");
    }
}
```

If you are using library code that throws an exception and you don't want to deal with it in a try/catch block, you can just say that your method throws this exception, and not think about it.

Alternatively, you can catch an exception to handle the problem inside the method, as shown here:

```
    public void someMethod() {
        // code for an operation that could cause an
        // exception...
        try {
        // ...some code
    } catch(Exception e) {
        // exception handling code here
    }
    finally {
        // this code executes whether an exception
        // was generated or not. The finally clause
        //is optional
    }
}
```

Working with JAR Files

A JAR file (Java ARchive) uses the same algorithm as a Zip utility does to compress numerous documents into one archive document. The benefit of a .jar file is that the developer can combine HTML, applets, class files, images, sounds, and everything else that makes up an application, and put them all into a JAR to facilitate easy distribution of the application. The jar tool is a Java application tool, regularly available with the SDK.

Note that you can add v to the command-line call options for verbose output, to indicate what is happening while your JAR is being created.

1. Creating a New JARNavigate to the directory in which you want to create the JAR.
2. Type the following at the command line:

```
jar cf myJar *.class
```

Adding All of the Files in a Directory to the JAR

1. Navigate to the directory in which you have the JAR.
2. Type the following at the command line:

```
jar cvf myApp.jar
```

Extracting the Contents of a JAR

FRIDGE

If you type the jar command and, instead of working, Windows tells you something like, "'jar' is not recognized as an internal or external command, operable program, or batch file," it means that the system cannot find the jar program to run it. So we have to put the jar program on the path (the list of places that Windows goes to find things to execute). We do this by setting the PATH environment variable to include the location of your bin directory (<JAVA_HOME>\bin), because that's where the jar program lives.

1. Navigate to the directory in which you have the JAR.
2. Type the following at the command line:

```
jar -xvf myApp.jar
```

You should see the program extracting the files into the pwd (Present Working Directory).

What Is the Java Virtual Machine?

The JVM interprets and executes bytecodes, which are the executable representation of a Java program. The JVM emulates a hardware platform (hence, the name Virtual Machine), including registers, program counter, and so forth.

What's In the SDK?

The SDK consists of the following items:

- A JVM, or Java Virtual Machine
- A Java compiler, which takes a plain text source code file and turns it into executable bytecodes
- appletviewer, a program for viewing applets
- A debugger
- Example java programs
- A tool for creating and managing security keys
- A remote method stub and skeleton generator
- A registry server for handling remote method invocations (method calls coming from objects on a different JVM)
- C and C++ headers for extending the JVM with interfaces to native code
- Native libraries for embedding a JVM into other native applications
- Native libraries for the portions of the JVM that are platform dependent, such as those that help generate graphical user interface components.

Chapter 34. PACKAGING & DEPLOYING JAVA APPLICATIONS

DO OR DIE:

- How to package your application in a compressed JAR file
- How to create an executable JAR
- How to make your application presentable with a shortcut and icon.

It is likely that you will take your Java skills and head for the Web. That is not a bad idea, considering that roughly 80% of all new Java development happens on the Web. Although 68% of statistics are made up on the spot, this one indicates that not much is happening on the desktop for Java. Sun is doing considerable work to change this perception, and indeed there are a number of fantastic things happening on the desktop with Java. To stay on top of cool new developments with Java GUI applications, check out the Swing Connection (and Swing Sightings in particular) online at http://java.sun.com/products/jfc. You also might enjoy project Looking Glass, which brings a real 3D feel to OS windowing.

But programs come in many forms. There are enterprise applications, systems development, desktop applications, embedded systems, and micro applications. And many of these environments either require or encourage your packaging of an application in some form. Packaging is the process of preparing your application for easy deployment or distribution. Deployment is the process of putting your application in a context where it can be run by a user.

Herding Cats

Imagine trying to distribute an application, uncompressed, to end users: "Now downloading file 2 of 768…". This is what my friends from Texas call herding cats. It is obviously easier to deal with one file than many files, so typically packaging involves taking the many files in your application and putting them into one or two manageable files that can then be FTP'd or copied across the network or burned onto a DVD for sale in stores.

Making One File: Packaging into a JAR

If you have used Microsoft's Visual Studio, you know that it is very easy to make a program you write into an executable file. In Java, you don't make .exe files as you do in Visual Studio. Instead, you make a JAR file, which can be executed on a system that has a Java Runtime installed. It can be a bit of a drag, more of a drag in my view than it should be for a language this mature. But there is always a trade-off between convenience and control. In the long run, it is nice to have the control; so, okay, I will stop whining about it and thank Sun for their prescience. Now let's see how to do it using the JAR tool.

Packaging Using the JAR Tool

A JAR file is a file compressed with the same algorithm used to compress Zip files. JAR stands for Java ARchive. You create a JAR file using the jar command. If you are on Windows, go to this directory: <JAVA_HOME>/bin/. This is where the tools in the SDK live, and therefore where you find the jar program.

GOT JAVA EXECUTABLES?

If you don't care about being a Java "purist" and know that all of your users are on a Windows environment, you can compile your program to native code, bundling your application as an .exe file, executable only in a Windows environment. Although the JDK does not ship with any native compilers, some IDEs allow you to do this (JBuilder for example). There are also a few free programs out there that, as you might imagine, work to greater or lesser degrees. However, as JoJo the Internet Guy says, this is not a five-star idea. You obviously lose portability, perhaps even some functionality, and there are alternatives. For example, you could write a batch file or shell script to make things a little less punishing for your users. We'll see how to do this in a moment. But if you must have things your way, here are some starting places:

JBuilder

http://www.xlsoft.com/en/products/jet/

http://www.excelsior-usa.com/jet.html

If you type jar at the console, the program will display usage options. Here, we will create a JAR file in the C:\apps directory called BlogApp.jar. It will contain all of the files with a .class extension in the target directory.

In running the command, the c is for "create," the v for "verbose" (which means, "tell us what you're doing in detail as you do it"), and f means that you will specify the resulting file name. Run the following command to create the JAR:

```
C:\>jar cvf BlogApp.jar blogger

added manifest

adding: blogger/(in = 0) (out= 0)(stored 0%)

adding: blogger/BloggerFrame$1.class(in = 1017) (out=564)

(deflated 44%)

adding: blogger/BloggerFrame$2.class(in = 864) (out=479)

(deflated 44%)

adding: blogger/BloggerFrame$3.class(in = 828) (out=458)

(deflated 44%)

adding: blogger/BloggerFrame$4.class(in = 741) (out=401)

(deflated 45%)

adding: blogger/BloggerFrame.class(in = 3801) (out=1945)

(deflated 48%)

adding: blogger/BloggerKeys.class(in = 670) (out=422)

(deflated 37%)

adding: blogger/blogs/(in = 0) (out= 0)(stored 0%)

adding: blogger/blogs/blog.html(in = 52) (out= 42)

(deflated 19%)

adding: blogger/StartApp.class(in = 884) (out= 520)

(deflated 41%)

C:\>
```

Executing the command, as the output shows, creates a new file in the current directory called blogger.jar. Class files are compressed as added, and a manifest is created. A manifest is a text file that specifies meta data about your application. It must be in a directory called META-INF and the file itself must be called MANIFEST.MF (both the file and the directory are all uppercase). The chief purpose of the manifest file is to point the runtime to the location of the class file containing the main method so it can start the program.

Adding the Manifest

The JAR tool can create a manifest for you if none exists. But then it will not contain some necessary information, such as the class containing the main method. You will need to then extract the JAR, add the information to the manifest, and recompress the JAR. Might as well just create the manifest right off the bat.

Navigate into the JAR and open the META-INF directory and then the MANIFEST.MF file. This is a plain text file in which you can specify things that you want the application to know at runtime. Although the JAR tool created this folder and this file, it did not pick out for us the class with the main method, which it will need to know in order to start the application automatically. So, we will just write our own manifest in a plain ASCII text file. Do this:

1. Make sure that your classes are compiled in the appropriate structure (that is, in their packages).

2. Write a plain text file called manifest.mf. In this file, type the following: Main-Class: package.otherpackage.MainClass. Do not include the .class extension. Your text here will be used as an entry in the real MANIFEST.MF file produced by the JAR tool.

3. Navigate to the working directory one level *above* the first-level package in your app.

4. Run the jar command like so: jar -cvmf MANIFEST.MF MyApp.jardirectory/subdirectory, where subdirectory contains your class files.

This will create a JAR file called MyApp.jar in the current working directory:

```
jar -cmf MANIFEST.MF MyApp.jar

    net/javagarage/test/StartApp
```

Note that there are many different subtle ways to manipulate JAR file creation. Check out the usage information if you need more control.

PREV < Day Day Up > NEXT

Creating JAR Files with Eclipse

This process is somewhat simpler using the Eclipse IDE. Any reasonable IDE should give you the ability to create a JAR for your application; we use Eclipse here because it is free and pretty reliable and we've been using it a lot in this book.

1. With an open project, click File > Export….
2. Select the JAR file and click Next.
3. Click on the application in the Select the Resources to Export window, because you may not want to include everything. Choose the classes directory (and the source code if you want to distribute it).
4. In the Select the Export Destination text field, choose where you want your JAR to end up, and give it a name.
5. Click Next, and Next again.
6. Click the Use Existing Manifest from Workspace radio button and browse to the location of your manifest. Click Finish and you're done.

Executing the JAR file

FRIDGE

If you have problems running this command, as many people do, try the following hack, which seems to do the trick sometimes: Put the line with the Main-Class entry immediately following the Manifest-Version: 1.0 line (if you create the complete manifest manually). Also, make sure there is a carriage return after your Main-Class entry, or it won't work properly. Sheesh.

To execute your JAR file, type the following command at the console:

```
java -jar MyApp.jar
```

If you need to pass arguments into your application, you can pass them in just as you normally do with the java command:

```
java –jar MyApp.jar myFirstArg mySecondArg
```

This passes your executable JAR file to the java program for execution.

Alternatively, in Windows, you can double-click the JAR file.

If you get an error message saying, "Could not find the main class," this is one time you shouldn't believe what you read. There obviously is a problem; it just might not be the problem you're told it is. It could mean that there is a dependency that your application has that is unfulfilled. For example, if your application sends an e-mail using the javax.mail package, you need to put the J2EE library in your application folder and add an entry for it in the manifest. Specify that you need the j2ee.jar file on your classpath this way:

```
Class-Path: j2ee.jar
```

Then, make that file accessible to your application.

Note too that JARs can store other JARs. This is commonly done to deploy a complete application. Say you have an executable JAR, and then some other files that you want to distribute with that executable JAR (like a README.txt, a properties file, etc.). That you can do, as long as you are okay with your users un-JARing that distribution JAR in order to get to your executable JAR.

But say you have a JAR file that is a library of classes that do some particular thing, such as make PDF files. You've downloaded this JAR, and your application code relies on it. You need to *un-jar* (extract) that JAR so that the classes in it are all flat with your application classes. Then you can re-JAR the whole thing all together. Otherwise, your executable code won't know how to find the classes it needs.

Making your Java Program Execute Automagically on Startup

So there I was on the corner of Central and Jefferson, just holding up a lamppost, if you got to know, when up comes this little sassafras grousin' for a light and just like that he says a me, "I got a Java program, see. And it needs to start when my OS starts, automatically." Here's how to do it in both Windows and Linux.

In Windows

Write a batch file that executes your program using the java command, and create a task that executes the batch file. Easy!

1. Navigate Windows Explorer to C:\WINDOWS\Tasks. Click Add Scheduled Task.
2. Click Next and browse to the batch file and click on it to select it.
3. Click the button that indicates when you want to run the program, type your password, and click Finish.

   Here is an example so you can get the feel for it.

   First, I create a file called MyStartupMessage.bat. It contains the following command:

   ```
   rem eben was here

   java net.javagarage.packaging.MyStartupMessage
   ```

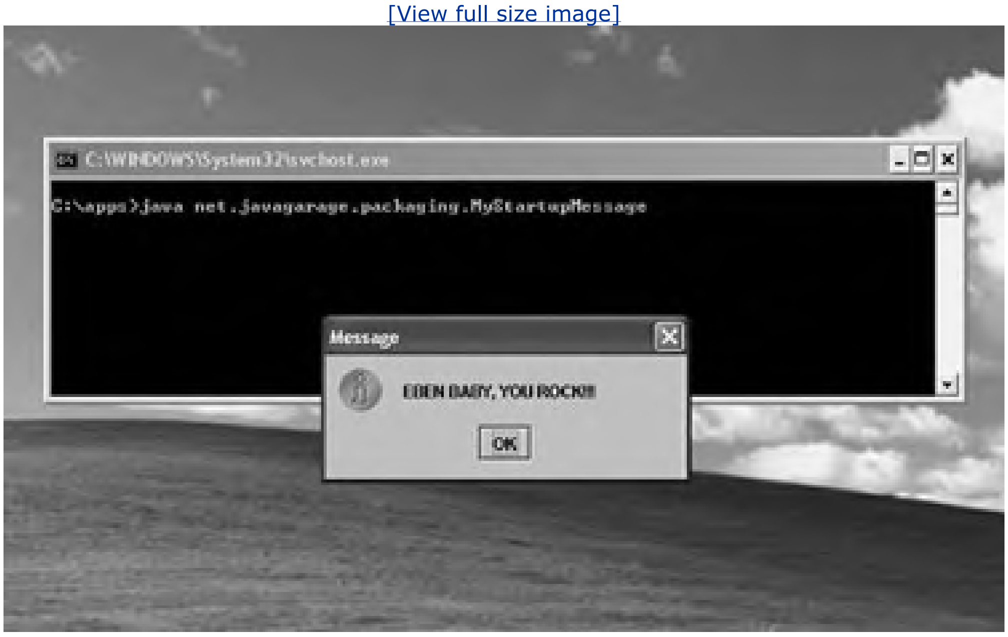
This batch file is in a folder called C:\apps. Also in that folder is the folder net, which contains the folder javagarage, which contains the folder packaging, which contains the Java program MyStartupMessage.class. That class contains the following important method:

```
public void helpSelfEsteem(){
  JOptionPane.showMessageDialog(null,
    System.getProperty("user.name").toUpperCase() +
    " BABY, YOU ROCK!!!");}

   //run the program
public static void main(String[] args) {
   helpSelfEsteem();
}
```

Just create the Windows task as indicated in the preceding, and you're good to go. Figure 34.1 shows the output I get.

Figure 34.1. The application executing automatically when the system starts up.

[View full size image]



```
C:\WINDOWS\System32\svchost.exe
C:\apps>java net.javagarage.packaging.MyStartupMessage
```

```
Message
EBEN BABY, YOU ROCK!!!
OK
```

In Linux RedHat 7.3 and Later

The steps are similar if your target OS is Linux. If you've got a Linux box with a graphical user interface like KDE, you can just navigate to System Tools > Task Scheduler and add the task.

But say we have installed Linux as a server and don't have any graphical user interface libraries installed (in which case we can't run the preceding message dialog program). The corresponding function is called cron. Here's how to do it. Let's make a lame-brained Java program called MyStartupMessage that writes out a file telling you what time the server started.

It looks like this:

```
//do whatever work you want to perform after startup
private static void doWork(){
    //on my windows system this is
    //C:\Documents and Settings\eben

    String fileName = System.getProperty("user.home");

    //on windows this is \, on unix it is /
    String slash =
                System.getProperty("file.separator");
    fileName += (slash + "startupMessage.txt");

    File file = new File(fileName);
    try {
        BufferedWriter out = new BufferedWriter(
                        new FileWriter(file));
        out.write("The server restarted at " +
                            new Date());
        out.close();
    } catch (IOException e) {
    System.err.println(e.getMessage());
}
```

Now check your user home folder; the file is in there with the timestamp.

PREV < Day Day Up > NEXT

Creating an Icon for Your Java Application on Windows

Say that you have created your JAR, and created a batch file that performs certain useful tasks such as setting environment variables, and now you want to make it pretty like a real, professional type application. To do this, create an icon, create a shortcut to the .bat file, and then change the icon used by the shortcut.

1. Right-click on your .bat file or executable JAR file. Select Create Shortcut.
2. Select Properties.
3. Click Change Icon….
4. Choose one of the many icons included with Windows just for a test. Browse to the location of your .ico file and click Apply for the real deal.

Don't be discouraged with all of this deployment business. There are ways of automating all of the necessary file copying, moving, deleting, generation, and even the Java compiling, JAR-ing, and more. The most popular way by far of automating all of this business is Ant. You can read about Ant, and download it for free at http://ant.apache.org. It integrates well with the most popular IDEs, and if you are using Eclipse (http://www.eclipse.org), then you already have Ant.

FRIDGE

What's that? You don't have an .ico file? Such a file is an icon, a special kind of bitmap. You need an editor that will help you create them; you'll have a hard time in Photoshop alone or other typical graphic editors. I like the free Icon Studio from CoffeeCup software (http://www.coffeecup.com/freestuff/). It's full-featured and you can't beat the price. Just download and install it, then create your own icon and choose it for use on your shortcut. Windows will replace it immediately, and you're good to go.

Chapter 35. TOOLKIT

DO OR DIE:

Demonstrate a number of complete, varied, useful, quality apps, including

- Name: A simple data class
- A Credit Card Validator
- Garage Pad: A Simple Text Editor
- An RSS Feed Reader
- Garage Doodler: A Drawing Pad

This topic offers a gallery of straight-shooting Java applications that illustrate how to use the concepts we've covered in complete, working, real-world type applications. After all, that is the purpose of all this—to get us writing complete applications. The idea is that you will really understand how the concepts fit together if you *see* them fitting together. It can be very frustrating to read a book filled with tiny code snippets, none of which compile on their own, so I didn't want to do that.

Here, therefore, is a collection of different small applications that you can study, incorporate into your own projects, or otherwise use freely as you see fit. Although I don't warrant the usability of this code for any particular purpose, it is presented here in order to support the topics of this book, to help you learn to program Java better. Translation: Use it, sell it, do whatever you want with it; I wouldn't launch my space shuttle on it though, and I wouldn't wonder (aloud, to me, in e-mail) why certain "features" are "missing," because its purpose is academic. Though it would generally be fine in production. Some concepts are presented in a simplified, or repeated manner, in order to help the ideas sink in.

Comments interspersed frequently throughout the files provide the discussion. This will hopefully encourage you to use code comments to communicate with the inheritors of your applications, instead of relying on model documents or using case diagrams.

A Name Data Class

Often in books, we see programs that learn us how to do things like calculate and draw and other action-verb-oriented things. In business, however, we often need to also represent things. Object-oriented programming has as one of its tenets that an object contains data and behavior. Sometimes, however, this behavior is either fairly limited, or is implemented fully in increasing usability. Take, for example, a simple thing like a person's name.

Many applications need to represent names, but they often are implemented poorly. On the one hand, a complete name gets represented as a single String. This makes it almost impossible to deal with in searches, comparisons. Other times, names are only slightly more conscientiously dealt with by offering separate Strings for each part of the name. This does make searching and comparison more accurate, but has the deleterious effect of decoupling related data, which is a maintenance nightmare. If your manager tells you that now your name collection form needs to include a middle name, for instance, you must update many different parts of the application. In a Stuts-based Web application, for instance, this would be a real drag.

If, however, you are dealing with a Name *class*, the organization of your data is shielded (encapsulated) behind it, so your maintenance is far easier. But you know that already. What are some other benefits to this approach? Well, for one thing, you gain total control over presentation. You can display this data exactly how you want, and offer ways for clients to include certain parts or omit others (such as the ever-questionable middle name). It also makes sure (well, encourages you, if you are being lazy) that you override equals and hashcode, and think hard about your application design and implementation.

The downside is that a Name is generally given to another entity, such as a Person. So you probably have a Person class, with a Name member, and maybe an Address member (which would be a complex object itself!), and so on, and all of a sudden you have proliferated classes and slowed down your application as it goes to the trouble of creating seven or eight objects when all you wanted was two lousy little Strings. Now, if you aren't going to use any of that additional information in a very sophisticated way, that might be overkill.

Obviously, you need to determine what your application is focused on, how many concurrent users it is likely to have, what the data will be used for, and what kinds of views you will need of the data before you go off the deep end like this. Sometimes a couple of Strings is just what the doctor ordered.

Hey! This class uses generics, in order to give you a taste for it. They are not explained in this book, and neither is the Collections API. Don't worry. The following ArrayList business simply says this: "Make a list (which is like an array that is resizable), and let it accept only objects of type Name." Then we just call the static sort method to sort them according to our implementation of the Comparable interface.

Demonstrates

Overloaded methods, overriding methods, overloaded constructors, using generic types, using Lists, implementing interfaces, sorting Collections, using switch/case constructs, constants, JavaDoc, formatting, and usability considerations.

Name.java

```
import java.io.Serializable;

import java.util.ArrayList;

import java.util.Collections;

import java.util.List;


/** <p>

 * Represents a person's name in numbing detail.

 * Meant for use by another class (i.e, "Person").

 * Implements the Comparable interface so that Name
 * objects can be easily sorted alphabetically.
 * Implements the Serializable interface, which is
 * merely a marker interface, in order to indicate
 * that it is safe to serialize objects of this type.
 * This class will only compile under SDK 1.5.
 * </p>
 * @author eben hewitt
 **/
public class Name implements Serializable, Comparable<Name> {

   private String title=""; // ie, "Miss" or "Dr."
   private String suffix=""; // ie, "junior" or "III"
   private String given=""; //first
   private String middle=""; //multiple middle
   private String family=""; //last

   public static final int FAMILY_GIVEN_MIDDLEINIT = 0;
   public static final int FAMILY_GIVEN = 1;
   public static final int GIVEN_FAMILY = 2;
   public static final int GIVEN_MIDDLE_FAMILY = 3;
   public static final int GIVEN_MIDDLEINIT_FAMILY = 4;
   public static final int FULL_NAME_US = 5;
   public static final int PREFIX_FULL_SUFFIX_US = 6;

   //overloaded constructors
   /**
    * Constructs Name with first and last names.
    */
   public Name(String given, String family) {
      this.given = given;
      this.family = family;
   }
   /**
    * Constructs Name with first, middle,
    * and last names.
    */
   public Name(String given, String middle,
            String family) {
      //re-use the constructor defined for just
```

```
         given + family
    this(given, family);
    this.middle = middle;
}
/**
 * Constructs complete Name object. You can have
 * multiple overloaded constructors. First calls
 * the earlier constructor—that call must be
 * first line of code here.
 */
public Name(String title, String given,
              String middle,
              String family, String suffix) {
    this(given, middle, family);
    this.title = title;
    this.suffix = suffix;
}

// helper methods
/**
 * Gets initial character of first name as an
 * initial (with an added ".").
 * For example "Eben" returns "E.".
 * @return The initial character of the first
 * name.
 */
public String getGivenInitial() {
    return given.charAt(0) + ".";
}
/**
 * Gets initial character of middle name as an
 * initial (with an added ".").
 * For example "Mann" returns "M.".
 * @return The initial character of the middle
 * name.
 */
public String getMiddleInitial() {
    return middle.charAt(0) + ".";
```

```
}
/**
 * Gets initial character of first name as an
 * initial (with an added ".").
 * For example "Hewitt" returns "H.".
 * @return The initial character of the last
 * (family) name.
 */
public String getFamilyInitial() {
    return family.charAt(0) + ".";
}
/**
 * Gets initial character of each of first,
 * middle, and last names
 * (with an added ".").
 * For example "Eben Mann Hewitt"
 * returns "E. M. H.".
 * @return The initial characters of the first,
 * middle, and last names.
 */
public String getInitials() {
    return getGivenInitial() + " " +
                getMiddleInitial() + " " +
                getFamilyInitial();
}

/**
 * Returns the various parts of a name together
 * in common representations.
 * @param repType The manner in which you want
 * the Name represented, useful
 * for different purposes:
 * <ul>
 * <li>(default) FULL_NAME_US: Eben Mann Hewitt
 * <li>FAMILY_GIVEN_MIDDLEINIT: Hewitt, Eben Mann
 * <li>FAMILY_GIVEN: Hewitt, Eben
 * <li>GIVEN_FAMILY: Eben Hewitt
 * <li>GIVEN_MIDDLEINIT_FAMILY: Eben M. Hewitt
 * <li>PREFIX_FULL_SUFFIX_US: Mr. Eben Mann
 * Hewitt, Esq.
 * @return The name, as represented by
 * the parameter.
 */
public String getName(int repType) {
    switch (repType) {
    case FAMILY_GIVEN_MIDDLEINIT:
         return getFamily() + ", " + getGiven()
                     + " " +
                     getMiddleInitial();
    case FAMILY_GIVEN:
          return getFamily() + ", " + getGiven();
    case GIVEN_FAMILY:
          return getGiven() + " " + getFamily();
    case GIVEN_MIDDLEINIT_FAMILY:
          return getGiven() + " " +
                getMiddleInitial() +
                " " + getFamily();
    case PREFIX_FULL_SUFFIX_US:
    //ternary operators used to remove
    //unnecessary whitespace
    //in the event that title or suffix is blank
          return (getTitle() != "" ? getTitle()
              + " " : "") + getGiven() +
              " " + getMiddle() + " " +
              getFamily() +
              (getSuffix() != "" ? (", " +
              getSuffix()) : "");

    default :
    case FULL_NAME_US:
          return getGiven() + " " + getMiddle() +
                 " " + getFamily();
    }
}
/**
 * Override of Object's toString method.
 */
public String toString() {
    return getName(GIVEN_FAMILY);
}


/**
 * Override of Object's equals method.
 * Determines object equality by testing family,
 * middle, and given names as well as class.
 *
 */
public boolean equals(Object other){
    //are they the same object?
    if (this == other)
         return true;
    //are they of same type?
    if (other != null && getClass() ==
               other.getClass()){
         //since the classes are the same type,
         //downcast and compare attributes
            Name n = (Name)other;
            if (n.getFamily().equals
                   (this.getFamily()) &&
               n.getGiven().equals
                   (this.getGiven()) &&
               n.getMiddle().equals
                   (this.getMiddle()))
         return true;
    }
         return false;
}

//overloaded getName() method:
public String getName(){
    return getName(PREFIX_FULL_SUFFIX_US);
}



/**
 * Required by the Comparable interface. Returns
 * 0 if the objects have the same
 * first-middle-last combination. Its purpose is
 * to help us perform during a call to
 * {@link java.util.Collections#sort}.
 * Alphabetical order is preferable here.
 * @return int indicating if the objects can be
 * considered equivalent. A value of 0 indicates
 * equivalency. A negative value indicates this
 * Name comes earlier; a positive value indicates
 * this Name comes later.
 * @throws ClassCastException
 * @see java.lang.Comparable#compareTo
 * (java.lang.Object)
 */
public int compareTo(Name anotherName) throws
  ClassCastException {
  if (! (anotherName instanceof Name))
        throw new ClassCastException("Object
                   must be instance " +
                   "of Name to compare. ");

  String thisFirstAndLast = this.toString();
  String otherFirstAndLast =
           anotherName.toString();

  return
  thisFirstAndLast.compareTo(otherFirstAndLast);

}

//get and set methods
public void setGiven(String given) {this.given =
       given; }
public String getGiven() { return given; }
public void setMiddle(String middle)
      { this.middle = middle; }
public String getMiddle() { return middle; }
public void setFamily(String family)
      { this.family = family; }
public String getFamily() { return family; }
```

```
    public void setTitle(String title)
        { this.title = title; }
    public String getTitle() { return title; }
    public void setSuffix(String suffix)
        { this.suffix = suffix; }
    public String getSuffix() { return suffix; }


    //included just for testing
    public static void main(String[] ar){
      Name homer = new Name("Homer", "J.",
            "Simpson");
      Name marge = new Name("Marge", "Simpson");
      marge.setMiddle("B.");
      System.out.println("Are they same? " +
            homer.equals(marge));

      System.out.println(marge.getName
  (Name.FAMILY_GIVEN_MIDDLEINIT));

      System.out.println("Implicit toString: " +
           homer);
      System.out.println("Is Homer equal to Homer?
           " +
           homer.equals(new Name("Homer", "J.",
               "Simpson")));

      //illustrate the sorting ability
      List<Name> names = new ArrayList<Name>();
      names.add(marge);
      names.add(homer);
      //sort and print

      Collections.sort(names);

      for (Name aName : names) {
          System.out.println(aName);
      }
    }
  }
```

Result

```
Are they same? false

Simpson, Marge B.

Implicit toString: Homer Simpson

Is Homer equal to Homer? true

Homer Simpson

Marge Simpson
```

If we had not sorted the previous list of names, then they would be returned in the order added ("marge" first, and "homer" second).

The previous Name class is useful for modeling data-type classes. Of course, classes in Java are meant to hold both data and the means of performing operations, but some classes offer a service and some classes represent simple nouns. Such as "Name." Cheers.

(source: AP, Rutgers University Study, Douglas L. Kruse)

AMERICAN COMPANIES SAVED MORE THAN $155,000,000 IN 1996 BY ILLEGALLY HIRING UNDERAGE CHILDREN. THIS PRACITCE CONTINUES TODAY.

PREV < Day Day Up > NEXT

Credit Card Validator

This little program validates credit card numbers according to their structure. That is, it does *not* tell you anything regarding the account associated with the credit card number passed to it. All it does is tell you that the argument passed could be a real credit card number. It does this using the Luhn formula. This app is easily incorporated into any e-commerce application you might have in place, and works well as a front-line validation before incurring the overhead required to check the account and process the transaction.

Demonstrates

Converting Strings, ternary operator, validating input, designing the methods of a class, JavaDoc, using the Character class, for loops, arrays.

```
package net.javagarage.misc;

/** <p>
 * Shows how to validate a credit card number.
 * Credit card numbers should check out against
 * the LUHN formula (MOD 10 algorithm).
 * <p>
 * Will check out fine against the numbers commonly
 * used for testing, such as 4111 1111 1111 1111
 * <p>
 * On a credit card, the first number identifies
 * the type of card (MC, Visa, Amex, etc), and the
 * middle digits belong to the bank, and the last
 * digits to the customer.
 * <p>
 * Gives you code that you might one day really need,
 * and demos how to use for loops, Character,
 * separate work, and use the ternary operator.
 *
 * @author eben hewitt
 * @see Character,
 **/
public class CreditCardValidator {

/**
 *Use private constructor to disallow
 *creation of objects. it isn't necessary
 *since we're just doing an operation, not
 *storing data
 */
private CreditCardValidator() { }

/**
 * Determines if the number checks out against the
 * algorithm—obviously it does not perform any
 * operation with respect to the user's account.
 * <p>
 * @param String the number to be validated
 * @return boolean true if the card is valid,
 * false otherwise.
 **/
public static boolean validate(String input) {
    //first off, make sure we're even
    //in the ballpark of a real number
    //(not NULL and between 13 and 16 digits)
    if (input == null ||
        (input.length() < 13) ||
        (input.length() > 16)){
    reject();
}
//remove spaces, dashes, etc
String number = cleanup(input);

//this will be our total
int total = 0;
//multiply all digits but the first one by 2
int position = 1;
//Starting with the second-to-last-digit...
for(int i = number.length() - 1; i >= 0; i—) {
// make each digit in base 10
int digit = Character.digit(number.charAt(i), 10);
//multiply every other digit by 2
//starting with the second one
digit *= (position == 1) ? position++ : position—;
//if the digit is greater than 10,
//try mod 10 + 1

total += (digit >= 10) ? (digit % 10) + 1 : digit;

}


//a valid number MOD 10 will be 0

//the result of this expression is the boolean return value

return (total % 10) == 0;

}


/**

 * Clean the passed string so that it contains

 * only numbers, not spaces or dashes or anything

 * extraneous.

 * @param String the number as entered by user

 * @return String the cleaned-up number

 **/

public static String cleanup(String input) {

char[] result = new char[input.length()];

//index of the char array

int idx = 0;

for (int i = 0; i < input.length(); i++) {

char thisChar = input.charAt(i);

//use static Character wrapper class

//method to see if it's a digit

if (Character.isDigit(thisChar)) {

//it is, so keep it

result[idx++] = thisChar;

}

}

/*

 * String has a constructor that builds a

 * String from a subset of a char[], using

 * the char[], the starting position (here, 0),

 * and the ending position (here, idx).
 */
return new String(result,0,idx);
}

/**
 * Used when the user passes bad information to
 * stop the show and help them out.
 */
private static void reject(){
System.out.println("usage->java CreditCardValidator number");
System.exit(-1);
}

//start the app
public static void main(String[] args) {
if (args.length == 1) {
System.out.println("Is valid number? " +
validate(args[0]));
} else {
//user didn't enter a card number
reject();
}
}
}//eof
```

Result

The result of running the validator with different arguments is shown here.

Passing 12345 yields this result:

```
usage->java CreditCardValidator number
```

That's because the program rejects the argument if it is obviously not anything like a credit card number (between 13 and 16 characters).

Passing 456789098765432 yields this result:

```
Is valid number? false
```

Passing 4111111111111111 yields this result:

```
Is valid number? true
```

In the next few sections, we'll look at putting together a text editor, an RSS newsreader, and a drawing pad. It will be very thrilling and educational.

Application: SimpleTextEditor: Garage Pad

This program, the Garage Editor, is a simple text editor that allows users to open files, save their file, copy and paste, and display help information.

This is a terrific little application because it demonstrates a number of concepts important in the creation of real Java programs.

Demonstrates

Creating a basic GUI window, creating scrollbars on demand, creating a toolbar containing menus that contain menu items, how to use the JFileChooser to save and open files, cleanly closing an open document, exiting from an application, displaying informational messages to the user (like JavaScript alerts or Visual Basic MsgBox), using keyboard mnemonics (using key combinations such as Ctrl+S for Save), and using the preferences API to save user-specific preference data regarding how the application is run. In general, this is a pretty tight application, and does all its work in one file.

LUHN ALGORITHM FOR VALIDATING CREDIT CARDS

The LUHN (or MOD 10) algorithm used for validating credit card numbers is widely known and accepted. Here is how it works:

1. Start at the second digit from the right. Double each alternating digit. (Leave the last digit on the right alone since it is the check).
2. Add each individual digit comprising the products from step 1 to each of the unaffected digits.
3. If the product of step 2 ends in 0 (that is, can be divided by 10 with no remainder), the number is valid.

   For example, using 4111111111111111 (the common credit card test number):

   number: 4 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

   multiply: 2 2 2 2 2 2 2 2

   product: 8 2 2 2 2 2 2 2

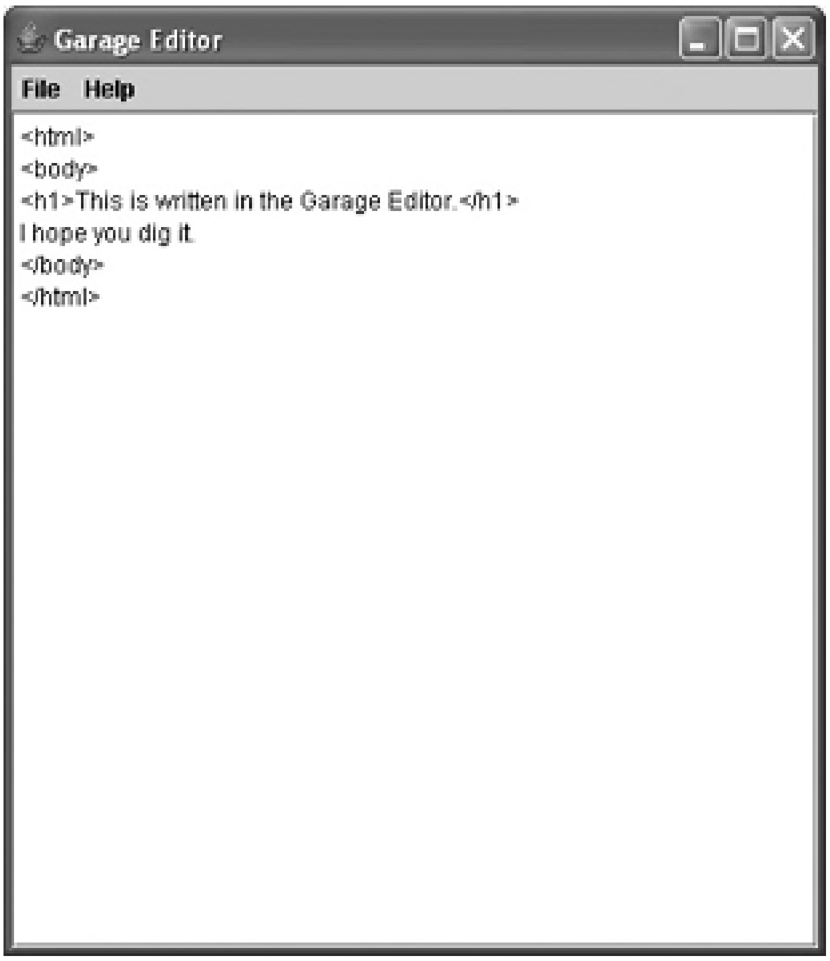
   Add them up (products are in parens):

   (8) + 1 + (2) + 1 + (2) + 1 + (2) + 1 + (2) + 1 + (2) + 1 + (2) + 1 + (2) + 1 = 30

   Divide by 10: 30/10 = 3. The number is good. (Because 30 MOD 10) = 0

Running the application and entering some text looks something like what you see in Figure 35.1.

Figure 35-1. The running text editor.



```
<html>
<body>
<h1>This is written in the Garage Editor.</h1>
I hope you dig it.
</body>
</html>
```

This looks like a lot of code, but there are also a lot of comments explaining what's going on.

GarageEditor.java

```
/**<p>

 * Presents a simple editor that allows

 * you to create a new file, open a file,

 * and save a file.

 * <p>

 * The chief benefit here is that you can see

 * how to interact with a meaningful program

 * without getting bogged down by layout manager
 * details.
 * <p>
 * Illustrates how to create a menu and menu items
 * <p>
 * Shows how to attach commands to be invoked when
 * an item is clicked.
 * <p>
 * Shows how to use common keystrokes for commands (ie,
 * press CTRL+S on the keyboard to Save the file).
 * <p>
 * Shows how to read files in and out for editing.
 * <p>
 * Shows how to use OK/Cancel dialogs
 * <p>
 * Shows how to use user preferences
 * </p>
 * Could benefit from threading to improve
 * responsiveness.
 * @author Eben Hewitt
 * @see JFrame
 * @see JTextArea
 * @see JOptionPane
 **/

import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.prefs.*;

public class GarageEditor extends JFrame {

   //holds user preferences
private Preferences prefs = null;

   //the user will type into this
   private JTextArea textArea;

   //lets you easily specify where you want
```

```
messages to go.
private static final int STDOUT = 0;
private static final int ALERT = 1;

//hold File menu commands
private static final String NEW_CMD = "New";
//default location at which to open window
private static final int DEF_WINDOW_X_LOCATION = 150;
private static final int DEF_WINDOW_Y_LOCATION = 150;

public GarageEditor() {
  // Add a menu bar and a JTextArea to the
  // window, and show it on the screen. The
  // first line of this routine calls the
  // constructor from the superclass to specify
  // a title for the window. The pack() command
  // sets the size of the window to
  // be just large enough to hold its contents.
      super("Garage Editor");

     //make a menu bar to hold menus
        JMenuBar menuBar = new JMenuBar();
     //add the File menu to it
        menuBar.add(makeFileMenu());
     //add the About menu to it
        menuBar.add(makeHelpMenu());

       //put the completed menu bar into
          the frame
        setJMenuBar(menuBar);

       //create a new JTextArea
       //give it some default text and set its size
        int rows = 25;
        int cols = 35;
       textArea = new JTextArea("choose
                file...",
                rows, cols);
        //white by default anyway, but highlights it
         textArea.setBackground(Color.WHITE);

         textArea.setMargin(new Insets(3,3,3,3));

          //make a set of scrolling controls to hold
          //the text area
          JScrollPane scroller =
                        new JScrollPane(textArea);

         //put the scroller into the content pane
         setContentPane(scroller);
         //stop app when user closes window
          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        pack();

      doLocation();
      //show the frame
      setVisible(true);
  }

 /**
  * Sets up the location of the window using
  * preferences data. Opens the editor in the
  * last location that the user had it in. If
  * not set, use a default value defined in
  * class constants above.
  */
 private void doLocation() {
   //get reference to preferences
   Preferences prefs = Preferences.userNodeForPackage(
   this.getClass());

   //retrieve the int. if there is no int stored
   //in prefs of the given string name, use the
   default
   int xLocation = prefs.getInt("WINDOW_X_LOCATION",
   DEF_WINDOW_X_LOCATION);
```

```
    int yLocation = prefs.getInt("WINDOW_Y_LOCATION",
    DEF_WINDOW_Y_LOCATION);

    //open the window at either the last
    //user location, or the default
    setLocation(xLocation,yLocation);
  }

    /**
     * Makes the File menu
     * @return JMenu containing the New, Open, Save,
     * and Exit items
     */
    private JMenu makeFileMenu() {
        //get the action command from the menu
        //to determine what the user wants to do
        //and call a separate method to do the
        //dirty work
        ActionListener listener = new
                  ActionListener() {
        public void actionPerformed(ActionEvent
                  event) {
           String command = event.getActionCommand();
           if (command.equals("New")){
                doNew();
           }
           else if(command.equals("Open...")){
                doOpen();
           }
           else if(command.equals("Save...")){
                doSave();
           }
           else if(command.equals("Close")){
                doClose();
           }
           else if (command.equals("Exit")){
                doExit();
           }
        }
        };

     JMenu fileMenu = new JMenu("File");

         //NEW
     JMenuItem newCmd = new JMenuItem("New");
     newCmd.setAccelerator(KeyStroke.getKeyStroke
           ("ctrl N"));
     newCmd.addActionListener(listener);
     fileMenu.add(newCmd);

     //OPEN
     JMenuItem openCmd = new JMenuItem("Open...");
     openCmd.setAccelerator(KeyStroke
     .     getKeyStroke("ctrl O"));
     openCmd.addActionListener(listener);
     fileMenu.add(openCmd);

     //SAVE
     JMenuItem saveCmd = new JMenuItem("Save...");
     saveCmd.setAccelerator(KeyStroke
     .     getKeyStroke("ctrl S"));
     saveCmd.addActionListener(listener);
     fileMenu.add(saveCmd);

     //CLOSE
     JMenuItem closeCmd = new JMenuItem("Close");
     closeCmd.setAccelerator(KeyStroke
     .     getKeyStroke("ctrl L"));
     closeCmd.addActionListener(listener);
     fileMenu.add(closeCmd);

     //QUIT
     JMenuItem exitCmd = new JMenuItem("Exit");
     exitCmd.setAccelerator(KeyStroke
         .getKeyStroke("ctrl E"));
     exitCmd.addActionListener(listener);
     fileMenu.add(exitCmd);
```

```
       return fileMenu;
        }

    /**
     * Creates the About Menu
     * @return JMenu The Help menu containing the
     * About item.
     */
  private JMenu makeHelpMenu() {
      //get the action command from the menu
      //to determine what the user wants to do
      //and call a separate method to do the
      //dirty work
      ActionListener listener = new ActionListener() {
      public void actionPerformed(ActionEvent event) {
            String command = event.getActionCommand();
            if (command.equals("About"))
                   doAbout();
            }
       };
  JMenu helpMenu = new JMenu("Help");

    //ABOUT
  JMenuItem aboutCmd = new JMenuItem("About");
     aboutCmd.setAccelerator
            (KeyStroke.getKeyStroke("ctrl A"));
  aboutCmd.addActionListener(listener);
  helpMenu.add(aboutCmd);

  return helpMenu;
}

private void doAbout() {
  JOptionPane.showMessageDialog(this,
            "Java Garage Hardcore\nWhoever 2004");
 }

 private void doNew() {
```

```
 // Carry out the "New" command from the File menu
 // by clearing all the text from the JTextArea.
       textArea.setText("");
 }

private void doSave() {
   // Carry out the Save command by letting the user
   // specify an output file and writing the text
   // from the JTextArea to that file.
        File file; // The file that the user wants
                      to save.
   JFileChooser fd; // File dialog that lets the
                         //user specify the file.
        fd = new JFileChooser(new File("."));
        fd.setDialogTitle("Save As...");
        int action = fd.showSaveDialog(this);

        if (action != JFileChooser.APPROVE_OPTION) {
                //user cancelled, so quit the dialog
                return;
        }

        file = fd.getSelectedFile();
        if (file.exists()) {
                 //if file already exists, ask
                 before replacing it
                 action =
                 JOptionPane.showConfirmDialog(this,
                 "Replace existing file?");

                 if (action !=
                       JOptionPane.YES_OPTION)
                       return;
        }

        try {
           //Create a PrintWriter for writing to the
           //specified file and write the text from
           //the window to that stream.
```

```
            PrintWriter out = new PrintWriter(new
                    FileWriter(file));
            String contents = textArea.getText();
            out.print(contents);
            if (out.checkError())
                throw new IOException(
                   "Error while writing to file.");
                    out.close();
        }
        catch (IOException e) {
                // Some error has occurred while
                trying to write.
                // Show an error message.
                    JOptionPane.showMessageDialog(this,
                 "IO Exception:\n" +
                 e.getMessage());
        }
    }

    private void doOpen() {
        //open the file the user selected by
        //printing its contents to the text area

        //will hold the user's file
            File file;
            //creates the complete dialog the user
            //needs to select file
            JFileChooser fd;
            //use the current directory
            //(specified as ".")
            //as a starting place
            fd = new JFileChooser(new File("."));
            fd.setDialogTitle("Select a File...");
            int action = fd.showOpenDialog(this);

            if (action != JFileChooser.APPROVE_OPTION) {
                    return;
            }
            //set the file to what the user selected
            file = fd.getSelectedFile();
            try {
                //reset the text area to be blank
                textArea.setText("");
                //read in the selected file
                BufferedReader in = new BufferedReader(
                            new FileReader(file));
                String lines = "";

                int lineCt = 0;
                String str;

                while ((str = in.readLine()) != null) {
                    lines += str + "\n";
                }

                textArea.setText(lines);
                in.close();

        } catch (Exception e) {
                JOptionPane.showMessageDialog(this,
                "Unable to open file:\n" + e.getMessage());
                }
        }

        private void doClose(){
                Object[] options = { "OK", "CANCEL" };
                int action =
                    JOptionPane.showOptionDialog(null,
                          "OK to close without
                          saving?", "Closing",
                          JOptionPane.DEFAULT_OPTION,
                          JOptionPane.WARNING_MESSAGE,
                          null, options, options[0]);

                if (action == JOptionPane.OK_OPTION) {
                    //reset the text area
```

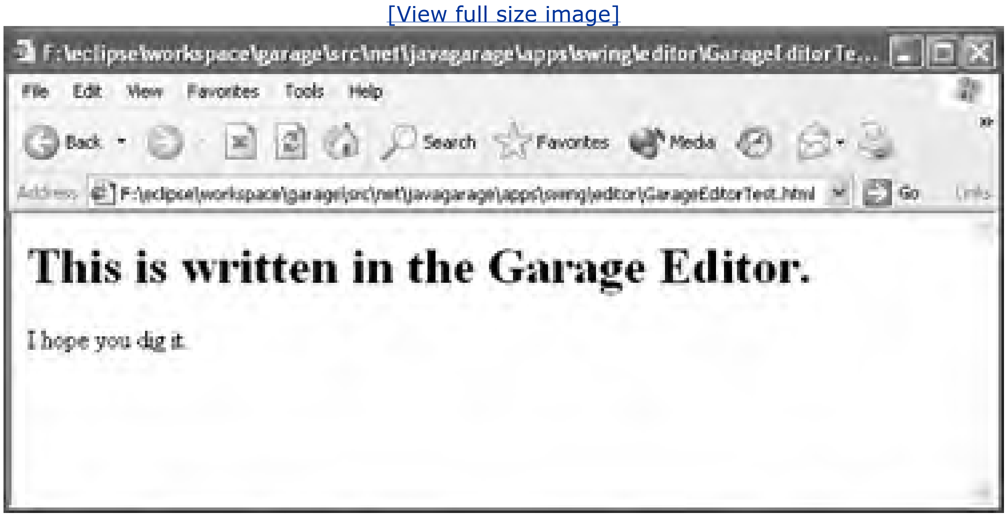
```
                textArea.setText("");
                return;
            } else {
                //do nothing and leave their
                file alone
                return;
            }
    }

  /**
   * Exit the application and stop the VM.
   */
  private void doExit() {
      //store the current location of the window
      //as a user preference so we can open it here
      //next time
      Preferences prefs =
         Preferences.userNodeForPackage
         (this.getClass());
      prefs.putInt("WINDOW_X_LOCATION", getX());
      prefs.putInt("WINDOW_Y_LOCATION", getY());
      // stop the application
      System.exit(0);
   }

    /**
     * Allows you to easily switch logging locations.
     * You could add a FILE case here too for moving
     * into production.
     * @param msg
     * @param type
     */
    private void log(String msg, int type){
        switch (type){
        default : //fall through
        case STDOUT:
        System.out.println("—>" + msg);
        break;
        case ALERT:
                JOptionPane.showMessageDialog(this, msg);
                break;
                }
            }

        public static void main(String[] args) {
            //start the application
                new GarageEditor();
        }
    }
```

Results

We can enter some text, and then save the file just as you'd expect. Let's do that and then open it in a browser to prove that it works (see Figure 35.2).

Figure 35.2. Entering text into the editor is a breeze....

[View full size image]

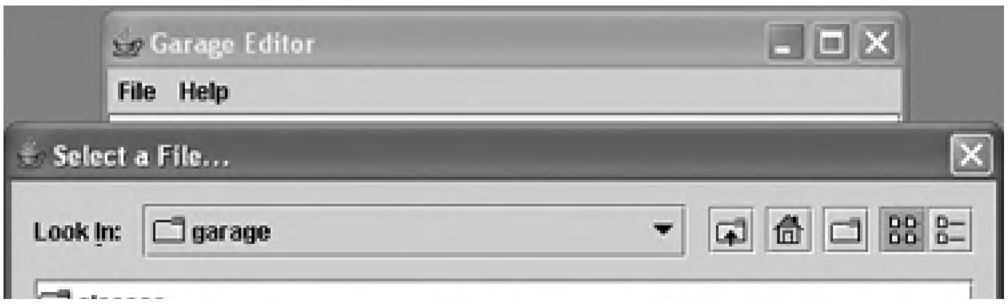

F:\eclipse\workspace\garage\src\net\javagarage\apps\swing\editor\GarageEditorTe...
File Edit View Favorites Tools Help
Back Search Favorites Media
Address F:\eclipse\workspace\garage\src\net\javagarage\apps\swing\editor\GarageEditorTest.html Go Links

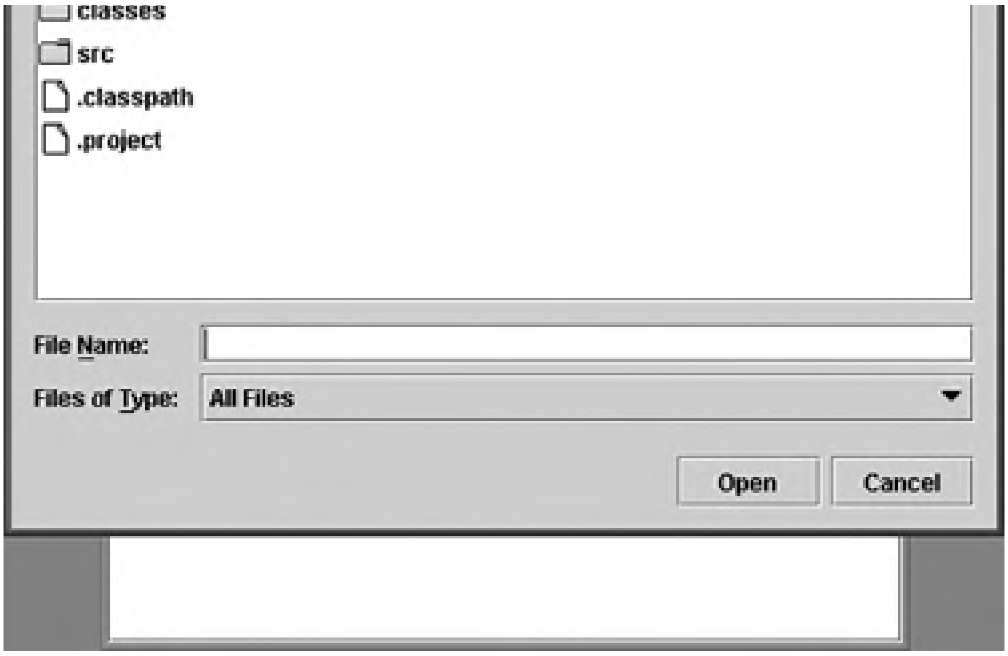
This is written in the Garage Editor.

I hope you dig it.

We also use the file chooser to browse to a local file (see Figure 35.3).

Figure 35.3. Using the JFileChooser.

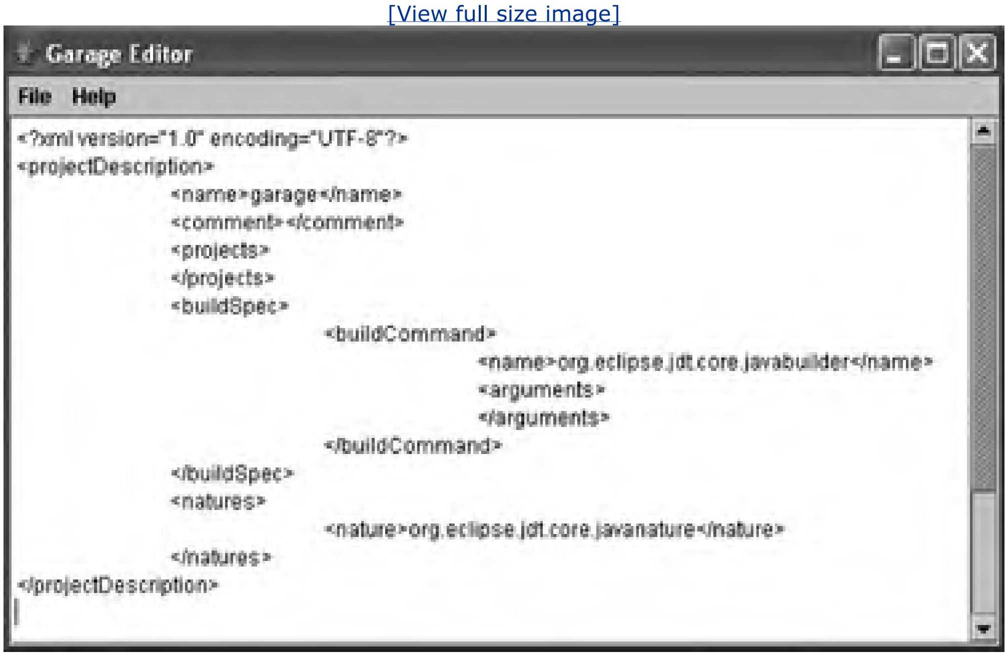




If we open an existing file written in XML, such as the Eclipse project file, we see that the program honors the line breaks, tabs, and so forth that were already present in the file (see Figure 35.4).

Figure 35.4. The editor honors formatting.

[View full size image]

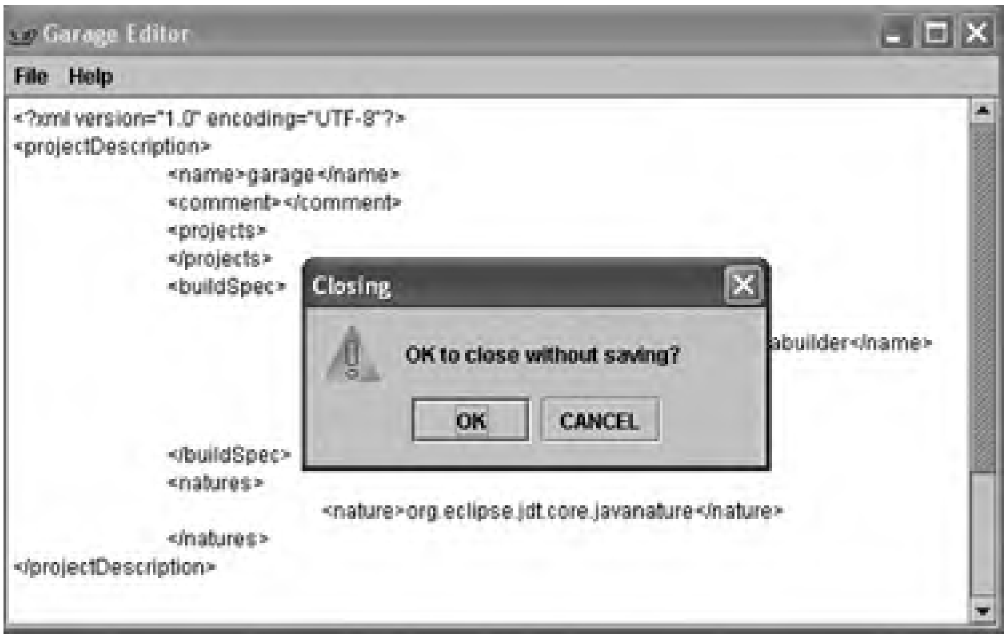


```
<?xml version="1.0" encoding="UTF-8"?>
<projectDescription>
		<name>garage</name>
		<comment></comment>
		<projects>
		</projects>
		<buildSpec>
				<buildCommand>
						<name>org.eclipse.jdt.core.javabuilder</name>
						<arguments>
						</arguments>
				</buildCommand>
		</buildSpec>
		<natures>
				<nature>org.eclipse.jdt.core.javanature</nature>
		</natures>
</projectDescription>
```

If you choose to close an open document without saving, the app asks if you're sure you know what you're doing. If you choose OK, changes are not saved; if you choose Cancel, you are returned to the editor (see Figure 35.5).

Figure 35.5. The JOptionPane at work.

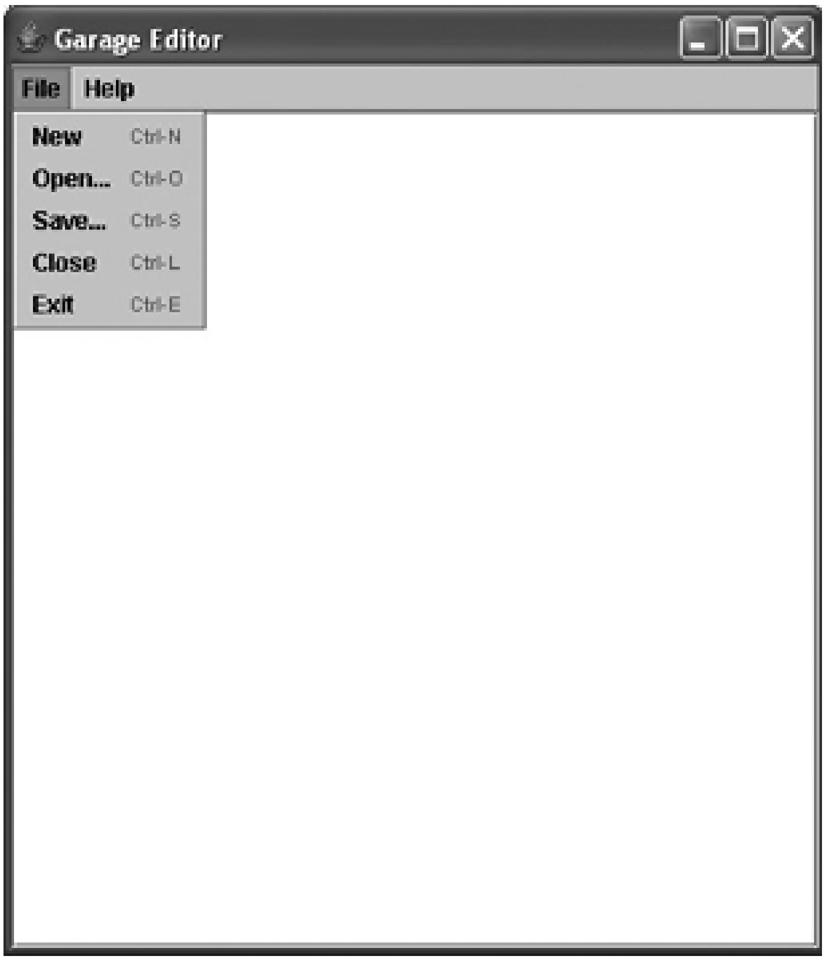
[View full size image]



```
<?xml version="1.0" encoding="UTF-8"?>
<projectDescription>
        <name>garage</name>
        <comment></comment>
        <projects>
        </projects>
        <buildSpec>
                                    ...builder</name>
        </buildSpec>
        <natures>
                <nature>org.eclipse.jdt.core.javanature</nature>
        </natures>
</projectDescription>
```

When you perform an action on the editor, you see the toolbar containing the File and Help menus. Choose a menu, and you see its items (see Figure 35.6).

Figure 35.6. The File menu and its items, including the mnemonics.



Application: RSS Aggregator

RSS is an acronym for Rich Site Summary (or, some will tell you, Really Simple Syndication). It is an XML-based technology for syndicating the content of news and blog Web sites. A program that is aware of RSS is called an aggregator. If you see a little orange XML icon in a Web page, it means that that content is available in an RSS feed.

In this episode, we'll write an RSS aggregator. It will pull together many of the technologies we have discussed, including GUI apps, and introduce some advanced topics we haven't discussed, such as networking, Collections, and caching.

Before we talk about what we'll write, let's figure out a bit more about RSS.

RSS is an XML format that is largely concerned with defining titles, news items, content, images, descriptions, and other structural information that news articles conform to.

There are many different versions of RSS. The first was .90, which was quickly obsolete. It was followed by version .91, wonderfully simple and straightforward to use, which makes it very popular for basic data. Version .92 allows for more metadata than previous versions. The common element between these versions of RSS is that they are each controlled by a single vendor (Netscape in the case of .90, and UserLand for the others). Version 1.0, however, is controlled by the RSS development working group. The current version is 2.0, and offers support for extensibility modules, a stable core, and active development. Note that this version is upwards compatible—that is, any valid 91 source is also a valid 2.0 feed.

The point in considering these different versions is that they are quite different indeed, and you are likely to encounter the many different versions on your travels. The application we write will allow you to create an XML file that defines the URLs of the RSS feeds you are interested in.

Here is what version .91 looks like:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<rss version="0.91">

<channel>

<title>WriteTheWeb</title>

<link>http://writetheweb.com</link>

<description>News for web users that write

    back</description>

<language>en-us</language>

<copyright>Copyright 2000, WriteTheWeb team.

    </copyright>

<managingEditor>editor@writetheweb.com</managingEditor>

<webMaster>webmaster@writetheweb.com</webMaster>

<image>

<title>WriteTheWeb</title>

<url>http://writetheweb.com/images/mynetscape88.gif</url>

<link>http://writetheweb.com</link>

<width>88</width>

<height>31</height>

<description>News for web users that write
```

```
   back</description>
</image>
<item>
<title>Giving the world a pluggable Gnutella</title>
<link>http://writetheweb.com/read.php?item=24</link>
<description>WorldOS is a framework on
which to build programs
that work like Freenet.</description>
</item>
</channel>
</rss>
```

Here is a version of RSS 2.0:

```
<?xml version="1.0" ?>
<!— RSS generated by UserLand Frontier v9.0 on 11/30/2003;
7:13:00 PM Eastern —>
<rss version="2.0">
<channel>
  <title>Scripting News</title>
  <link>http://www.scripting.com/</link>
  <description>All scripting, all the time, forever.</
description>
   <language>en-us</language>
   <copyright>Copyright 1997-2003 Dave Winer</copyright>
   <pubDate>Sun, 30 Nov 2003 05:00:00 GMT</pubDate>
   <lastBuildDate>Mon, 01 Dec 2003 00:13:00
          GMT</lastBuildDate>
   <docs>http://blogs.law.harvard.edu/tech/rss</docs>
   <generator>UserLand Frontier v9.0</generator>
   <managingEditor>dwiner@cyber.law.harvard.edu</
```

```
managingEditor>
   <webMaster>dwiner@cyber.law.harvard.edu</webMaster>
<item>
   <description><img
src="http://monster2.scripting.com/z/
images/archiveScriptingCom/2003/11/30/xmasTree.gif"
width="44"
height="66" border="0" align="right" hspace="15"
vspace="5"
alt="A picture named xmasTree.gif">Thanks to
Vitamin C, <a href="http://www.herbs.org/greenpapers/
echinacea.html">echinacea</a> and lots of rest, my first cold
of the winter appears to be over.</description>
   <pubDate>Mon, 01 Dec 2003 00:05:11 GMT</pubDate>
   <guid>http://archive.scripting.com/2003/11/
30#When:7:05:11PM</guid>
   <category>/Boston/Weather</category>
   </item>
</channel>
</rss>
```

This version is obviously more verbose, but you see that the basic elements—channel, item, image—are intact. What is added is extra metadata, such as publication date, last build date, and so on.

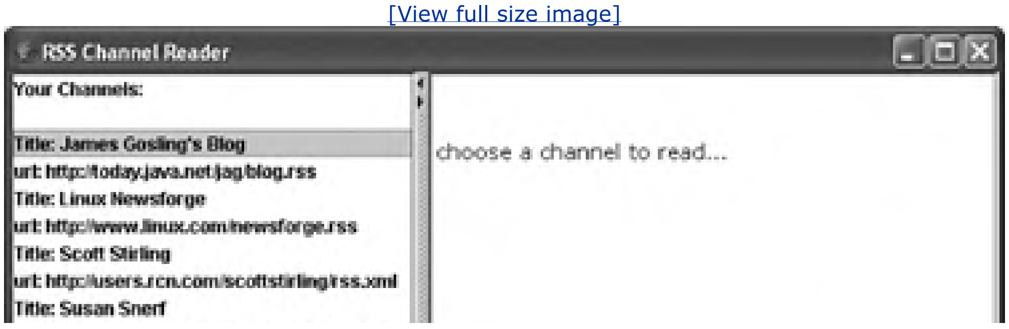
In our application, let's create an XML document that indicates the URLs of the RSS news feeds we want to view. Then, our Java app will display this list of links by reading in this XML document, and fetch the RSS data when we click on the link.

However, because it is native XML, it is not very readable in this format, and wasn't meant to be viewed this way. So, we use a basic XSL stylesheet that accounts for many of the common RSS tags in order to transform the source into HTML.

Our completed app will look like Figure 35.7 when we launch it.Click the URL on the left side. The app will open a network connection and retrieve the data at the other end, transform the data using the XSL stylesheet, and save the resulting HTML to a cache. Every subsequent time in this session that you click on that URL, the app retrieves the content from the cache.

Figure 35.7. The RSS Aggregator app displays channels to choose from for news reading.

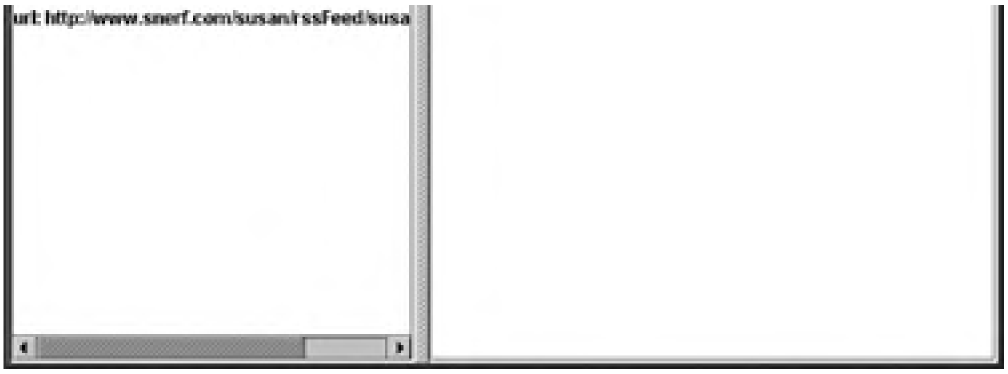
[View full size image]



Your Channels:

Title: James Gosling's Blog
url: http://today.java.net/jag/blog.rss
Title: Linux Newsforge
url: http://www.linux.com/newsforge.rss
Title: Scott Stirling
url: http://users.rcn.com/scottstirling/rss.xml
Title: Susan Snerf

choose a channel to read...

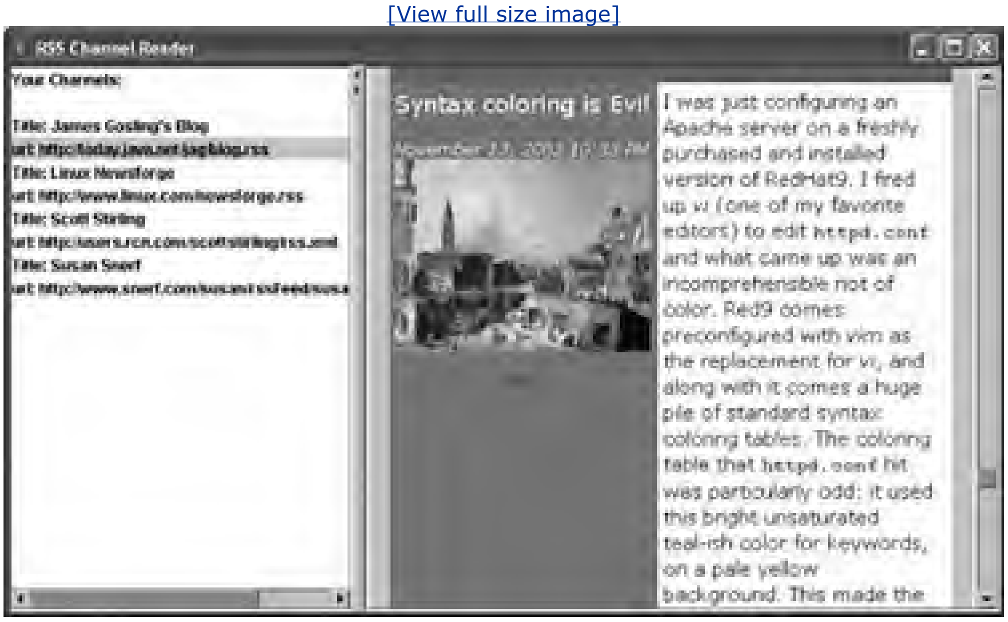


url: http://www.snerf.com/susan/rssFeed/susa

If the document contains a link, you can click on the link and the app will act as a browser, retrieve that content, and display it (see Figure 35.8). Note that to save space, the app does not check the version of the RSS document and do anything special based on that info, nor is the stylesheet anything more than rudimentary. For those reasons, most RSS feeds you throw at it should display, but they will likely display with varying degrees of beauty.

Figure 35.8. Browsing to a complete entry in James Gosling's blog.

[View full size image]



Your Channels:

Title: James Gosling's Blog
url: http://today.java.net/jag/blog.rss
Title: Linux Newsforge
url: http://www.linux.com/newsforge.rss
Title: Scott Stirling
url: http://users.rcn.com/scottstirling/rss.xml
Title: Susan Snerf
url: http://www.snerf.com/susan/rssFeed/susa

Syntax coloring is Evil
November 13, 2003 10:33 PM

I was just configuring an Apache server on a freshly purchased and installed version of RedHat9. I fired up vi (one of my favorite editors) to edit httpd.conf and what came up was an incomprehensible riot of color. Red9 comes preconfigured with vim as the replacement for vi, and along with it comes a huge pile of standard syntax coloring tables. The coloring table that httpd.conf hit was particularly odd: it used this bright unsaturated teal-ish color for keywords, on a pale yellow background. This made the

If you click on a title, your browser will clear. As mentioned earlier, clicking on the same title a second time in the same session retrieves data from the cache. The program's output to the standard out illustrates this:

```
F:\eclipse\workspace\garage\src\apps\rss\source\channels.xml

File

F:\eclipse\workspace\garage\src\apps\rss\source\cache\blog.rss

not in cache

F:\eclipse\workspace\garage\src\apps\rss\source\cache\blog.rss

Getting file
```

```
F:\eclipse\workspace\garage\src\apps\rss\source\cache\blog.rss

from in cache
```

Let's look at the files that make up the application in order of appearance.

First, NewsReader.java contains the main method. It starts the application by creating a new ReaderFrame and displaying it.

NewsReader.java

```
package net.javagarage.apps.rss;


/**
 * Starts the application by displaying the ReaderFrame.
 * @author eben hewitt
 * @see ReaderFrame
 */
public class NewsReader {

public static void main(String[] args) {
    start();
}

public static void start() {
    ReaderFrame frame = new ReaderFrame("RSS Channel
            Reader");
    frame.pack();
    frame.setVisible(true);
}
}
```

It's pretty straightforward. Because the main job of NewsReader.java is to create a ReaderFrame, let's take a look at his job.

ReaderFrame.java

```
package net.javagarage.apps.rss;


import java.awt.Dimension;

import java.io.BufferedReader;

import java.io.BufferedWriter;

import java.io.FileWriter;

import java.io.IOException;

import java.io.InputStreamReader;

import java.io.PrintWriter;

import java.net.MalformedURLException;

import java.net.URL;

import java.util.Map;

import java.util.StringTokenizer;

import java.util.Vector;


import javax.swing.JEditorPane;

import javax.swing.JFrame;

import javax.swing.JLabel;

import javax.swing.JList;

import javax.swing.JScrollPane;

import javax.swing.JSplitPane;

import javax.swing.ListSelectionModel;

import javax.swing.event.HyperlinkEvent;

import javax.swing.event.HyperlinkListener;

import javax.swing.event.ListSelectionEvent;

import javax.swing.event.ListSelectionListener;


/**

 * <p>

 * The frame that holds the content of the app.

 * It encapsulates the necessary JFrame code and

 * gathers the content.

 * Note in particular the getRSSData() method, which

 * demonstrates a pretty good use of caching. The

 * data is retrieved only when the user

 * requests it, and it is fresh the first time, but

 * subsequent reads of the same file in the same
 * session are read from the cache,
 *  which stores the transformed result.
 * </p>
 * <p>
 * Created: Sep 12, 2003 4:38:09 PM
 * <p>
 *
 * @author HewittE
 * @since 1.2
 * @version 1.0
 * @see
 */
public class ReaderFrame extends JFrame implements
ListSelectionListener, HyperlinkListener {
private Vector imageNames;
   private JLabel channelLabel = new JLabel();
private JEditorPane contentPane;
   private JList list;
   private JSplitPane splitPane;
   private String root = RSSKeys.RESOURCE_ROOT;

   /**
    * Constructor. See inline comments
    * @param title The app name that appears in the
    * window's upper left corner.
    */
public ReaderFrame(String title) {
super(title);
initFrame();

        //make the channel list
        list = new JList(getChannelList());
        //allow display of only one item at a time

list.setSelectionMode(ListSelectionModel.
                  SINGLE_SELECTION);
        //select the first item in the list
        //when the app starts
        list.setSelectedIndex(2);
        list.addListSelectionListener(this);
        //to hold list of available channels
        JScrollPane listScrollPane =
                    new JScrollPane(list);
try {
//create the right-hand side to show the transformed HTML
contentPane = new JEditorPane("file:///" +
RSSKeys.RESOURCE_ROOT + "start.html");
} catch (IOException ioe) {
System.out.println(ioe.getMessage());
}
//to hold the contents of selected channel
contentPane.setEditable(false);
contentPane.addHyperlinkListener(this);
JScrollPane channelScrollPane = new JScrollPane(contentPane);

        //make a split pane with two scroll panes
        splitPane = new
JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
                                   listScrollPane,
channelScrollPane);
        //allows the divider to be toggled on or off
        //with one click
        splitPane.setOneTouchExpandable(true);
/*set minimum sizes for the two components
in the split pane.
*/
Dimension minimumSize = new Dimension(300, 300);
splitPane.setMinimumSize(minimumSize);

//set preferred size for the split pane.
splitPane.setPreferredSize(new Dimension(650, 400));
//set the divider at the end of the channel list pane
```

```
splitPane.setDividerLocation(250);
//put the content into the frame
getContentPane().add(this.getSplitPane());
}

/**
 * do typical things to set up the window
 *
 */
public void initFrame(){
setDefaultLookAndFeelDecorated(true);
//shut down the app when the window is closed
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
//put window somewhere
setLocation(300,300);
}

/**
 * This method is separate from the parseList in
 * order to help keep the implementation-specific
 * details to a minimum in how we populate the list.
 * That is, we need a Vector of Strings.
 * @return Vector The list of channels as
 * user-friendly text.
 */
public Vector getChannelList() {
String channelList = XMLReader.transform(RSSKeys.LIST_XML,
RSSKeys.LIST_XSL);
return parseList(channelList);
}

/**
 * @param s
 * @return Vector The list items as parsed from the
 * channels.xml document
 */
protected static Vector parseList(String s) {
//default size of list will be 6
```

```
Vector channels = new Vector(6);

channels.add("Your Channels: ");

StringTokenizer tokenizer = new StringTokenizer(s, "\n");

while (tokenizer.hasMoreTokens() ) {

String channelName = tokenizer.nextToken();

channels.addElement(channelName);

}

return channels;

}


/**

 * Gets the split pane that holds the two

 * JScroll panes

 * @return JSplitPane The main frame

 */

public JSplitPane getSplitPane() {

  return splitPane;

}


/**

 * We are implementing the ListSelectionListener,

 * which means that we promise to do something when

 * the user clicks an item in the list.

 * <br>What we do is either display a clever message

 * if the user has clicked a channel name instead of

 * a URL. In the lucky event that the user clicks

 * that channel's URL instead, we pass that string

 * into the getRSSData method in order to get the

 * viewable page back.

 */

public void valueChanged(ListSelectionEvent evt) {

if (evt.getValueIsAdjusting())

   return;


//setup cache to hold transformed files
```

```
Map cache = Cache.getInstance().getCache();


JList theList = (JList)evt.getSource();
String content = "No channel selected";
String rssSite = "";
if (theList.isSelectionEmpty()) {
channelLabel.setText("Selection empty");
} else {
rssSite = (String)theList.getSelectedValue();
if (rssSite.startsWith("url:")) {
rssSite = rssSite.substring(4);
contentPane.setText(getRSSData(rssSite));
} else {
contentPane.setText("<code>");
}
}
   channelLabel.revalidate();
   }

   /**
    * The main workhorse of the app. Gets the url
    * passed in and gets its content from
    * the Internet the first time.
    * After that data has been read
    * once (per session) subsequent views are read
    * from the cache.
    *
    * @param rssSite The internet address of the XML
    * RSS feed that you want to read
    * @return The RSS transformed into HTML, which is
    * viewable in the JEditorPane
    * @see Cache
    */

   public String getRSSData(String rssSite) {
   String data;
   String result = "";
   try {
URL url = new URL(rssSite);
String xsl = RSSKeys.RSS_XSL;

//Note that the cache is a singleton
Map cache = Cache.getInstance().getCache();

BufferedReader in = new BufferedReader(new
InputStreamReader(url.openStream()));

//get only the file name, without any of the path
String fileName = url.getFile();
fileName = fileName.substring(fileName.lastIndexOf("/") + 1);

String savedXML = RSSKeys.CACHE + fileName;

if (cache.get(savedXML) == null) {
System.out.println("File " + savedXML +
       " not in cache");
PrintWriter out = new PrintWriter(new
       BufferedWriter(new
FileWriter(savedXML, false)));
//get data from URL
while((data = in.readLine())!= null) {
out.write(data);
}
out.close();
in.close();
//transform the file into HTML using XSL
result = XMLReader.transform(savedXML, xsl);
//cache the data
cache.put(savedXML, result);
} else {
System.out.println("Getting file " + savedXML +
        " from in cache");
result = (String) cache.get(savedXML);
}
```

```
  } catch (MalformedURLException mfe){
  System.out.println(mfe.getMessage());
  } catch (IOException ioe){
System.out.println(ioe.getMessage());
}

  return result;
  }
  /**
   * Receives a hyperlink event when user clicks on
   * a link that might be in one of the blog pages
   * and sets the page to that URL.
   */
  public void hyperlinkUpdate(HyperlinkEvent e) {
  URL url=e.getURL();

  if ((e.getEventType() ==
HyperlinkEvent.EventType.ACTIVATED) &&
  (url.toString().startsWith("http"))) {
try {
contentPane.setPage(url);
} catch (Exception ex) {
System.out.println(ex.getMessage());
}
  }
  }
}
```

As you can see, ReaderFrame does the main work of the application. It relies on a number of other documents. The first is channels.xml, and it is our own custom XML file that defines the names and locations of the channels that we want to view. Obviously, you can change these values to ones that suit you better.

channels.xml

```
<?xml version="1.0"?>
<channel-list>
<channel>
<title>James Gosling's Blog</title>
<url>http://today.java.net/jag/blog.rss</url>
</channel>
<channel>
<title>Linux Newsforge</title>
<url>http://www.linux.com/newsforge.rss</url>
</channel>
<channel>
<title>Scott Stirling</title>
<url>http://users.rcn.com/scottstirling/rss.xml</url>
</channel>
<channel>
<title>Susan Snerf</title>
<url>http://www.snerf.com/susan/rssFeed/
susansnerfrss.xml</url>
</channel>
</channel-list>
```

This stylesheet is used to transform the contents of the channel list.

channels.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/
Transform" version="1.0">
<xsl:output method="html" indent="yes"/>
<xsl:strip-space elements="*"/>
<xsl:template match="channel-list/channel/title">
Title: <xsl:apply-templates/> <xsl:value-of
select="title"/>
</xsl:template>
<xsl:template match="channel-list/channel/url">
```

```
<xsl:template match="channel-list/channel/url">

url: <xsl:apply-templates/><xsl:value-of select="url"/>

</xsl:template>

</xsl:stylesheet>
```

But how do we get the opening, default page? The JEditorPane reads in the start.html page, which looks like this.

start.html

```
<html>

<body>

<br>

<br>

<font face="Verdana">choose a channel to read...</font>

</body>

</html>
```

The file RSSKeys.java contains nothing other than String constants used by the application. This is an easy way here to use declarative style programming, where we don't store data that the app relies on (but which is specific to the environment) inside the program. This way, it is easy to change these values. Note that you probably want to do this business in a Properties file.

RSSKeys.java

```
package net.javagarage.apps.rss;

/**

 * @author eben hewitt

 * Holds constants that different classes need.

 * Note the naming convention of all uppercase and

 * underscores in between words for constants.

 */

public class RSSKeys {
```

```
public static final String RESOURCE_ROOT =
"F:\\eclipse\\workspace\\garage\\src\\apps\\rss\\
    source\\";
public static final String CACHE = RESOURCE_ROOT +
    "cache\\";
public static final String LIST_XML = RESOURCE_ROOT +
"channels.xml";
public static final String LIST_XSL = RESOURCE_ROOT +
"channels.xsl";
public static final String RSS_XSL = RESOURCE_ROOT +
"rssStyle.xsl";
}
```

Remember that on Windows, we need to escape the backslash character that is used as the file path separator.

The next important file is the XMLReader.java class. Its job is to read in the specified document as an input stream and transform it using the stylesheet. The XML document URL comes from the event generated in the GUI list (which itself was generated from the XML channels file), and the XSL stylesheet location is in the RSSKeys class.

XMLReader.java

```
package net.javagarage.apps.rss;


import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.MalformedURLException;
import java.net.URL;
import javax.xml.transform.Templates;
import javax.xml.transform.Transformer;
import javax.xml.transform.stream.StreamSource;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerFactory;


import java.io.StringWriter;


/**
 * <p>
 * Reads XML documents and transforms them using XSLT.
 * </p>
 * @author eben hewitt
 *
 */
public class XMLReader {


public static InputStream
          getDocumentAsInputStream(URL url)
throws IOException {
    InputStream in = url.openStream();
    return in;
   }


public static InputStream
getDocumentAsInputStream(String url)
    throws MalformedURLException, IOException {
    URL u = new URL(url);
    return getDocumentAsInputStream(u);
   }


   public static String transform(String xml,
            String xsl) {
   System.out.println(xml);
         TransformerFactory tfactory =
TransformerFactory.newInstance();
         try {


              // compile the stylesheet
              Templates templates =
tfactory.newTemplates( new
StreamSource(
              new FileInputStream(xsl)));
            StringWriter sos = new StringWriter();
            StreamResult out = new StreamResult( sos );

            Transformer transformer =
templates.newTransformer();
      transformer.transform(new StreamSource(xml), out);
            sos.close();
            String result = sos.toString();
            return result;
      }
      catch ( IOException ex ) {
         System.out.println(ex.getMessage());
      }
      catch ( TransformerException ex ) {
         System.out.println( ex.getMessage() );
      }
      return null;
   }
}
```

Here is the XSL stylesheet used to transform the content read in from a channel into pretty HTML.

rssStyle.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/
Transform" version="1.0">
<xsl:output method="html" indent="yes"/>

<xsl:strip-space elements="*"/>

<xsl:template match="rss">
    <table border="1">
    <xsl:apply-templates/>

    </table>

</xsl:template>


<xsl:template match="rss/channel">

<tr>

<td>

<a>

    <xsl:attribute name="href">



<xsl:value-of select="link"/>

</xsl:attribute>



<b>



<xsl:value-of select="title"/>



</b>

    </a>

</td>

</tr>

<xsl:apply-templates/>

</xsl:template>

<xsl:template match="rss/channel/link">

</xsl:template>


<xsl:template match="rss/channel/title">

</xsl:template>


<xsl:template match="rss/channel/description">

</xsl:template>


<xsl:template match="rss/channel/language">

</xsl:template>
```

```
<xsl:template match="rss/channel/copyright">

</xsl:template>


<xsl:template match="rss/channel/managingEditor">

</xsl:template>


<xsl:template match="rss/channel/webMaster">

</xsl:template>


<xsl:template match="rss/channel/image">

</xsl:template>


<xsl:template match="rss/channel/item">

<tr>

  <td>

  <a>



<xsl:attribute name="href">



<xsl:value-of select="link"/>



</xsl:attribute>



<b>



<xsl:value-of select="title"/>



</b>

  </a>

  </td>

  <td>
```

```
    <xsl:value-of select="description"/>

    </td>

</tr>

</xsl:template>


</xsl:stylesheet>
```

The Cache class here holds the generated content from previous visits to URLs. This is a fairly common way to implement a cache, and has the added benefit of demonstrating the Singleton design pattern.

Cache.java

```
package net.javagarage.apps.rss;


import java.util.HashMap;

import java.util.Map;

/**

 * @author eben hewitt

 * Singleton. Which means there can only be one

 * instance of this class per JVM.

 *

 * Holds a HashMap collection, which

 * allows easy retrieval of object values based on a

 * key name. So we can pass in the URL string of a

 * file name along with the content of the

 *  file, and get the file out later. The obvious

 * benefit is speed, also, singleton is a pretty

 * importantpattern and one that is used frequently

 * in a manner similar to this.

 */

public class Cache {

private Map cache = new HashMap();

private static Cache instance = new Cache();
private Cache() {}


protected Map getCache() {

return cache;

}

protected static Cache getInstance() {

return instance;

}

}
```

Application: DrawingPad: Garage Doodler

This Swing GUI application is comprised of two separate but related bits of functionality. The main thing it does is it allows the user to draw freehand on a canvas, and then captures and saves the doodle as an image file. To get this functionality, we use the ImageIO class, available since SDK 1.4. The second thing this app does is it allows you to open an image file from your hard drive and view it in the frame.

This application is worth looking over. It may seem long at first, but there are a lot of comments in this sucker. I think that Swing apps can be fairly confusing for a number of reasons. One reason is that Sun has frequently updated the APIs that are used to make GUIs. That means that you often see several different ways of doing effectively the same thing. Some of those ways may be more recent or efficient or stable, however. So you've got to be careful.

There are a lot of things that have to happen to make a Java GUI app go. If you have used Microsoft Visual Studio even for a few minutes, you can see that you can create a functional window in under a minute without writing a line of code. In Java, you have to do a lot of the writing yourself, and some components may not behave as expected. So I have commented just about every line of the application. Remember that the aim of the toolkit here is to give you something that works, so you can see how all of the pieces fit together, and something that you can build on if you want to.

Demonstrates

This application demonstrates a number of cool things, and a number of things that are important to doing real Java development. The following Swing classes are used: JFrame, JPanel, JComponent, JOptionPane, and JFileChooser. We also incorporate many classes from the older AWT package to handle coordinates, graphics, colors, events, and layout. These classes include Rectangle, Graphics, Point, Dimension, and Color. The application also demonstrates how to change the cursor from an arrow to a crosshair, and how to use keyboard shortcuts on your menus.

We see how to implement a menu with commonly required menu items that act differently than we have worked on previously. We see how to use the ImageIO class to read and write image data, and we subclass the FileFilter class to make sure that our JFileChooser only allows image file types to be opened. Use of JFileChooser to save an image is also presented.

This all means that we have to deal with mouse motion events, event handlers and listeners, see how anonymous classes are used, and how to do good subclassing. By using the JOptionPane, we show any exceptions to the user so that he can call your direct line to tell you exactly what the problem is; this is much better than burying the exception in an `out.println` statement.

Limitations and Extension Points

FRIDGE

Remember that one thing we're trying to do is make the code do the talking. That's why it's okay with me to have all of this code. I want the emphasis to be on the code itself, and let it do most of the communicating when possible. There are too many technical books that explain 20 things by showing you the API and then give you a lot of disconnected snippets of code; that often leaves you not knowing what to do when you sit down at the keyboard. It also makes it hard to integrate into a real app. This method is a key tenet in *Extreme Programming*, popularized by Kent Beck, which is an added bonus.

The application does not allow you to do a few things. First, you cannot change the background color of the pad, and you cannot change the color of the pen you use to draw with. This functionality can be added with some ease by looking into the JColorChooser API. This guy works somewhat like the JFileChooser, which should make it feel familiar after using the JFileChooser in this app. The JColorChooser provides a very rich set of controls that allow the user to pick a color using RGB values, an eyedropper, and more.

Another good extension point for this application would be to allow the user to choose a different line thickness. A simple way to do this would be to show a series of buttons on the tool bar that create the pen with the specified thickness. The way to do this is to create your desired thickness with a float value, and then call the `setStroke()`method on the Graphics2D object. That code would look something like this:

```
Graphics2D g2d = (Graphics2D)g;

float thickness = 5.0f;


// This is solid stroke, but you can also make dashed

BasicStroke stroke = new BasicStroke(thickness);

g2d.setStroke(stroke);
```

For a more user-friendly but complicated implementation, you could allow the user to choose the thickness with a JSlider control.

DrawingApp.java

```
package net.javagarage.apps.swing.draw;


import java.awt.BorderLayout;

import java.awt.Color;

import java.awt.Cursor;

import java.awt.Dimension;

import java.awt.Graphics;

import java.awt.Image;

import java.awt.Point;

import java.awt.Rectangle;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import java.awt.event.MouseAdapter;

import java.awt.event.MouseEvent;

import java.awt.event.MouseMotionAdapter;

import java.awt.image.BufferedImage;

import java.awt.image.RenderedImage;

import java.io.File;

import java.io.IOException;

import java.util.ListIterator;

import java.util.Vector;


import javax.imageio.ImageIO;
import javax.imageio.ImageIO;

import javax.swing.ImageIcon;

import javax.swing.JComponent;

import javax.swing.JFileChooser;

import javax.swing.JFrame;

import javax.swing.JLabel;

import javax.swing.JMenu;

import javax.swing.JMenuBar;

import javax.swing.JMenuItem;

import javax.swing.JOptionPane;

import javax.swing.JPanel;

import javax.swing.KeyStroke;

import javax.swing.filechooser.FileFilter;


/**<p>

 * This application allows you to do a couple

 * different related things. First, it is a GUI

 * interface that allows you to use a JFileChooser

 * dialog to select an image file type to open

 * and view.

 * <p>

 * The second thing you can do is draw a doodle onto

 * a canvas and save it as an image file.

 * <p>

 * There are several classes in this file.

 * The main class is called DrawingApp, and it allows

 * you to open image files. Another class called

 * AnImageFilter extends FileFilter, and provides a

 * filter implementation that the  JFileChooser dia

 * log uses in order to make sure that only

 * image file types are opened by the user.<br>

 * There is a class called Doodler, that registers

 * mouse events to allow you to draw on the canvas.

 * <p>

 * You might extend this to include a JColorChooser

 * dialog to allow the user to change the background

 * color of the canvas, or to change the color
 * of the pen.
 * <p>
 * thank you to rockstar pawel zurek for his very
 * helpful ideas on this app.
 * @author eben hewitt
 **/
public class DrawingApp {
//the main window
protected JFrame frame;
//the panels hold the content
protected JPanel panel;
protected JPanel contentPane;
//holds the File drop-down menu
protected JMenuBar menuBar;
//this is where the user draws
protected Doodler canvas;

private boolean alreadyHasImage = false;

//start the app
public static void main(String[] args) {
//create instance by calling default constructor
new DrawingApp();
}
//constructor
public DrawingApp(){
//make the window pretty with static method
//which must be called before instantiating the window
JFrame.setDefaultLookAndFeelDecorated(true);
//now make the window
frame = new JFrame();
//this title will appear in upper-left of frame
frame.setTitle("Garage Draw Pad");
//initialize the frame to dispose of the jvm
//instance it is using when user closes the window
frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
//instantiate the panel
panel = new JPanel();
//give it a really simple layout type so we can
```

```
//add stuff to the panel

panel.setLayout(new BorderLayout());

//make it be 300x300 when it opens

panel.setPreferredSize(new Dimension(300,300));

//add the working area to the window

//in AWT we didn't have to do this.

//we have to do it in Swing because a JFrame

//has a number of layers, and the add call will be

//ignored on the root pane. this makes sure we get

//the layer of the frame where our stuff happens, and

//not where frame management is going on.

frame.getContentPane().add(panel);

/*

 * strangely, in my view, we have to set the

 * background color of the <i>frame</i> to white if

 * we want the drawing we make to be saved

 * with a white background; otherwise, the

 * background will be gray, instead of the white the

 * user sees when he clicks New. that's because of

 * the layers in a frame.

 */

frame.setBackground(Color.WHITE);


//create the menu

menuBar = new JMenuBar();

//call our method that makes the menu and then

//add the menu to the menuBar

menuBar.add(makeFileMenu());

//must do this to work with it

menuBar.setOpaque(true);

//add the menu bar to the frame

frame.setJMenuBar(menuBar);


//put it at this x,y location on the screen

frame.setLocation(350,350);

//the pack method is inherited from java.awt.Window.

//it makes the window displayable by sizing the

//components to fit into it. it then validates

//the layout.
```

```
frame.pack();

//show the window on the screen.

//the user can now interact with it, so return from

//the constructor and wait until the user does

//something.

frame.setVisible(true);

}


/**

 * Makes the menu by adding file items to the menu.

 * It also handles the job of creating a listener

 * for each of those items (such as New, Open, etc),

 * to tell the app what action to perform when the

 * user chooses that item.

 * @return The completed menu, ready to add

 * to the JMenuBar.

 */

private JMenu makeFileMenu(){

//get the action command from the menu

//to determine what the user wants to do

//and call a separate method to do the work

//ActionListener is an interface that extends EventListener

ActionListener listener = new ActionListener() {

//when an action event occurs (the user clicks a

menu

//item) the actionPerformed(ActionEvent) method is

//called. so we override it to do what we want.

public void actionPerformed(ActionEvent event) {

String command = event.getActionCommand();

//user clicked File > New

if (command.equals("New")){

//call our custom worker

doNew();

} else if(command.equals("Open...")){

doOpen();

} else if(command.equals("Save...")){

doSave();

} else if (command.equals("Exit")){

doExit();
}

}

};

//this is the File menu that will be added to

//the menubar. so we have to add each menu item

//to it first.

JMenu fileMenu = new JMenu("File");


//NEW

JMenuItem newCmd = new JMenuItem("New");

//the user can alternatively type the control key + N

//to generate the same event as clicking

newCmd.setAccelerator(KeyStroke.getKeyStroke("ctrl N"));

//register the listener

newCmd.addActionListener(listener);

//put this item onto the File menu

//items get added in order

fileMenu.add(newCmd);


  //other commands work same way...


//OPEN

JMenuItem openCmd = new JMenuItem("Open...");

openCmd.setAccelerator(KeyStroke.getKeyStroke("ctrl O"));

openCmd.addActionListener(listener);

fileMenu.add(openCmd);


//SAVE

JMenuItem saveCmd = new JMenuItem("Save...");

saveCmd.setAccelerator(KeyStroke.getKeyStroke("ctrl S"));

saveCmd.addActionListener(listener);

fileMenu.add(saveCmd);


//QUIT

JMenuItem exitCmd = new JMenuItem("Exit");

exitCmd.setAccelerator(KeyStroke.getKeyStroke("ctrl E"));

exitCmd.addActionListener(listener);

fileMenu.add(exitCmd);
```

```
return fileMenu;

} //end makeFileMenu()


/**

 * action on NEW

 */

private void doNew() {

//do work of "New" command from the File menu

try {


//if there is a canvas already, wipe it clean

//to draw a new picture

if( canvas != null ) {

//alert user

JOptionPane.showMessageDialog(null,"This will destroy current doodle");

//gets rid of the old panel

panel.remove(canvas);

}

//make this new object on which to draw

//and set its size to 300x300

canvas = new Doodler(300, 300);


//show a white background so the user sees it.

//note that since we won't be saving

//the <i>panel</i> to an image file,

//the background of the image wouldn't be white

//with only this!

panel.setBackground(Color.WHITE);


//put the canvas in the content panel

panel.add(canvas, BorderLayout.CENTER);


//create a crosshair cursor to draw with

Cursor cursor =

Cursor.getPredefinedCursor(Cursor.CROSSHAIR_CURSOR);

//make the canvas use the cursor we just created
```

```
canvas.setCursor(cursor);

//call this to repaint the window

frame.validate();

} catch (Exception e){

System.out.println("Exception in doNew():

        " + e.getMessage());

System.exit(1);

}

}


/**

 * action on OPEN

 */

private void doOpen(){




//create new file chooser dialog box

//the dialog will use the user's home directory

//as a starting place

JFileChooser fileChooser = new JFileChooser();

//if you wanted to specify a different dir to start

//in, you could do this for example in linux:

//JFileChooser f = new JFileChooser(new

        File("/home/dude"));


//set the file chooser to only see .gif file types

//see the AnImageFilter class here for details

fileChooser.addChoosableFileFilter(new

        AnImageFilter());


fileChooser.setFileSelectionMode(JFileChooser.FILES_

        AND_DIRECTORIES);


//open the file chooser and make the frame its

//parent which means, if you close the frame, the //dialog closes too

int returnValue = fileChooser.showOpenDialog(frame);


//when user clicks open, do this

if(returnValue == JFileChooser.APPROVE_OPTION) {
```

```
//create a file object based on the chosen file

File file = fileChooser.getSelectedFile();


try{


if(alreadyHasImage) {

JOptionPane.showMessageDialog(frame,"Opening removes the current image");

//gets rid of the old panel

frame.getContentPane().remove(panel);

//make a new clean one

panel = new JPanel();

//add the panel to the content pane

frame.getContentPane().add(panel);

}


//read it in as an image

BufferedImage image = ImageIO.read(file.toURL());

//use the file to create an ImageIcon object

//and make it the value of a label that we can

//easily display on the panel

JLabel imageLabel = new JLabel(new ImageIcon(image));


/*

 * Note that by adding a doodler object to the

 * canvas, we will be able to draw on any open image

 * that is a previously saved doodle.

 * But a photo we won't be able to draw on.

 */

canvas = new Doodler(300, 300);

panel.add(canvas);


panel.add(imageLabel);


alreadyHasImage = true;

frame.validate();


} catch (Exception e){

System.out.println("Exception in doOpen(): " + e);
System.exit(1);

} //end catch

} //end if

} //end doOpen


/**

 * action on SAVE

 */

private void doSave() {

//file to be saved

File outFile;

//lets user specify where to save file

JFileChooser fileChooser = new JFileChooser();

//specify some things about the file chooser dialog

fileChooser.setDialogTitle("Save As...");

   //show the dialog and make the frame its parent

int action = fileChooser.showSaveDialog(frame);

if (action != JFileChooser.APPROVE_OPTION) {

//user cancelled, so quit the dialog

return;

}

//point the file the user chose to this object

outFile = fileChooser.getSelectedFile();


if (outFile.exists()) {

//if file exists already, make sure that the

//user wants to replace it

action = JOptionPane.showConfirmDialog(frame,

  "Replace existing file?");

if (action != JOptionPane.YES_OPTION)

return;

}

try {

//get the coordinates of the corners of the

//canvas so we have the part we want to save

Rectangle rect = canvas.getBounds();


//create an image ready for double-buffering
```

```
//from the screen area bound

//by that rectangle. This method returns null

//if the component is not displayable.

Image image = canvas.createImage(rect.width, rect.height);


//creates the context required for drawing the image

Graphics g = image.getGraphics();


//paint onto the canvas the lines created by the user

canvas.paint(g);


//save the image file using jpeg compression format

ImageIO.write((RenderedImage)image, "jpg", outFile);


    //catch any problems encountered during the save

} catch (IOException e) {

//print any exception to a messagebox (like a JS alert

//or like MessageBox.Show(...) in C#)

JOptionPane.showMessageDialog(frame,

"IOException in doSave(): " + e.getMessage());

System.out.println(e.getCause().getMessage());

System.exit(1);

}

}


/**

 * Exit the application and stop the VM.

 */

private void doExit() {

// stop the application

System.exit(0);

}

}//end DrawingApp class


/**

 * class to define the images only filter

 * we must subclass FileFilter
 */
class AnImageFilter extends FileFilter {

//find the extension of this file type
//so we know whether or not to include it
public static String getExtension(File f) {
String extension = null;
String fileName = f.getName();
int i = fileName.lastIndexOf('.');

if (i > 0 && i < fileName.length() - 1) {
extension = fileName.substring(i+1).toLowerCase();
}
return extension;
}

/* define the file types we are willing
 * to accept. also show directories
 * so the user can navigate into them
 */
public boolean accept(File f) {
if (f.isDirectory()) {
return true;
}
//we're going to save drawings with jpg compression,
//so this is for show really
String extension = getExtension(f);
if (extension != null) {
if (extension.equals("tiff") ||
extension.equals("tif") ||
extension.equals("gif") ||
extension.equals("jpeg") ||
extension.equals("jpg") ) {

return true;
} else {
return false;
}
}
```

```
return false;
}
//what the user will see in "Files of Type..."
public String getDescription(){
return "*.tif, *.tiff, *.gif, *.jpeg, *.jpg";
}
} //end AnImageFilter


/**
 * Doodler class uses AWT canvas to draw on.
 * Note that while you may instinctively want to
 * extend the AWT Canvas class here, it is not
 * necessary and will mess things up: it will make
 * your File menu inaccessible! The reason is that
 * Canvas is an AWT (meaning heavy-weight)
 * component and JMenuBar is a lightweight Swing
 * component. Remember: AWT components will ALWAYS
 * cover Swing components.
 * JPanel would also have worked here.
 */
class Doodler extends JComponent {
//these ints hold the coordinates
private int lastX;
private int lastY;
private Vector plots = null;

private Vector pointData = null;
//declare two ints to hold coordinates.
//remember that because these are class-level variables,
//they will be initialized to their default values
//(0 for int).
private int x1, y1;
private Graphics graphics = null;

//constructor
public Doodler (int width, int height) {
//call the default constructor of the superclass
```

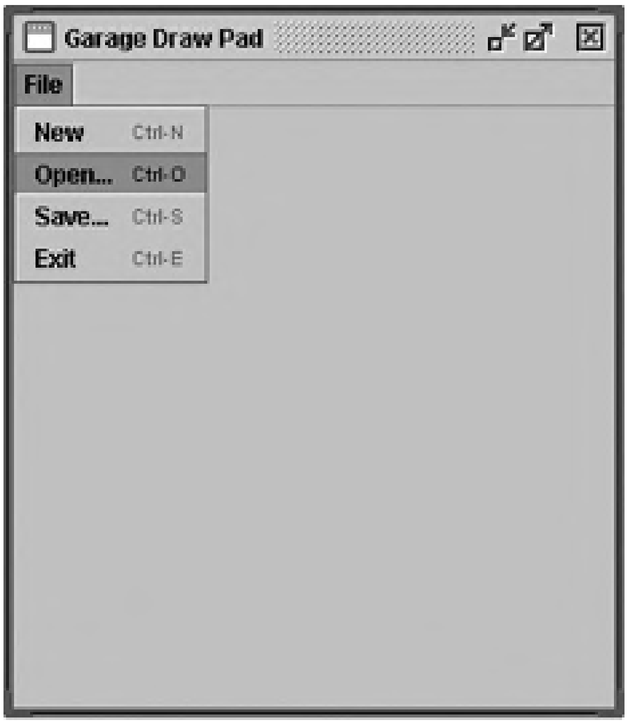
```
//(JComponent)—<i>not</i> Canvas!

super();

this.setSize(width,height);

plots = new Vector();

pointData = new Vector();

/*

 * Listen for the event that is fired when the

 * mouse button is pressed, because we don't want

 * to draw whereever the mouse goes—only when it

 * is pressed.

 */

addMouseListener(new MouseAdapter() {

public void mousePressed(MouseEvent e) {

x1 = e.getX(); y1 = e.getY();

pointData.add(new Point(x1, y1));

}

public void mouseReleased(MouseEvent e) {

plots.add((Vector)pointData.clone());

}

});

/*

 * Listen for the movement of the mouse being

dragged

 * so that we can capture each point across which it

 * was dragged and add it to our vector that stores

 * all the points; then let the drawLine method of

 * the Graphics class connect the dots.

 */

addMouseMotionListener(new MouseMotionAdapter() {

public void mouseDragged(MouseEvent e) {

int x2 = e.getX(); int y2 = e.getY();

pointData.add(new Point(x2, y2));

graphics = getGraphics();

graphics.drawLine(x1, y1, x2, y2);

x1 = x2; y1 = y2;
}
});
}
/*
 * Paints our doodle line by storing each point over
 * which the pressed mouse passes in a vector.
 * @see java.awt.Component#paint(java.awt.Graphics)
 */
public void paint(Graphics g) {
//returns a way to iterate over
//all of the elements in this list
ListIterator it = plots.listIterator();

while (it.hasNext()) {
Vector v = (Vector)it.next();
Point point1 = (Point)v.get(0);
//remember you can do more than one thing in a for
//loop's first statement
for (int i=1, size = v.size(); i < size; i+=2) {
Point point2 = (Point)v.get(i);
g.drawLine(point1.x, point1.y, point2.x, point2.y);
point1 = point2;
} // end for
} // end while
} //end paint override

}//end Doodler class
```

As you can see by reading over it, there are a few different class files in this one source file. Recall that only one of the classes may be public, and it must be named the same as your source file.

Let's walk through the code and see what kinds of things happen on your screen when you execute the program.

Figure 35.9 shows what the app looks like when you first load it and click on the File menu.

Figure 35.9. The Doodler application when the application is executed and the File menu is clicked. Notice that the frame is decorated—not just the Windows default.



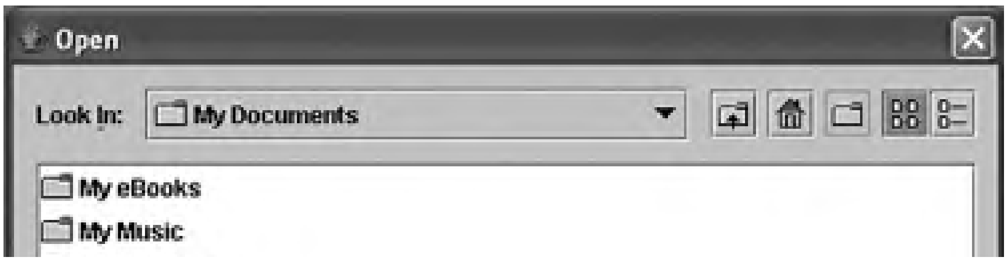
First, let's read in an image. When we type Ctrl + o on the keyboard, or click the Open menu item, this executes the doOpen() method. We did not have to put this functionality into a separate method, but it does make the JMenu business a little easier to port to a new app, and is helpful for the code reader—if she's not interested in the doOpen() method right now, it's easy to skip by it and find what she's looking for. As we'll see later with the doNew() method, there are times that you want to create a whole new object, or even a new thread, and let that guy take over for a while.

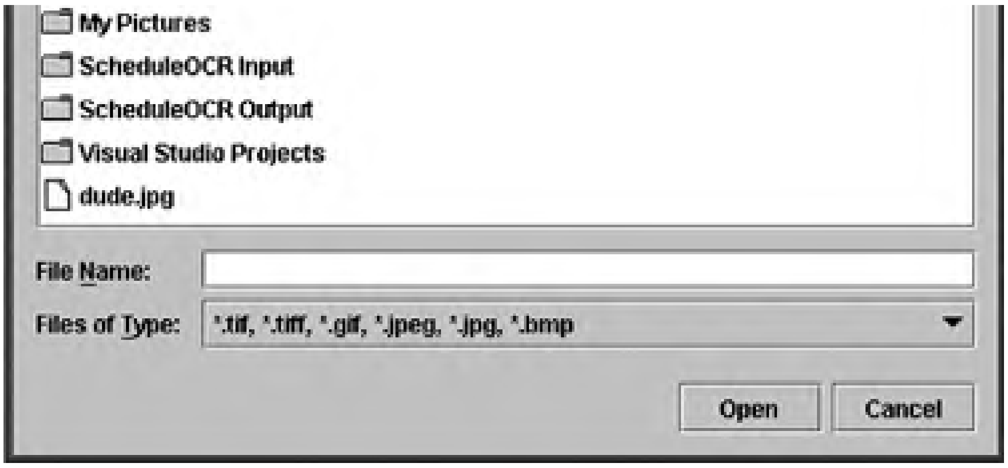
FRIDGE

The DrawingPad app won't read in a bitmapped image. The ImageIO class gets an ImageReader to decode the URL passed to it. The way ours is written, we don't decode a .bmp, so null will be returned by the static call to ImageIO.read(file.toURL());. As a consequence, a NullPointerException will be thrown if you try do that.

But back to our regularly scheduled program. We were opening a file. To do this, we get a JFileChooser dialog, which is a standard part of the API (see Figure 35.10). This is a really terrific component because it is lightweight and has a lot of functionality. After the user selects the file he's interested in, we use a BufferedImage reference to read in the data using an ImageInputStream and create it as an ImageIcon, which we pass to the JLabel constructor. We then add the JLabel to the panel, and ask the container to redraw itself.

Figure 35.10. The JFileChooser is a great component for both opening and saving files of all types. Notice that the filter is applied, and so we only see folders, and files of one of our pre-defined extension (image) types.

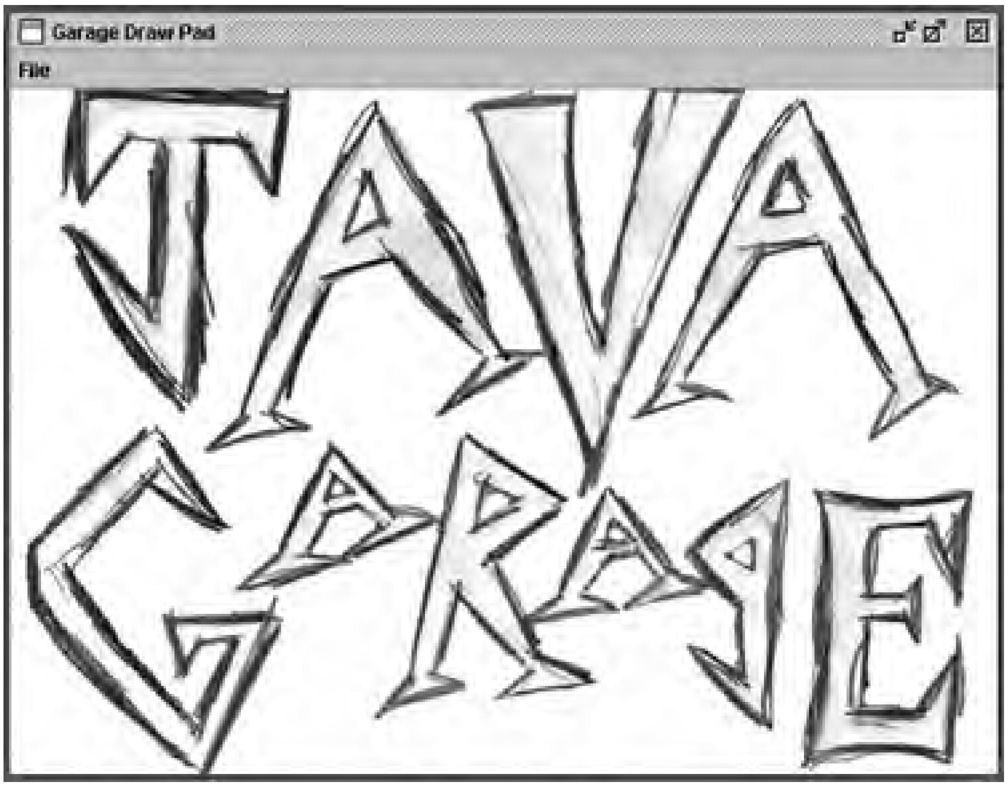




Note that the JFileChooser opens in the default user location; in the case of Windows, this is My Documents. We could also specify a different location in the constructor.

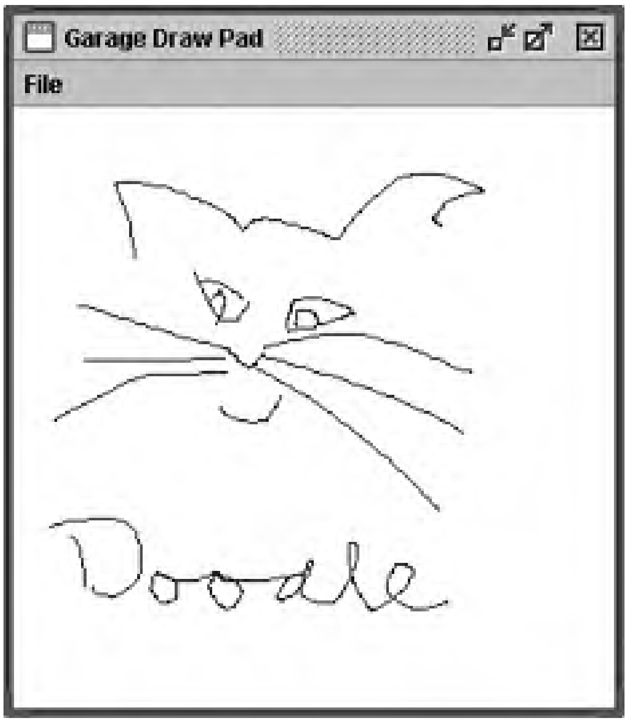
Now, you'll notice that the image may be larger than our 300x300 frame size, in which case we can just drag a corner of the window, and it will automagically resize itself to fit the image (see Figure 35.11). We could have used a JScrollPane to allow the user to view only part of the image, but that seemed like not-very-useful functionality for the added complexity. Also, we've already seen how to use JScrollPane in the RSS newsreader app.

Figure 35.11. The image you open may be larger than the frame, in which case you can drag any one side of the frame to resize the viewing area to snugly fit the image.



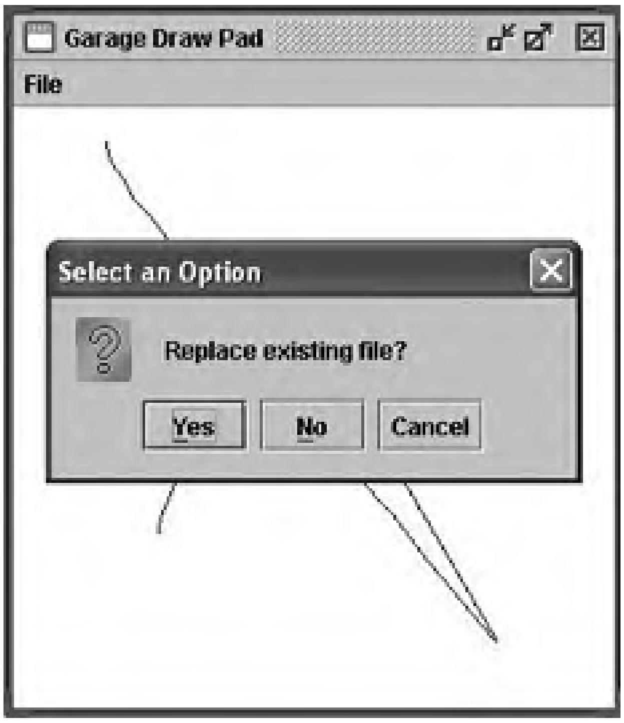
It would be nice if we could draw on a photographic image that we open, but we cannot do that. However, if you create a new doodle in this app, save it, and then open it later, you will be able to continue drawing on it (see Figure 35.12).

Figure 35.12. This is a picture of my kitty cat Doodlehead. It's an original artwork.



If you attempt to save a file using the name and path of a file that already exists, the application detects this and warns you that you are about to overwrite the existing file (see Figure 35.13). If you don't want to do that, you can hit Cancel and nothing happens. You can then click Save again and choose another name or path.

Figure 35.13. The drawing pad uses JOptionPane's showConfirmDialog method to create a dialog alert with a Cancel button.



Replace existing file?

Of course, clicking the Exit button on the menu cleanly exits the application and shuts down the Java Virtual Machine.

Ok. Thanks a million. I hope that you find a load of stuff in the toolkit you can pillage and use in your own apps.

Chapter 36. SYSTEM.EXIT…

dude yt?

ya

is that it?

huh?

this java garage is all ovah.

ya

what a rip off. i hate technical books.

da whole reason I got this Garage in da first place is so I could jazz it up my flux capacitor. I need two point two one jiggawatts!

YOU LOVE BIG PIPES I KNOW YOU DO DON"T DENY IT!

^_^

no dude my mind is like a blue screen o death on this java schlepp.

nod

uh...I remember some stuff though. we can use statements, design classes, write a console application, and write a GUI app. can read and write and images, and use regular expressions and exceptions and Streengs and formatting. and yadda yadda.

all right. that's true.

thank you.

so what happens next/

next you get More Java Garage.

ack a sequel. Java Revolutions.

that covers database access with Generics, Collections, JDBC, multithreading, internationalization, reflection, tuning, design patterns and more yee-haw like that.

but what about the web?

that's the next part. it is really a good idea to know what you're doing in java when you start working with web apps. it is a whole different deal. but everything you learned in this book will help you on the web stuff.

oh yeah? all that crepe about GUI programming???? :(

there are applets, you know. while you don't have access to the local file system in an applet, there are other ways to put that GUI programming to work on the web. for instance, web start.

whats webstart?

it is a way of packaging apps for deployment from the web that lets the user run them locally. so it is like the best of both worlds: you get all the richness of a fat client, and none of the maintenance headaches.

but what about JSPs and servlets?

that is the logical next step. the thing to do is start applying your java knowledge to JSPs. they allow you to write html code intermingled with java code and tags that execute java code so you can have dynamic applications that run on the web. now with jsp 2.0 it is a lot easier to get up and running than before. they have added a lot of tags to the regular distribution.

can I start doing web apps with the Java 5 SDK?

not the standard edition. you have to get the enterprise edition right now it's J2EE 1.4

cool. thanx

np.

you wanna get some lunch?

i was going to get me some Commander Taco.

dude. you got to stop eating in your car.

:P

need some company?

yeah, come and stuff.

i promise not to lecture you anymore about EPL soccer rules

ha. dude. you can yammer whatever the fook you wants

cool

come down.

gimme a sec with this patch install. ok. cya

ok cya

CHAR		HEX	DEC
(ETX)	^C	0x03	&#003; END OF TEXT

JAVA GLOSSARY ON STEROIDS

A

abstract

Used to specify a class that cannot be instantiated. Subclasses may inherit from an abstract class, that is, *extend* them. Also, modifies a method whose implementation you wish to defer to another class.

Abstract Windowing Toolkit (AWT)

Package of graphical user interface classes implemented in native platform component versions. AWT classes allow for the creation of buttons, frames, event handlers, and text in application interfaces. AWT's popularity has ebbed significantly in favor of the newer, more portable and lightweight Swing.

API

Application programming interface. The specification that dictates how a programmer must write to access the state and behavior of objects and classes.

applet

A (generally small) program written in Java that extends the java.applet.Applet class. By extending this class, the program becomes capable of executing in a Web browser or other device that supports applets in general. Applets helped create tremendous popularity for the Java programming language early in its life because they allow programs to be dynamically downloaded and securely executed. Applets can be viewed from the command line with appletviewer, a program of the standard SDK. Alternatively, applets can be viewed in a Web page by having the user download a plugin and referencing the applet with either the <object> or <applet> tags, both part of standard HTML. Microsoft's illegal incorporation of proprietary technology into its Java Virtual Machine seems to have had significant negative impact on the popularity of applets, as the necessary 5MB plugin download from Sun makes for a thicker client and additional user support.

argument

An argument is an item of data passed into a method. An argument may be an expression, a variable, or a literal.

array

An ordered set of data items, all of which must be of the same type. Each item in the array can be referenced by its integer position. The first element of a Java array is 0.

ASCII

Acronym for American Standard Code for Information Interchange. An ASCII code is the 7-bit numerical representation of a character, which can be understood by many different computing platforms.

B

bean

Beans are standard Java classes that conform to certain design and naming conventions (private variables and getter and setter methods). When people say, "JavaBean," they mean something that comes from the java.beans package. JavaBeans and Enterprise JavaBeans have very little relation (no more than, say, Java and JavaScript). The original purpose of beans was to make it easy for software vendors to write programs (such as IDEs) to allow users to visually manipulate the software component represented by the bean (like in VB). To achieve this, beans all share the following characteristics: Properties, Customization, Persistence, Events, and Introspection.

bit

Contracted form of *binary digit*, the smallest unit of information in a computer. A bit is capable of holding one of two possible values: on or off, commonly represented as true or false, or 0 or 1. The term is said to have been coined by the mathematician Claude Shannon.

bitwise operator

An operator that manipulates individual bits within a byte, generally by comparison, or by shifting the bits to the left or right.

block

Any Java code between curly braces. For instance: `{ int i = 0; }` is a block of code. Colloquially, developers refer to a "try/ catch block."

Boolean

The object wrapper for the primitive `boolean`.

boolean

One of the primitives in Java, a `boolean` represents either a true or false value. The default value is false. Note that boolean represents not a 1 or 0, but the literal value true or false.

bounding box

Used in graphical user interface (GUI) programming, refers to the smallest geometric area surrounding a shape.

bounds

Used to constrain the types that you can invoke on wildcards in generic method definitions. A lower bound means that the runtime value supplied for the wildcard must be higher up the inheritance hierarchy than the wildcard lower bound. An upper bound indicates that when you supply an actual type at runtime, it must be a subclass of the upper bound type. A lower bound is expressed with `? extends T`. Say you wanted to allow your wildcard to be "replaced" at runtime with only classes that are subclasses of java.lang.Exception. You would write `? Extends Exception` in your method definition. An upper bound is expressed with `? super T`. You can also use a parameter type as the bound of your wildcard, like this: `? extends E`. This means, "the argument supplied to this method at runtime must be a subclass of whatever the actual parameter type supplied is."

See also **[wildcard]**

break

A keyword in Java used to resume program execution at the next statement immediately following the current statement, except when a label is used, in which case execution returns to the label.

Byte

An object wrapper for a byte primitive.

byte

A Java keyword used to declare a primitive variable of type byte. Eight bits form one byte. A byte is a signed integer in Java; its minimum value is -27 and its maximum positive value is 27-1.

bytecode

Machine-independent code that is generated by the Java compiler and executed by the Java interpreter. The code in a .class file is bytecode, serving as input to the Java Virtual Machine. Bytecode can also be converted to native machine code by a Just In Time compiler.

C

case

A Java keyword, case is used inside a switch block to define a set of statements to be executed conditionally, depending on whether the case value matches the supplied switch value.

cast

To convert one data type to another. There are implicit primitive casts, in which you do not write the code to perform the conversion—it is done automatically. An explicit primitive cast demands that you write the code to perform the cast, because you are narrowing the primitive (putting it into a smaller container—for example, casting a long into an int). Narrowing conversions can cause loss of precision, which is why you must explicitly perform it.

catch

A Java keyword used to declare a block of statements to be executed in the event that an exception is thrown from code in the corresponding try block.

char

A Java keyword used to declare a primitive variable of type char. A char is used to represent Unicode character values. It is declared with single quote marks around the value, as in: char grade = 'a'; chars are considered 16-bit unsigned integer primitives, which means that Java allows direct conversion between char and other primitive integers. Its possible values range from 0 to 65535 (or 216-1).

class

A Java keyword representing the declaration of an implementation of a particular object. Classes are used to create instances of objects. Classes define instance and static variables and methods. They specify the interface that the class implements, and the immediate superclass that the class is extending. All behavior in Java programs occurs inside classes. Every class implicitly extends the class java.lang.Object. The class java.lang.Class can provide the runtime type of an object.

class file

The file created when a .java source code file is compiled. The name of the generated file will match the name of the source code file. The class file contains bytecode to be executed by the runtime.

class method

More commonly called a static method, a class method is one that can be invoked without reference to any particular object. Defines static methods when the execution will always be the same, regardless of the particular object invoking it. For example, a method that makes a connection to a datasource is often defined statically, because it is always performed in the same way, and is always returned the same result (a database connection) regardless of what instance calls it. Invoke class methods on the class itself. For example, the Collections class contains a number of static methods for convenience, such as sort(). You invoke it like this: Collections.sort(myArrayList);. The java.lang.Math class is another class that defines only static methods (which makes sense—the pow() method, which calculates the result of the first argument raised to the second argument, will always perform the same action).

Classpath

An environment variable indicating the location of class libraries used by the Java Virtual Machine. If you write a program that uses a particular library (for example, a JAR file that contains classes that help you convert data into PDF format), you need to put that JAR on your classpath so that the JVM can find it when it executes your code that references the classes in that JAR. Here's another common example: If you want to send an e-mail from a Java program, you can't do it using just the J2SE; you need to download J2EE, put the j2ee.jar file on your classpath, and then you can use the classes in the javax.mail package to send the e-mail.

class variable

Also known as a static variable, a class variable is defined at the class level, using the static modifier, and does not rely on any particular instance of the class to be invoked.

comment

Text that explains to yourself or other programmers what the purpose of this code is, what other code it affects, and other metadata such as the author's name and the created date. Single-line comments begin with //. Multi-line comments begin with /* and end with */. Javadoc comments, which the javadoc program uses to generate HTML documentation, use /**.

compile-time

The time at which a source file is compiled into a class file. Some errors are caught during compile-time.

compiler

The program distributed with J2SE called javac, which translates Java source code (in .java files) into Runnable bytecode (.class files).

constructor

A constructor is defined in a class file, and is used to create instances of the class. Constructors have the same name as their class, and are invoked using the new keyword. A Java class may define zero or more constructors. If no constructor is defined, the constructor for the superclass is automatically invoked. Additional constructors must specify a unique argument list, just like overloaded methods.

continue

A Java keyword used inside loops. Invoking it causes program execution to resume at the end of the current loop. As with break, if a label is used, execution resumes at the label.

D

deadlock

Deadlock is an unrecoverable state sometimes encountered in thread programming. It occurs under the following circumstances: thread A has a lock on object X, and is waiting for the lock to object Y; meanwhile, thread B has the lock on object Y, and is waiting for the lock on object X. In the event of a deadlock, program execution will simply suspend indefinitely, making it difficult to debug.

declaration

A statement that creates an identifier and associates attributes with it. Declaring a data item may or may not reserve memory space. A method declaration may or may not also provide the implementation.

default

A Java keyword used within a switch block to indicate the code block to execute if no case value matches the switch value.

do

A Java keyword used to declare a style of loop.

DOM

Document Object Model. An XML specification for representing data as an object tree.

Double

An object wrapper for the primitive double.

double

A Java keyword used to declare a 64-bit primitive of type double. Floating point numbers have ranges dependent upon the size of the mantissa and exponent fields. Double.MAX_VALUE and Double.MIN_VALUE indicate these for the IEEE double-precision data type. These represent the largest and smallest possible positive values.

double precision

A double precision variable is an IEEE, standard, floating point variable that is capable of representing numbers in the range 4.9 x 10-324 to 1.8 x 10308.

E

Encapsulation

A concept in object-oriented programming that dictates that the implementation of data and behavior within an object must be hidden, exposing only a public (client-facing) view of the interface, thereby enforcing a contract for acceptable client interaction with the object. Encapsulation of your code increases portability and reuse, and is a very good thing to do. Ways of encapsulating your code include making private variables, with public getX() and setX() methods.

error

A state from which a program cannot recover. java.lang.Error is a subclass of java.lang.Throwable, and is the parent class of all Java error classes. A runtime error can occur in the following circumstances: when an exception is not caught by any catch block (in which case the controlling thread invokes the uncaughtException method and terminates), when you try to cast a class into a type that it cannot be cast to (in which case a ClassCastException is thrown), or when you try to store an object in an array meant to hold different object types (in which case an Array StoreException is thrown). There are also LinkageErrors, which are thrown when a loading, linkage, preparation, verification, or initialization error occurs in the classloader.

exception

A programmatic state that subverts the normal flow of execution. Exceptions should be handled by anticipating the possibility of their occurrence, and surrounding the potentially problematic code with a try/catch block.

extends

A Java keyword to indicate that the current class is a subclass of the class it extends. By extending a class, a class inherits the functionality defined in its parent class (also called the superclass). This functionality can then be added to or, if the parent data item is not declared final, overridden.

F

field

A variable defined in a class, also called a member.

final

A Java keyword indicating a data item (class, field, or method) that cannot be overridden. A final class cannot be subclassed. For example, the following statement is illegal: `public class SuperString extends String {}`.

finally

A Java keyword indicating a block of statements that should execute whether or not the code within a corresponding `try` block threw an exception. A `finally` clause is not required, and is typically used to clean up resources (such as to close a connection to a database or socket, or to close a file opened in a corresponding `try` block).

Float

An object wrapper for the primitive `float`. This wrapper class defines a number of possibly useful constants, including `NaN` (Not a Number), `MAX_VALUE`, and `POSITIVE_INFINITY`.

float

A Java keyword used to declare a primitive 32-bit floating point number variable. Its numerical range is (1.4 * 10^-45) (Float.MIN_VALUE) through (3.4028235 * 10^38) (Float.MAX_VALUE).

for

A Java keyword used to declare a kind of loop that iterates until a specified condition is met.

G

Garbage Collection

Name for the process by which the Java Virtual Machine frees memory by reclaiming resources that are no longer referenced or accessible by the program. This feature of Java makes it easier to program in than other languages in which the programmer must explicitly free objects. The garbage collector executes on the heap, because that is what object data is stored.

generics

The name given to the collective functionality that allows a programming language to provide generic types and generic methods. In Java, generics are often compared to C++ templates. This comparison is understandable, as templates and generics aim to provide similar kinds of functionality. However, you should probably abandon this comparison readily, as the two are implemented very differently.

See also **[Generic Type]**
See also **[raw type]**
See also **[wildcard]**

Generic Type

Generic types include both generic classes and generic interfaces. These declare one or more variables, called type parameters, and are special because they are of unknown type until runtime. An example of a generic type is `ArrayList<E>`. The `<E>` indicates that you can create an instance of the ArrayList class by passing a specific type to it at runtime, like this: `ArrayList<String>`. This invocation means that this ArrayList will hold only objects of type String. The runtime will know that, and then you won't have to cast down from Object when you get the elements back out.

See also **[raw type]**
See also **[parameterized type]**

GUI

Graphical user interface. Pronounced "gooey." Components that allow the user to visually interact with a program. In Java, GUIs are created using (primarily) the Swing library.

H

heap

The heap is where Java objects are stored. Sometimes it is called the garbage collectible heap because this is the only area in which the garbage collector runs, ensuring that there is as much free memory as possible.

Hexadecimal

Base 16 numbering system in which 0-9 and A-F represent the numbers 0 through 15. Java hexadecimals are prefixed with 0x. Note that the letters used to represent a hexadecimal number are not case sensitive—that is, 0xAFC is identical to 0xafc.

I

identifier

A name used to represent a data item in a Java program such as a field, method, or class.

implements

A Java keyword used in the class declaration to indicate the name of an interface that this class provides an implementation for. A class can implement multiple interfaces; when this is done, separate interface names with a comma, as in this example: `public final class String extends Object implements CharSequence, Comparable, Serializable`.

import

Allows the programmer to reference individual class names in code without using the fully qualified class name (that is, without using the package name). Only classes in the java.lang package are imported automatically—and because of this, we can simply type `String` in our code, instead of `java.lang.String`. Note that as of Java 1.5, you can perform static imports, which means that you can import the static methods and fields of a class so that you don't have to prefix the class name when you invoke them.

Inheritance

In object-oriented programming, the concept that a class automatically receives data and functionality (variables and methods) from its parent class. In Java, all classes inherit from java.lang.Object. The parent class is also referred to as the superclass, in which case, the inheriting class is referred to as the subclass.

Instance

One particular object of a class—that is, the object referred to by this. To create an instance of an object, use the `new` keyword. For example, this code creates a new instance of the Integer class: `Integer x = new Integer(42);`.

instanceof

A Java keyword indicating the operator that determines if a reference is of a certain type. The operation returns true if the reference passed can legally be cast to the given type. For example, the statement `return "Hey!" instanceof Object;` will return true, because String extends Object. Note that it also works when the reference type is an interface: `return (new ArrayList() instanceof Collection)` also returns true.

int

A Java keyword used to declare a 32-bit signed integer primitive in the range of 231 to 231-1 (or -2,147,483,648 to 2,147,483,647).

interface

A Java keyword used to define a set of methods and constants without implementation. A class that implements the interface must provide an implementation for each method defined in the interface. An interface can be used as the reference type for a class that implements the interface. For example, ArrayList implements the methods of the Collection interface, so we can legally write this: `Collection myArrayList = new ArrayList();`. We can then use our myArrayList object anywhere that a Collection is called for.

Iterator

An interface that offers a way to allow a caller to access each element in a collection without exposing the collection itself to the caller. The benefit is one of increased encapsulation. Iterator is an improvement over the older Enumeration, in that they feature clearer, more precise method names, and offer a way to remove elements from the collection.

J

Java Virtual Machine

Part of the Java Runtime Environment, the JVM is the program that interprets and executes bytecode in compiled Java class files. The JVM is invoked with the java command and supplies programs for a context in which to run, providing resources such as memory management via garbage collection, network access, and security. The JVM acts as an abstraction of the hardware layer, enabling programs written in Java to run on any platform with a Java Virtual Machine. There are different JVM specifications, including one for smart cards and micro devices.

J2EE

Java 2, Enterprise Edition is a free download, separate from the Standard Edition, that offers APIs for database, advanced networking, server-side, and distributed functionality.

J2ME

Java 2 Micro Edition is required for writing small footprint programs runnable micro devices including smart cards, set-top boxes, and handhelds.

J2SE

Java 2 Standard Edition provides the basic development environment for creating portable, secure, network-ready software applications. When declaring an interface, you may only use the public and abstract modifiers. That is, you cannot declare a static method in an interface.

JAR

A Java ARchive is a file format used to aggregate and compress a number of files into one. Because the jar utility, which is included with the standard SDK, uses the same compression algorithm as .zip files, JARs can be manipulated using Zip utilities.

JDBC

Java Database Connectivity is an API that allows programmers to access tabular data stored in relational and object databases, text files, and spreadsheets. JDBC is included with the J2SE.

JDK

Java Development Kit also referred to as SDK (Software Development Kit).

JFC

Java Foundation Classes. Set of Java class libraries, included with J2SE, that support the building of rich Graphical User Interface applications. The JFC includes AWT (Abstract Window Toolkit), Drag and Drop APIs, Java 2D, Swing, Accessibility APIs, and Internationalization support.

JIT Compiler

A Just In Time compiler takes the bytecodes of a .class file at runtime and compiles them into code native to the machine on which it's running. A JIT compiler therefore replaces the functionality of the Java bytecode interpreter. This happens on-the-fly, method-by-method (hence, "just in time").

JNDI

Java Naming and Directory Interface is a set of APIs that support interaction with directory services such as LDAP.

JNI

Java Native Interface allows code that runs in a JVM to interact with code written in other languages, such as C, C++, and assembly. JNI, which is part of the J2SE, is used by programmers when they cannot or do not wish to write their entire application in Java. This situation would occur when you already have some native code written that you'd like to reuse, or if Java does not support the operation you need to perform.

Java RMI

Java Remote Method Invocation is for use in distributed environments, and allows an object in one virtual machine to call methods on an object in another virtual machine.

Java Runtime Environment

The JRE is a subset of the SDK. It includes the Java HotSpot runtime, client compiler, the Java Plugin, and nearly twenty standard libraries (the core classes). It does not include the compiler, the debugger, or other such tools. The JRE is the minimum runtime required for executing Java applications; if your machine ain't got a JRE, it won't run your Java program. That's why JREs are freely distributable, because developers often need to package one along with their applications. Note that the SDK is not freely distributed by developers.

K

keyword

A word reserved by the Java programming language for its exclusive use, which is therefore not available for developer use in naming variables or methods. For example, `String for = "x";` is illegal because `for` is a keyword. Note that two terms, `const` and `goto`, are reserved but not used.

L

Local inner class

A local inner class is a class defined within the scope of a method. Local inner classes are often also anonymous classes. An important caveat regarding their use: Local inner classes cannot see variables defined in their enclosing method *unless* those variables are marked final.

Local variable

A variable declared within a method. Local variables are sometimes (rarely) referred to as automatic variables. Local variables are distinct from class variables in that they must be initialized before you can use them. Local variables' visibility is only the life of the method—when the method in which they're defined returns, local variables vanish.

Long

Object wrapper class for long primitives.

long

Java keyword used to declare a primitive variable of type long, which uses 64 bits to represent integral values ranging from -263 to 263-1 (or -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807). Yes, that's over 9 *quintillion*.

M

members

The data elements that make up a class: inner classes, methods, and variables.

method

An element defined inside a class that performs a function, can accept arguments, and return a single result. A method body consists of the block of statements inside curly braces.

modifier

A keyword placed in the definition of a class, method, or variable that changes how the element acts. There are visibility modifiers (typically called access modifiers), such as public, protected, and private, and other modifiers, such as final or native.

modulus

The operator that returns the remainder of an integer division operation, represented in Java as %. For example, 5 % 2 returns 1.

multiple inheritance

Refers to the capability of a programming language to support a class extending more than one class. This is possible in C++, but not in Java. This decision was made on purpose in order to help code clarity and encourage good design.

multithreaded

A program that executed multiple threads concurrently.

N

NaN

Not a Number. A floating point constant representing the result of attempting to perform an erroneous mathematical operation (such as trying to divide by zero). Referencing the static variable Double.NaN prints "NaN", and calling the non-static method isNaN() on a Float or Double object returns a boolean. For example: return new Float(3.14).isNaN() returns false.

null

Represents the absence of a value, using the ASCII literal null.

O

Object

An object is an instance of a particular class, making it the fundamental unit of an object-oriented application. Objects typically store data in variables, and provide methods with which to operate on that data.

OOP

Object-oriented programming. Important concepts in OOP include Encapsulation, Inheritance, and Polymorphism, all of which Java supports.

overloading

A method is overloaded when the same class defines multiple methods with the same name but different argument lists. Constructors may also be overloaded.

overriding

A method is overridden when a subclass defines a method with the same name, return type, and parameter list as a method in a superclass.

P

package

A Java keyword that is used to indicate the package name to which this class belongs. Conceptually, a package is a group of classes with a logically similar purpose.

parameterized type

A generic type, with actual arguments supplied for its type variables. For example, ArrayList<E> is a generic type. You can invoke it as ArrayList<Product> and then that is called the parameterized type.

primitive

A Java primitive is a simple type that evaluates to the single value stored in that variable (as opposed to a reference type, whose value is its memory address). Primitives cannot have methods or hold any other data than their single value. You cannot cast a primitive type to an object reference, or vice versa.

private

A Java keyword used as an access modifier to indicate that this member is not visible to any other class but the one in which it is defined.

process

A space in a virtual addressing system containing one or more threads of execution.

protected

A Java keyword used as an access modifier to indicate that this member is visible only to the class in which it is defined, any subclass of that class, or any class in the same package.

public

A Java keyword used as an access modifier to indicate that this member can be accessed by any other class.

R

raw type

A parameterized type invoked without its parameter. ArrayList<E> is a parameterized type. You can either supply a parameter for E, or you can forget about it, in which case your invocation is ArrayList aList = new ArrayList();, and aList is then referred to as a raw type.

reference

A data element (object or interface) that serves as a pointer to an address in memory where the object is stored. A reference variable is a name used to reference a particular instance of a Java class.

return

A Java keyword used to indicate that a method has completed execution, and should give control back to the caller. return is optionally followed by a value or expression as required by the method definition. The following are all legal: return;, return "choppers";, return (x < (7 + y));, and return new Double(42);.

runtime

though "at runtime" is sometimes used to mean "during execution of the program."

See also **[Java Runtime Environment]**

S

scope

A member's scope dictates its visibility. An automatic (local) variable is one defined inside a method body; its scope is the method itself, and it is not visible to code outside the defining method.

shift operator

The shift operators move the bits of an integer number to the right or the left, which results in another number.

short

A Java keyword used to declare a 16-bit signed primitive integer type in the range of -215 to 215-1 (or -32,768 to 32,767).

signature

A method's name and the type and order of its parameter list.

stack

The place where Java stores references to objects. The object data itself is stored on the heap. The stack is set to an initial fixed size, whereas the heap can grow to consume all available system memory. When a JVM thread is created (when a program starts), a separate, private stack is created for that thread. The stack stores frames, which contain the data.

stack trace

A list of called methods, in descending order in which they were called. The list represents the threads and monitors in the Java Virtual Machine, typically that led to an exceptional state. If the JVM experiences an internal error (say, because an exception is thrown), it will signal itself to print out the stack trace. Useful in debugging, the method call on the top of the stack trace should be the last method called when the program encountered an error.

static

A Java keyword used to define a variable, method, or standalone block of code as being executable without an instance of the class. Because static variables are called on a class, there is only one copy of a static variable, regardless of how many instances of the class there may be. Static methods may only operate on static variables. Notice that you can just write inside a class `static{...}`, which declares a block of code (not a method) that executes when the class loads (it would be polite of us to notice too that this technique is sometimes frowned upon, as it is obdurately defies object orientation). It is useful in invoking libraries used by native method calls.

static method

A method called directly on a class, not an instance of the class. Static methods do not have knowledge of instance variables. For example, the following statement is a static method declaration: `public static void main(String[] args)`. It makes sense that the famous `main` method would be static—how could you start your application if you had to make an instance of some object first in order to call the `main` method? You would be making objects, in which case you would have already started your application. The following example shows using a class name to call one of its static methods without an instance of it: `System.exit(0);`. Methods declared in an interface may *not* be declared static.

static variable

A variable available directly from the class itself, not an instance of the class. Also known as class variables, static variables will have the same value across every instance of the class.

stream

Data read in or written out as a sequence of bytes or characters.

String literal

A sequence of characters contained within double quotes.

subclass

A class that extends another class, either directly (by using the keyword `extends` in its class declaration) or indirectly. All Java classes are subclasses of java.lang.Object. Sometimes subclasses are called child classes.

super

Java keyword used to call the constructor with the matching parameter list of a parent class.

superclass

A class from which another class is derived. Also called the parent class. java.lang.Object is the superclass of all Java classes.

Swing

A set of lightweight GUI components that run uniformly on any platform that supports the Java Virtual Machine.

switch

Java keyword used to evaluate a variable of integral primitive type (either an `int` or a type that can be implicitly cast to an `int`, such as `byte`, `char`, `short`, or `int`) in order to match its value against values declared in `case` statements. If the `switch` value does not match any `case`, a default statement is executed, if implemented.

synchronized

Java keyword indicating that the method or code block that it contains is guaranteed to be executed by at most one thread at a time. Notice that you can write a perfectly legal (though useless as written here) block of code `synchronized (new Object()){}`, which demonstrates that you can synchronize access to an object without a method.

T

this

Java keyword referring to the current object instance.

Thread

The java.lang.Thread class defines the behavior of a single thread of control inside the JVM. It can be subclassed to create threads within your application. The other way to create a thread is to write a class that implements the java.lang.Runnable interface.

thread

A thread represents the basic unit of execution in an application. One process may have multiple threads running at once, with each performing a different function. When a thread is executed, local variables are stored in a separate memory space, so that different instances do not confuse or overwrite each other's values.

throw

Java keyword used when the programmer means to generate a new instance of an exception type for handling by the caller. For example: `throw new GarageException("Houston, we have a problem:");`.

throws

Java keyword used in method declarations to indicate that the method will pass the specified exceptions up to the caller to deal with.

transient

Java keyword indicating that a field should not be considered part of the persistent state of an object, and should therefore not be serialized in an ObjectStream. Does not apply to any element but variables.

try

Java keyword specifying a block of code as one in which an exception might occur, and which consequently must be accompanied by one or more `catch` blocks to deal with the exception.

U

unary

A unary operator is one that affects only a single operand. For example, + and - are unary operators used to signal the positive or negative value of an integer.

Unicode

Unicode defines a unique number for every character in every language, on every platform. It is the 16-bit character set that serves as the official implementation of ISO10646, and is capable of representing characters from many languages. Although the standard Latin character set used in English requires only 7 bits to represent, more complex characters, such as those written in Arabic, Japanese, Hebrew, Georgian, Greek, and Korean require 16 bits. Unicode is used not only in Java, but also in XML, LDAP, CORBA, and others.

V

Virtual Machine

The Java Virtual Machine serves as the main interpreter for Java bytecodes, and consists of a bytecode instruction set, a set of registers, a stack, a heap, and a method storage area.

Visibility

The level of access that methods and instance variables expose to other classes and packages. You define visibility using the modifiers public, protected, and private (note that there is also default visibility, though you don't write default explicitly).

void

Java keyword used to indicate in a method declaration that the method does not return any value. Methods returning void return implicitly when they complete the final statement. Note that this does *not* mean that use of the return keyword is mutually exclusive with use of the void return type—that is, the following is perfectly legal (but probably useless): public void dumbMethod(){ return; }.

volatile

A modifier indicating that a variable can be accessed and modified asynchronously by concurrent threads. For example: volatile int myNumber;.

W

widening conversion

What happens when a value of one type is converted to a wider type—that is, a type with a wider range of possible values. Converting from a short to an int is a widening conversion; converting from a char to a byte is not. The Java runtime knows that your variable couldn't suffer any loss of data by putting its value in a container bigger than the one it already has. For this reason, widening conversions happen automatically. You can lose data when performing a widening conversion that is done automatically. Numerical range is the important thing. Example (float's numerical range encompasses long's, but long (64-bits) can hold more digits than float (32-bits)):

```
public class ConversionTest {

public static void main(String[] args) {

   float a;

   long b = 4203020102023324L;

   a = b;

   System.out.println("float: " + a);

   System.out.println("long: " + b);

   }}
```

Output:

```
float: 4.2030201E15

long: 4203020102023324
```

wildcard

Used in regular expressions to mean any character. In generics, the wildcard is represented as ?. This is used in a generic method declaration to indicate an unknown type. The wildcard can be type-bounded by either upper or lower limits.

See also **[bounds]**

wrapper class

The primitive wrapper classes in the java.lang package correspond to each of the eight primitive variable types: the Boolean primitive wrapper class is java.lang.Boolean; the char wrapper class is Character, and so forth. The purpose of the primitive wrappers is to provide utility methods (such as Double.parseDouble()), constants (such as Boolean.TRUE and Float.POSITIVE_INFINITY), and object encapsulation for primitives.

PREV < Day Day Up > NEXT

Chapter 1. WHAT IS THE JAVA GARAGE

[View full size image]

```
login as: eben
Sent username "eben"
eben@www.thegarage.net's password:
[eben@dev eben]$ what is the garage?...
```

Programming on the Internet, in Internet time, is a collaboration. There are a million sites out there to find out how to write a little snippet of code. Go to Google and search for C# Custom Cursor, and up come 7,000 web sites to tell you how to change the appearance of the mouse pointer in C# for .NET. So you don't need me, do you smart guy?

It works once you know what you're doing. It'd be a rough ride learning that way.

You came here. You wanted something. You felt it your whole life. That tech books could be something more.

The garage is a place to go to be alone. To be with your friends. It's an accessory. It's a style, a way of life, it's a friend, it's a companion, it's an IRC server, it's what a book would be if a book could be a wiki with exactly what you need and nothing you don't. And no banner ads.

Says the Oracle: "How could our jobs be so fantastic, so filled with art and innovation and dripping with collaboration and speed and the thrill of making it work—and yet our books be so dull and dusty and cardboard and boring and in a format that's essentially unchanged for more than 35 years?"

You came here to learn what you need to know about Java until everything turns green.

So let the garage be green.

Our work and our play are so interweaved they are indistinguishable. This is how we think now. There is no such thing as the weekend. Luckily for us, our work is fun. It is an absolute delight. IT is like Wonka's factory for smart people. Can our books be that? Force the boring writers to listen to Yoko Ono nonstop until they relent in their boringness.

And the garage god said, "Make It So."

In the garage you do your projects. On your time.

So our garage has a toolkit, which means you got a load of code to work with, and it's well documented so you understand why it works the way it does. Cuz that helps you remember it next time. And there are FAQs to look up quick deals that you need to use in your daily job. And a glossary on steriods that doesn't just point to a pointer of a pointer of a pointer, but tells you the whole definition, and gives you a context.

Working in IT happens with windows open and processes running and the ftp here and the zone file there and the Photoshop and your IDE all open and going and getting it done. Show me the Java programmer who doesn't need to know anything about html and networks and naming and sql and xml and administration and uml. A book about Java isn't written for any human; it's written to be Da Word. I reject the word. That's not how I work, or how anyone I've ever met in IT works. We are situated in a context with a lot of technologies that work together. Perhaps our books can be like this?

And thus the Architect wrote it.

Garages are all different. I live in the hot and hollow, barenaked desert of the American southwest. Sometimes scorpions find their way into my garage. And rattlers sometimes, and thick brown spiders. Another garage will be different.

Because it is different, each garage has a blog, where the writer goes to rant. This keeps everybody honestpuck. It makes sure there is a human voice. Selling something isn't allowed. We're just going to talk for a while. And at the end of the conversation, you'll see green.

Cuz in a garage, you bang it around until it works.

So each garage has real, working code, and instructions like recipes for making usable, neatokeen applications that you can incorporate into your own projects. Cuz that's the point. Too bad how lotsa programmer book guys eject from the cockpit at the very last second, leaving the real work as "an exercise for the reader." In the garage, we don't get much of that kind of "exercise." Maybe just lifting a beer once or twice. When Willy Wonka is your role model, you should be able to eat everything.

Lick the pages. They taste like schnozberrys….

The Garage is where you go on Saturday. You hack it out when you get a second. So none of this long claptrap with 5,000 pages, and I'm breaking my wrist carrying that sheet to work, and all I need is for my wrist to get jiggy with the exchange server come 5 p.m., if you know what I'm sayin'. Short, focused, recipe deals that give you the how and the why. Uh, I guess we won't be doing nothing like, "The All-Time History of Programming blah blah and then dude invented fire and then blah blah and then came Linux." How about instead, "Retrieving User Input from the Console." Then, boom. How to do it. Why it works. How to integrate it. Done.

The garage is a place. I don't know where you're from, but my garage don't really go in a straight line. It's more like I hang out there til the game comes on.

Now. This is our Java Garage.

The keymaker says, "That door will take you home…".

PREV < Day Day Up > NEXT

Chapter 2. JAVA BUZZ

D00dle: yt?

Zoomz: y

D00dle: I was thinking about this cuz I was walking by the Taco Tico and I stand there getting tortured by Avril Lavigne and I'm all, All I wants this stoopid burrito and you can't give it a me without I gotta have this claptrap making my ears bleed.

D00dle: stop whining dude

Zoomz: no so I was thinking just like this: if i was in a band called the Yeah-Yeahs all my songs would go "Yeah Yeah Yeah"

Zoomz: dude I would SO buy your album.

D00dle: thanks!

D00dle: you do java rite?

Zoomz: ?

D00dle: what's the story?

D00dle: we got a new cto loves unix esp freebsd

Zoomz: does he hate m$?

D00dle: duh

Zoomz: thats like so late nineties.

Zoomz: is he cute

D00dle: as if. how do you know its a he

Zoomz: does that mean you have to port all your apps

D00dle: yeah18 mos work on the milk lady's project dead.

Zoomz: so you're learning java

D00dle: y

Zoomz: so read a book knob

D00dle: i hate books punk just tell me why thiss guys all hopped up about java

Zoomz: dude I got like 10 words for you:

Simple

OO

Network-ready

Robust

Secure

Architecture Neutral

Portable

Interpreted

Fast

Multithreaded

Dynamic

D00dl3: LOL! that's all marketing dude. i can't believe you would actually write that to me. Do you hate me?

Zoomz: that is just true.

D00dl3: every company says their thing is secure my grandma is secure.

Zoomz: did you ever hear of a virus written in java?

D00dl3: no.

Zoomz: Did you ever hear of a virus exploiting a java application?

D00dl3: no.

Zoomz: well then. would you be in this port trouble today if you had written in java originally?

D00dl3: no. but what about fast, robust, simple, and dynamic? that's where you really sink low bro.

Zoomz: you're right—that's all marketing hoo-hoo. not that it isn't true, but like so is C++ C# VB C Ada Eiffel Lisp Delphi

D00dl3: You don't have to convince me. I can learn it or lose my job to some 14 year old whippersnapper. How did i get to feel so old and i'm not even 25? I need to learn it and i want to learn it because i know it is an important language now so it doesnt matter. So just how do i do that so i can get back to Never Winter Nights?

Zoomz: read java garage.

D00dl3: i hate books.

Zoomz: its not a book: its garage...

Chapter 3. JAVA EDITIONS AND PLATFORMS

Sort out the different Java editions without losing your mind.

Dear Sirs and Madams,

I just want to programming with Java. Yet, I cannot start. How can I do this?

Thank you,

xaolin

do you have the JDK?? go to java.sun.com and download the JDK.

^_^ zilly

Dear Mister Zilly,

Everything I have seen assumes that you already know this. I do not know this. Please: what is JDK?

Xaolin

am not *mister* zilly i am devo d-e-v-o. now. Java Devlopment Kit. That is the compiler, to compile source code into executable code, and a runtime, to run it. And other stuff.

^_^ zilly

Dear Zilly,

I go to the download my thought is: what is the difference between each of different editions?

Which do I get? I prefer to have only the best.

Many courtesies,

Xaolin

Hi Xaolin,

There isnt really a best one, the different editions arent like more powerful or something they dont have superpowers like but that would be cool. Here is the difference between those three J2SE, J2EE and J2ME:

J2SE: this is Java 2 Standard Edition. Download this one. It is the one you want to get started, as it is the most frequently used, and includes a compiler and a runtime so you can write Java applications. Heres what you can do with it:

- Use Swing to create full-blown Graphical User Interface applications.
- Write applets, which are small programs that run inside a Web browser.
- Connect to remote computers and write Client/Server applications with sockets.
- Work with the local file system.
- Parse text with regular expressions.
- Read, write, and transform XML.
- Execute code in a remote Java Virtual Machine using Remote Method Invocation.
- and oh so much more.

The other ones are J2ME and J2EE.

J2ME is Java 2 Micro Edition. This edition is geared toward applications embedded in consumer products such as cell phones, PDAs, set top boxes, smart cards, and car navigation systems. One thing that distinguishes J2ME from the other editions is that it is the only one that is subdivided into profiles and configurations. A profile defines libraries for different platforms and the JVM requirements for supporting a particular micro platform (for example, there is one profile for PDAs and another for wireless devices). A configuration is composed of a virtual machine and a small set of libraries that provide base functionality to devices. Currently there are two configurations: Connected Limited Device Configuration and Connected Device Configuration. A runtime is included, which takes a very small footprint. The smart card runtime in the Card Virtual Machine, for example, consumes only 128K of memory.

J2EE this is Java 2 Enterprise Edition. It adds significant functionality to the Standard Edition. It includes support for the following technologies:

- JavaServer Pages and servlets allow developers to create dynamic server-side Java applications.
- Enterprise Java Beans, which are components that support transactions, security
- JDBC (Java Database Connectivity) allows developers to invoke Structured Query Language operations from Java code to interact with databases.
- JavaMail lets you work with email messages
- Java Message Service (JMS) allows the distributed, asynchronous posting and retrieval of message objects.
- JNDI is the Java Naming and Directory Interface, which lets you perform typical directory operations. But JNDI is not tied to any particular implementation, so it can be used to interact with established services such as LDAP, DNS, and NIS.
- Java Authentication and Authorization Service (JAAS), which is used to determine if users and groups are allowed to enter a system (authentication), and what specific tasks they are allowed to perform once allowed in (authorization).
- JAX-RPC: the Java API for XML-based Remote Procedure Call. This allows you to develop SOAP-based Web Services clients and endpoints.

j2ee apps are typically server-based applications that use a combination of these technologies to get their jobs done. But all you need for a long time is the J2SE.

^_^ zilly

Dear Zilly,

Thank you for this kind response. I must warn you that this j2ee is not acceptable. I must have connect to database for my programming, yet regretfully I am not for using j2ee. What is it to do?

Xaolin

DUDE

you can use the services that j2ee makes available even if you are just doing a standard j2SE app. later, we can talk about JDBC and how to do that. All of them are free. So you dont have to pay to get the language or to write programs in the language which is sweeeeeeeet.

^_^ zilly

Dear Zilly,

I am not pleased to bother you. I confuse again: You refer to Java 2. And yet, these versions are not Java 2, but Java 5.0. Is there somewhere else I am supposed to look?

In Modesty,

Xaolin

Xaolin,

That is a very good question. You are in the right place. There is no such thing as Java 2.0 or 3.0 or 4.0. The most recent version is 1.5 following 1.4 and 1.3, etc. At the 2004 JavaOne defeloper conference, 1.5 was renamed to Java 5.0. This is a little like how Sun came up with Java 2. Java 1.2 contained significant changes to how graphical user interface elements are rendered in desktop applications with the addition of the Swing libraries (or Java Foundation Classes). Three days after they released this version as 1.2, Sun thought that the change was so significant that they should start referring to it as Java 2. But 1.2 already had a foothold in the market, so both names stuck. As a consequence, Java 2 still refers to subsequent releases, such as 1.3, 1.4, and 1.5. So you will see a lot of things referrring to Java 1.5 or Java 5.0—hey, refer to the same version. Ack.

^_^ z

Dear Zilly,

Is there a difference between SDK and JDK? Which should I get? You mentioned JDK in an earlier post.

Thank you and blessings,

Xaolin

Xaolin,

Dont worry about that other little sleight of hand: the two terms refer to the same thing. The Java Development Kit (JDK) was simply renamed the Software Development Kit (SDK) in November of 1999. Maybe it sounds more universal that way.

^_^ zilly

Dear Zilly,

I apologize, but I go to download it and again a problem. There are two links next to every Operating System: SDK and JRE. What is this now? Which one should I get? I thought that Java was cross-platform—yet there are one SDK and JRE for each of these OS: Windows, Linux, Solaris SPARC processor and Solaris x86 processor. Which one do I get? Why isn't Java cross platform now?

Cheerfully but with regrets,

Xaolin

Hey,

First, the cross-platform matter. Java applications that you write are cross-platform. Sun likes to say Write Once, Run Anywhere. That means that you can execute the same Java code on any machine that has a Java runtime available. Which means, for now, Linux, Solaris and Windows.

Your Java applications use the runtime to execute in—but the runtime itself, which is written in native code, is dependent on a particular platforms architecture. Hence the different options.

There are, however, a number of ports of the Java SDK and/or JRE that make Java available on the following systems, though these ports are not produced by Sun:

- Mac OS
- Tru64 Unix
- SCO
- AIX
- HP-UX
- NetWare
- IRIX
- NonStop
- OS/2, OS/390, OS/400
- VxWorks
- NetBSD
- FreeBSD
- Reliant Unix

These different systems are not tested or maintained by Sun. Also check out http://www.blackdown.org/java-linux/ports.html for more information on current port status (Blackdown was the first team to port Java to Linux).

The FreeBSD Unix-based operating system has just recently made an arrangement with Sun to distribute FreeBSD binaries for the JRE and SDK. The FreeBSD Foundation refers to their JRE as Latte Diablo and the SDK is called Caffe Diablo. So you can develop and execute Java programs on FreeBSD but, for now, you have to get it from them. You can do so at: http://www.freebsdfoundation.org/downloads/java.shtml.

The Linux distros include:

- Red Hat7.3 and later
- SuSE 8.0 and later
- TurboLinux 7.0
- SLEC 8

Most Java applications should work fine on Debian, but check the excellent FAQ at http://www.debian.org/doc/manuals/debian-java-faq/ for details.

Bye xaolin i have to go my mom is taking me to my trumpet lesson now :P

^_^ zilly

Chapter 4. COMPILING AND RUNNING JAVA APPLICATIONS

Installing the SDK

First you have to get it. Visit http://java.sun.com/j2se/1.5.0/download.jsp.

Accept the license agreement and download the SDK. Make sure that as you follow the instructions below, you change the file names as necessary for the exact distribution you get.

On Windows

Just run the executable installer.

Open a command prompt. If you can type "java -version" and see something meaningful, you're good to go.

If not, read on.

On Linux

Get the RPM binary for Linux. You have to give the j2sdk* file permission to execute.

```
$ chmod a+x j2sdk-1_5_0-linux-i586-rpm.bin
```

Now execute the file.

```
$ ./j2sdk-1_5_0-linux-i586-rpm.bin
```

Agree to the license.

Change to super user:

```
$su root

Password: ******
```

To install it if a previous version of Java is not already installed, type

```
$rpm -iv j2sdk-1_5_0-fcs-linux-i586-rpm
```

If Java is already installed and you are updating to the new version, type

```
rpm -Uv j2sdk-1_5_0-fcs-linux-i586-rpm
```

Ensure that everything went all right by typing

```
/usr/java/j2sdk1.5.0/bin/java -version"
```

This executes the java program, passing it the "-version" flag. If all has gone well, you will see version information indicating that you are running Java 5.0 or Java 1.5.0.

Now you will need to set your path and classpath.

FRIDGE

You might need to set your path and your classpath before you are able to compile, especially if you aren't writing your code in an IDE. See "Setting the Classpath" in this chapter for details on how to do that.

This section covers how to compile Java source code using the tools distributed with the Sun J2SE 5.0 (a.k.a. 1.5).

Compiling Source Code

To compile Java programs after you have installed the SDK, open a console. In Windows, go to Start > Run and type cmd to get a command prompt.

Note that if you have multiple JDKs on your system, you may need to supply the –source 1.5 flag to the javac command to ensure that your code is compiled correctly.

```
C:\garage\src>javac -source 1.5

    net\javagarage\demo\MyProgram.java
```

This compiles the Java source code into a class file in the same directory.

If all goes well during compilation, you get a new command prompt, and nothing else. You are now ready to run your program using the java command.

Compiling into a Directory Other Than Your Source Directory

It is common practice, and it's a good thing, to keep your source files in one directory and your class files in another. Typically, when you start a project you do something like this:

```
/application root dir

/application root dir/src

/application root dir/classes
```

You create two directories under the root of your application. Put all of your source files under the src directory and compile such that your classes go into the classes directory. Then you can use the JAR archiving tool to package just your classes and deploy those when your application is done.

By default, the javac tool compiles your classes into the same directory that the source file is in. So, you need to pass a command to the compiler to handle this. If you use an IDE such as Eclipse, you can almost always set the compiler output path in a project properties window.

To compile to a directory other than the one your source is in, use the -d flag, followed by the name of the directory you want the .class files to end up in.

```
C:\j15\net\javagarage\demo\swing\

    layouts>javac -source 1.5 -d

C:\j15\classes GridLayoutExample.java
```

The preceding example takes the file GridLayoutExample.java and compiles it into a class called GridLayoutExample.class under the C:\j15\classes directory with its packages intact. In other words, if the GridLayoutExample class is in a package called net.javagarage.swing.demo, you can find the class file in C:\j15\classes\net\javagarage\swing\demo.

Running Programs

After you have compiled your source files into bytecode using the javac command, you are ready to execute them. To do so, you need to follow these directions.

Navigate to the directory that stores your top-level package. For the examples in this book, that is a directory called C:\garage\classes. If you are using an IDE, your classes directory might be different. For example, an Eclipse project might store files at C:\eclipse\workspace\garage\classes. In any case, it is the directory that contains your packages, and where you specify the compiler to output code.

Run the java command, passing it the fully qualified name of your class (including all of the package names).

Make sure to use the . separator for packages—not your system file separator (/ on Linux and \ on Windows). This is a common mistake, because you have to supply the file separator when compiling.

For example, here is how to run the CommonFileTasks.class program:

```
C:\garage\classes>java net.javagarage.demo.MyProgram
```

Make sure that you leave off the .class extension. That can be hard to remember because you have to include the .java extension when compiling.

Make sure that the class you invoke on the command line is the one that contains your public static void main(String...args) method. This is the method that is called when the JVM starts your program, and it will tell you if it cannot be found.

If you see this error,

```
Exception in thread "main"

    java.lang.NoClassDefFoundError:

MyProgram
```

it means that the java runtime program cannot find your bytecode file, MyProgram.class.

The Java runtime will look for your bytecode file in the present working directory. So if your .class file is in C:\garage\classes, change to that directory using the cd <dirname> command on Windows and Linux. This is often not convenient, however. To remedy the problem, you can set the classpath system variable.

Setting the Classpath

If java is still unable to find your program, you might have to change your CLASSPATH variable. The CLASSPATH variable is where the java program looks to find classes to include when it executes.

In your CLASSPATH setting, make sure that . is included, to indicate the present working directory, as well as any custom directory you make to store class files, such as C:\garage\classes.

In Windows 2000, set the CLASSPATH like this:

1. Click Start > Settings > Control Panel > System > Advanced Tab > Environment Variables.
2. Find the one called CLASSPATH (or make it if it doesn't exist) and add the preceding directories to it.
3. Create this variable as a system variable if it does not exist. Also, make sure that the JAVA_HOME location correctly points to the top-level directory of your Java installation. This is probably C:\Program Files\Java\J2SDK1.5.0 on Windows.

Setting the Classpath at Compile Time

You can set the classpath at compile time. This is useful if you have a quick class you want to try out, and it relies on a library or other class that is not currently on your classpath. You do so using the -cp flag. You can also use -classpath if you are exceptionally fond of typing.

Try it like this:

```
javac -cp C:\\garage\classes SomeClass.java
```

Note that you can combine flags as well:

```
javac -source 1.5 -cp

    /user/eben/garage/SomeClass.java
```

Setting the Path

Setting the Path variable allows you to run the Java SDK executable programs, such as java and javac, from any directory on your system. It is a very good idea to set this variable, for convenience. If you do not set the Path variable, you need to specify the location of the javac compiler and java runtime every time you try to compile and run a program. Here's how to set it.

On Windows

In Windows 2000, set the Path like this: click Start > Settings > Control Panel > System > Advanced Tab > Environment Variables. Create a variable called path and set its value to the location of the <J2SDK-install-location>\bin directory. For example, it might be this:

```
C:\Program Files\Java\J2SDK1.5.0\bin
```

The Path can be a series of directories separated by semi-colons (;). Windows looks for programs in the Path directories from left to right.

On Linux

Linux uses a : to separate path variables. You can set the path on Linux by editing your /etc/profile file (su root first) to the location of your Java installation. Your installation may already have a JDK on it (RedHat and many others come with the Java SDK). So just type java at a terminal and see if you get usage information. If you do, you can use it. But this is likely to be a 1.4 JDK for some time.

On Mac OSX

Mac OS version 10.3 comes pre-installed with a Java SDK, which at this writing is version 1.4.1_03. While many of the examples in this book will work with that version, some important syntax has been added, and you probably will want to upgrade. To do so, just go to Finder and getting a terminal window—then you can follow the Linux instructions since Mac OSX is a mix of BSD distros under the hood.

There should be only one bin directory for a Java SDK in the path at a time. Any directory specified subsequent to the first is ignored. So if there is already a PATH variable set (say, because of an IDE or previous version), you can update it to j2sdk1.5.0\bin.

You can verify that this is working correctly by typing the java command at a prompt like this:

```
C:\garage>java -version
```

It should output version information similar to the following:

```
java version "1.5.0"

Java(TM) 2 Runtime Environment, Standard Edition

    (build 1.5.0)

Java HotSpot(TM) Client VM (build 1.5.0-32c,

    mixed mode)
```

You should now be able to compile Java source code and execute Java programs from any directory in your system. Note that if you use an IDE, you might need to make sure that it is pointing to your 1.5 JDK, and not a separate JDK that it ships with, which could be a different version that's incompatible with some examples in this book. For more information on the different tools that come with the SDK, see the SDK tools section at the back of the garage.

PREV < Day Day Up > NEXT

Chapter 5. WHERE TO WRITE CODE

DO OR DIE:

- Get some objectives
- Write them here
- Write about those objectives in the space provided below

This whole part is about where to write code. You can write your code in a plain text editor like vi or emacs or Notepad, or you can use a program that is specially suited to the task.

Integrated Development Environments

To make a Java program, you write Java source code, which gets compiled into class files, which are composed of bytecodes, which are then executed by the Java Runtime Environment when you run your program. That's a number of steps. The more complex your program, the more external details are involved in completing the system. It can be difficult to maintain two or three or four hundred class files for each project you're working on. It gets tiring repeating lengthy classpath arguments when testing your code. Perhaps you are working with other developers and would like to coordinate your efforts by using a common code repository. Or perhaps, being new to the language, you would like to have an environment in which to write code that gives you hints about what you are doing.

All of the above reasons are good reasons to take a little time, and perhaps money, to get a good Integrated Development Environment.

IDEs are very good at organizing projects. But unless you take a non-trivial amount of time learning the IDE itself, you won't find much more benefit to using an IDE than using some text editor and a shell script or Ant build. That being said, there are a number of IDEs that you can get for free or buy.

Friend, here's a list of some good, free Java IDEs:

- **Eclipse**. Available from http://www.eclipse.org. This is the one I use at work. The performance is improving, and this one will be really cool once it is integrated with IBM Rational tools in the form of XDE. There are a lot of plugins now available for Eclipse; you can write EJBs and integrate with JBoss, Tomcat, JUnit, Ant, and many other tools with ease. Its extensibility makes it very attractive. A notable drag about using it is its lean XML support.
- **Sun ONE Studio 5**. Available from http://wwws.sun.com/software/sundev/jde/. Sun bought NetBeans, and this is their Java IDE. This one has a free version and another version that is a couple thousand dollars. A new Sun IDE is stepping up to the plate in the form of Java Studio Creator, which looks and acts a little more like the Microsoft IDE Visual Studio. But for now, that IDE is meant to support JavaServerFaces programming for Web stuff, which we aren't discussing in this book. But it is a groovy tool, and you can read more about it here: http://wwws.sun.com/software/products/jscreator/.
- **Jext**. Available from http://www.jext.org/. This one is really more of a code editor, but it supports a lot of languages, and I like it. This one will soon become part of the Debian Linux distro.
- **Jcreator**. Available from http://www.jcreator.com/. Comes in Lite and Pro Editions.
- **BlueJ**. Available from http://www.bluej.org/. Good, simple tool for learning with visual aids.

Some good Java IDEs that aren't free:

- **IntelliJ**. Available from http://www.intellij.com. This is what many developers used at Macromedia to write the JRun 4 JSP, servlet, and EJB app server. Perhaps with less glamor, I used IntelliJ at my last job. I loved it: it works great, is integrated with Ant and CVS, and provides easy EJBs and plugin facilities. This is not integrated with any deployment environment, which could be good or bad depending on where you run stuff. If you have Apache Web Server and Tomcat and JBoss for EJBs, you're just fine. If you have Borland App Server, you're just fine too. But you don't get tools specific to that environment.
- **Jbuilder**. Available in Personal, Developer, and Enterprise editions from http://www.borland.com/jbuilder/. There are some very nice code generation facilities in JBuilder, but that's not the best way to learn in my view, and the price tag can be steep. Not available for Mac OS or HP-UX. Heavily integrated with Borland's other tools.

You can write Java source code in any plain text editor. You don't need an IDE, but it is pretty silly to think that you can do a project of any matter without one. If you can't decide, get Eclipse or NetBeans. If you want an excuse to go shopping, need an easy way to create desktop apps without writing Swing code, and aren't worried about proprietary JARs in your app, get IntelliJ. You'll probably use what you need to use as dictated by your current or prospective employer.

In this book, and at my work, I use Eclipse. But it doesn't matter. If you want to know exactly what it is that an IDE does that people think they can charge thousands of dollars for, we can talk about that for a second.

What I Hate About IDEs in 50 Words or Less

IDEs are great when you're learning, and great after you've learned a lot. Don't spend your time right now learning some IDE; spend your time learning Java. That might mean typing things like `public static void main(String...args)` a bunch instead of checking a box in a wizard. That's a Good Thing for now.

What IDEs Are Good at

I won't tell you how much easier your life will be if you get this or that IDE. You hear that a lot in tech books, about how such and such feature is going to make your life so much easier. That always makes me nuts. The fact is, until you learn the IDE itself, and especially until you learn Java, your life will get worse. Way worse. That's because it is hard work to learn something new of any complexity. More often than not, you've got a boss breathing down your neck, and you need to get going fast, and computer languages are complicated.

And with an IDE, you've really got two things to learn: getting around in the IDE, and the language you're interested in learning. And it is a bad idea to start learning some IDE before learning the language in question (in this case, Java). It's a bad idea because it is nonportable. You can't take your tremendous knowledge of JBuilder with you to IntelliJ (for instance). And one thing they never do is help you write well-designed, slim code. You need to at least know how to run java, javac, and set up your classpath before using an IDE. If you don't, you'll be forever on crutches.

On the other hand, you're going to use one because that's what all the cool kids do, and it is simply idiotic to try a project of any complexity without one. Most importantly, many IDEs do have terrific features that eventually are useful, including debuggers, code generators, Java-specific project organizers, and so forth. So, to make us feel a little less cynical about ponying up and clicking the download link, let's look at some of the many real benefits we get with most IDEs:

- **Debugging**. This is arguably the best reason to get involved in a relationship with an IDE. It can be very daunting to debug programs while you are learning, and debuggers improve your chances of working through problems quickly. You can specify a point to stop in the middle of the execution of your program, then step line-by-line through the code as it executes, and see all of the current values of all of your variables at each step. That is very helpful to see exactly where code is failing, and why. Using a debugger is not an immediately intuitive thing, but after you get a handle on it, this skill is more or less transferable between IDEs.

- **Code Completion**. This is a fantastic aspect of IDEs that is hard to do without, and is an excellent learning assistant. When you type the name of a package, a small menu pops up for you to choose available classes in that package. Or when you type a variable name, it knows to what class it is assigned, and pops up all of its fields and methods. Also, many IDEs will make suggestions about what you're trying to do. They'll tell you stuff like, "you declared this array but then never referenced it," or do you want to import this or that class. My favorite warning is from IntelliJ: "silly assignment." That's exactly what I thought throughout graduate school.

- **Integration with external tools**. You get an easy, consistent interface for configuring many external tools. While you're learning Java, this isn't too big of a deal. But you do get a way to run programs, and applets, and execute an Ant build.

- **Language Support**. After you've learned the basics of the Java language, you'll likely begin to expand the complexity and reach of your programs. Reach often means heterogeniety. IDEs often support multiple languages, including not only Java but also others you'll need, such as HTML, XML, JavaScript, JSP, and so forth. That comes in handy, and if your IDE is sort of configurable, you can go off the deep end.

- **Project Handling**. Notice that I didn't call this "Project Management" because that sounds misleading: an IDE does not do schedules or budgets or any of that managerial jazz. But it does make a clear way to organize your code, make packages, and deploy a complete application. This also means you can view not only the files in your project, but the structure of your code (its constructors, fields, and methods).

- **Refactoring**. Refactoring is something we'll talk about in more depth later. It means taking existing code and making it better without adding functionality. Refactoring is extraordinarily important to the long life and health of your projects. The IDE version of this often means making it easy to rename or move a package or class. Sometimes it means more. Again, this "something more" is something I'd like to do myself.

- **Code Generation**. There are a lot of repetitive tasks in writing code. The most common examples in object-oriented programming are probably getter and setter methods. Often, developers write classes that contain private variables and public methods to retrieve and update their values; those methods are called accessors and mutators, or more commonly (and perhaps less idiotically), getters and setters. It is wicked tedious to write all those, especially if you decide that your userID String should be an int. Or whatever. Now you have to go change two methods too. IDEs keep this kind of housekeeping in order with little hassle on your part. Be wary of generating huge blocks of meaningful code, however. If you want an IDE to do all your work, save your time and go do VB. I don't mean that. I take it back. I don't want you to go do VB. I think Java is better. I don't really like all that code generation for stuff like updating your database at all. In fact, I hate it. Don't you?

So I'm not going to tell you what to do. That would be beside the point, because probably you're going to use whatever they have at your school or your work. But if you're shopping around, this gives you some ideas. All I want to stress is: write your code by hand for a while—possibly a long while—so that you have real, portable skills.

Chapter 6. PRIMITIVE TYPES

DO OR DIE:

- Dig the primitives.
- Yeah, baby, I'm lovin' the primitives.

This short topic introduces the eight primitive types in Java.

About Java Primitives

Java primitives are not objects. They are simple values. They are used to represent single characters, numbers, and the logical values true and false.

Primitive types in Java have a non-trivial benefit over primitives in languages such as C and C++: the ranges of Java primitive types are not dependent on the underlying system. They do not change from system to system as you port your app, because of the Java Virtual Machine. This keeps you from the dangers of overflow problems you can run into in those languages. When you specify an int in C or C++, you don't know what the actual range is; your program will have to use whatever the range of the primitive is on the target platform. For example, in Java, you don't have to worry that on a 16-bit processor, your int will allow only a 16-bit range, but on a more recent processor, 64 bits. This could cause serious problems for your data, which would not be truncated, but would instead *wrap*.

Java stores its data as different types for different kinds of things. That's why it's called a strongly typed language.

For the numeric number, the lower range is calculated as 2 to the power of the number of bits minus 1. The upper range of the data types is calculated as 2 to the power of the number of bits it can hold minus 1, minus 1. No, that's not a typo. Consider this. How many bits are in a byte? Eight. The Java data type byte is capable of holding 8 bits of data. So now you know the range of possible values. Let's show our work like Mr. Foss made me do endlessly back in the 10th grade. He tortured me, and now you must be tortured too. To determine the range of a byte we do this math: 2 to the power of 8 bits - 1, -1. Which is: 2^{8}-1 or 2^{7}-1. And 2^{7} is 128, minus 1 is 127. We subtract one in order to accommodate 0, which is counted as positive. So the highest value we can represent with a byte is 127. The low range is one step easier to get: 2^{8}-1 is 2^{7} is 128. So a byte in Java is capable of representing the values -128 to 127.

Integer Types

The Java integer types are byte, short, int, and long.

byte

Occupies 8 bits or 1 byte, which is:

-2^7 to 2^7-1 or -128 to 127

Default value of 0

Example: -17, 123

short

Occupies 16 bits or 2 bytes, which is:

-2^{15} to $2^{15}-1$ or -32,768 to 32,767

Default value of 0

Example: 31,098, -9001

int

Occupies 32 bits or 4 bytes, which is:

-2^{31} to $2^{31}-1$ or -2,147,483,648 to 2,147,483,647

Default value of 0

Example: 50, 2147000000, -53000

The int is probably the most commonly used integral type in Java. That's because it is often not worth trying to save memory space by using a smaller type, such as a byte. Here's why: when you perform a comparison operation or an arithmetic operation with a byte, it gets promoted to an int anyway by the runtime, then the operation is performed, and then you have an int. So it's extra work, and a little confusing. But there are many situations where you need specifically one of those types.

long

Occupies 64 bits or 8 bytes, which is:

-2^{63} to $2^{63}-1$ or -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

Default value of 0

Example: 9,223,372,036,854,775,807, 0L

The long takes up a good deal of memory, and so it is typically reserved for use in situations that exceed the capacity of the int, such as representing the total population of the earth, the national deficit in dollars, or the total number of hours I've spent goofing off on the Internet.

You can distinguish between a long and other integral primitive types by writing a literal L after the number value, like this: 87999065L. If you don't do that, the compiler will assume that your number is an int.

Real Numbers

The Java floating point types are float and double. Floating point types are numbers with a decimal place.

float

Occupies 32 bits or 4 bytes, with 6 or 7 significant digits

Default value of 0.0

Range: 6-7 significant decimal digits

Example: 3.1459F

You specify the float by placing an F literal after the float value. If you don't, the compiler will treat your number as a double.

Java floating point numbers follow the IEE754 specification, which includes positive and negative signed numbers, positive and negative zero, and a special value referred to as NaN (Not a Number).

NaN is used to represent the result of an illegal or invalid operation, such as an attempt to divide by zero. The Float and Double wrapper classes both have a method called isNan(), which returns a true or false depending on whether the specified number is Not a Number.

double

Occupies 64 bits or 8 bytes, with 14 or 15 significant digits

Default value of 0.0

Range: 15 significant decimal digits

Example: 67, 2.0E10, 111,222,333,444.567, 67.99D

The double value in the preceding example includes an E. This is engineering notation for an irrational number representing the base of the natural system of logarithms. It has an approximate numerical value of 2.7182818284. This notation was first used in the mid-seventeeth century and has been calculated to 869,894,101 decimal places. You can use E in your double values like this: the number 8.9E5 is another way of writing 8.9 * 10^5.

You can place a literal D after the double value in order to explicitly indicate in your code that you mean for the value to be a double, as distinct from a float. This isn't necessary, however, as double is the compiler's default.

Characters

A single character is represented in Java with the char primitive. The Unicode char occupies 16 bits or 2 bytes, stores a - z, A - Z, 0 - 9, and numerous other characters including those on your keyboard, and characters in languages such as Hebrew, Korean, Arabic, and other doublebyte characters.

char

The char stores 0 to 216-1 or 0 to 65,535

Default value is '\u0000', which is null in Unicode

You can write Unicode char values using a \u and the code number that represents the desired character. Here are some examples of Unicode values.

```
'\u0030' is 0

'\u0039' is 9

'\u0041' is A

'\u005A' is Z

'\u0061' is a

'\u007A' is z

'\u000A' is LF (line feed will cause a compiler error)

'\u000D' is CR (carriage return - will cause

            compiler error)
```

A sequence of characters is represented by a java.lang.String object, not a char. And Java Strings are objects, not primitives. The char primitive is used to represent only a single character. This is different than in C and C++, in which a String is a primitive type. A char primitive may only hold one single character, and is actually a numeric type. So you use Strings to represent things like a name or a book title. You will learn a lot about Strings in Chapter 13. You can look up the codes used to represent Unicode characters at www.unicode.org.

Because the char is an integral type, it can be used in certain situations with mathematical operators. This ability allows us to "increment" a char. The following code uses the char with mathematical operators to print out the English alphabet from a to z, with each character separated by a comma.

Alphabet.java

```
package net.javagarage.demo.primitives;


/**
* Prints the 26 letters of the English
* alphabet.
*/
public class Alphabet {


public static void main(String[] arg){


    char letter = 'a';
    //initialize two ints
    int x = 1, y=26;


    while (x < y){
        System.out.print(letter + ",");
        //use the ++ operator to increment char by 1
        letter++;
        x++;
     }
}
}
```

The ASCII character set corresponds to the first 127 values in the Unicode character set.

Note that in the preceding example, we use a single statement (remember that a statement is some code that is delimited by a semi-colon) to initialize the values of two separate integers. This is legal, but some managers pooh-pooh the practice, citing that it is harder to read.

PREV < Day Day Up > NEXT

Logical Representations

The boolean is the only logical primitive type in Java, meaning it is capable of helping you perform logical operations.

boolean

The only type used to represent true and false values in Java is boolean. The word is written out completely, not shortened to bool as in C++, C#, and other languages.

- Possible values are true or false.
- Default value is false.

Note that in Java, the boolean values are literals: you may not substitute a 0 for false or a 1 for true as in other languages. If you are in the habit of using 0 and 1 in this way, break it quick. As a side note, if you are in the habit of returning -1 from a method to indicate a non-standard state, break that habit too. Java has a robust exception handling facility for such business.

So, yeah. I guess that's all I want to say about primitives right now. Let's get on to more interesting things. Like grouting some tile. Have you ever been forced to clean the teeth on your dad's rusting rakes all day on a Saturday? Me neither. But man that sounds boring.

Chapter 7. OPERATORS

DO OR DIE:

- Tumble over ice
- Add twist
- Serve

Operators are used in Java more or less as you would expect. Let's face it, operators are stoopid boring, mostly obvious, and 90% intuitive. But the section 23A of the Law of Tech Book Writing states that all programming authors must talk about operators. Okay. I'll make you a deal. We'll play nice, and deal with operators, and just get it over with as painlessly as possible.

Java features operators for arithmetic, incrementing and decrementing, logical operations, and bitwise operations. First, let's take care of the unpleasant business of operator precedence.

Operator Precedence

The compiler will follow certain rules when determining how to execute a mathematical statement containing multiple operators.

Lucky for us, they are the standard rules regarding operator precedence:

1. Operators inside parentheses are evaluated first.
2. Multiplication and division (left to right).
3. Addition and subtraction (left to right).

The following code illustrates these concepts.

OpPrecedence.java

```
public class OpPrecedence {

  public static void main(String[] args) {

    short x = 10 + 9 / 3 * 2 - 5 + (4 - 2);

    System.out.println(x);

  }

}
```

That code is a complete class that you can type in, compile, and run. I know we haven't talked about classes yet, but I'm going to try to do that a lot so you can get comfortable typing the code in and running it quickly anyway.

If you carefully read the preceding rules on operator precedence, you probably have fallen asleep by now. In the event that you managed to remain awake, you will know that the result is 13. Here's why:

4 - 2 is 2. Hang on to 2.

9 / 3 is 3

3 * 2 is 6

10 + 6 is 16

16 - 5 is 11

11 + 2 is 13

Straightforward enough.

Arithmetic Operators

These are +, *, -, and / for addition, multiplication, subtraction, and division. That's about it.

There is also the modulus operator, which looks like this: %. If you can't remember modulus, which actually comes in handy with surprising frequency, it is used to return the remainder of performing a division operation.

For example, 10 % 3 = 1, because 10 divided by 3 is 3 (which we don't care about) with a remainder of 1 (which is what we care about when we use modulus).

There's one more thing: you can use a special shorthand in combination with any of the arithmetic operators (excluding modulus) to modify an existing value.

Say you have a variable of type int called x and its value is 0, and you want to add 1 to it. You can do this the boring old nerdy way, like this:

```
x = x + 1;

//x is now whatever it was before (0 in this case),

//plus 1
```

Or you can do what all the cool kids do, and write it more concisely, like this:

```
x += 1;

//x is now whatever it was before (0 in this case),

//plus 1
```

With this incredible time-and space-saving feature, you can shave hundreds of minutes off of your programming time. (Maybe, I guess, if you programmed for billions and billions of years. And were a pretty slow typist.)

Although the way I have written it here is the most common way you see this feature used (with the addition operator adding 1 to the value), you could go crazy with it, using other operators:

```
int x = 18;

x /= 6;

//x is 3
```

The + operator is called an overloaded operator. That means it does something different in different contexts. We have seen how it works with integer values. It also can be used to concatenate (stick together) Strings. Like so:

```
String s = "pound";

String t = "com" + s; //compound

String x = "before ";

x += t; //value of x is now "before compound"
```

Incrementing and Decrementing Operators

This is the kind of thing you need to do with some frequency, especially within loops. Java makes incrementing and decrementing a variable by 1 easy with these special operators:

```
x = 0;

x++;

//x is 1
```

You can subtract the same way:

```
y = 10;

y--;
```

The preceding operators are called post-increment and post-decrement, because they add or subtract 1 *after* the variable has been evaluated.

FRIDGE

You may only use these operators with variables—not with literals. That is, you can't do this: 10++;. That's illegal! You can't do this either: String x -= "something";. No funny stuff, now.

You can also use these operators for pre-incrementing and pre-decrementing, which adds or subtracts 1 before the variable has been evaluated. Like this:

```
--x;

++y;
```

Here is a class you can compile and run to demonstrate these operators:

OperatorTest.java

```
public class OperatorTest {

public static void main(String[] args) {
  int x = 10;
    //evaluate, then increment
  System.out.println("post decrement: " + x++);
    //prints 10
  System.out.println(x); //x is now 11
    //decrement, then evaluate
  System.out.println("pre decrement : " + —x);
    //prints 10
}
}
```

Relational Operators

There are relational operators used for testing whether a value is greater than or equal to another value. They are shown in Table 7-1.

7-1. The Logical Operators

OPERATION	OPERATOR	EXAMPLE
Equal to	==	(x == 1)
Not equal to	!=	(a!= b)
Less than	<	(i < 10)
Greater than	>	(i > j)
Less than or equal to	<=	(j <= 10)
Greater than or equal to	>=	(x >= y)

Conditional Operators

There are operators used to express boolean relationships between expressions (see Table 7-2).

7-2. Boolean Operators

Operation	Operator	Example

AND	&&	int x = 1;
		int y = 10;
		((x == 1) &&
		(y==10))

OR	||	((y >2) || (true))
NOT	!	(! x)

The boolean operators are used in expressions to reduce to a boolean value of true or false.

The operators short circuit, which means that if it is not necessary for the second expression to be evaluated, it is not evaluated. In the following example, the second expression is never looked at by the runtime:

```
if ( (false) && (true) ) {}
```

Because the first expression is false, we know that both the first expression AND the second expression evaluate to true. So we don't go there.

By the same token, the second expression will not be evaluated in this example.

```
if ( true || false)
```

In a boolean OR operation, only one of the terms needs to be true for the statement to evaluate to true, so because the first one is true, we already know the whole thing must be true.

Ternary Operator

This operator is a shorthand version of the if/else construct that some developers find confusing, and others like. It takes the following form:

```
boolean expression ? action if true : action if false
```

The first part is an expression that evaluates to either true or false. The code after the question mark executes if the expression is true; the code after the colon executes only if the expression is false.

It serves as a replacement shorthand for the following construct:

```
if (boolean expression) {
   action if true;
} else {
   action if false;
}
```

So it looks like this in code:

```
public int returnLesserOfTwo(int x, int y) {
   return (x < y ? x : y);
}
```

If you pass that method an x value of -20 and a y value of -80, it returns -80. I like it. Just don't nest them if you want to eschew obfuscation. Here's what I mean. Three nested ternary operators just isn't pretty.

```
private static boolean badTernaryNest() {

   return

(6%3>1)?(5>10?(2<=3):(3*4<10?4>3:7==7)):false;

}
```

Although you can do that (it returns false if you're interested), please don't. It is almost guaranteed to incur the wrath of anyone who has to read your code.

Binary Numbers and Logical Operators

For all of my griping in this chapter, the bitwise operators are actually very cool. The bitwise operators let you manipulate the individual bits that make up an integer value. You do so using the operators & (AND), | (OR), and ^ (XOR, the EXCLUSIVE OR). Using these operators with an integer value lets you determine the value (0 or 1) for the bit in the specified position.

If you haven't used these operators before, it might sound ridiculous to think of "2 OR 6 = 2" as being a meaningful expression. But it is. For that expression to make sense, we need to examine the bits of a number, which means you need to be able to visualize a base 10 number as a base 2 number. Base 10 numbers are what we use every day. It includes the numbers that we can make using the digits 0-9. Base 2 numbers use only the digits 0 and 1 for representing values.

The bitwise operators take a number in base 10 and look at it as its base 2 counterpart and do something with the bit value at the specified position. Still sounds opaque. Let's circle around again.

How do you represent a base 10 number as a number in base 2? You only have a 0 and a 1 available to you. What you do is generate a number line that starts with 1 and goes out to the left. The numbers represented on the number line are the powers of 2. Here is such a number line that goes out only a few powers of 2:

```
64 32 16 8 4 2 1
```

We get this number line because 2_0 is 1, 2_1 is 2, 2_2 is 4, 2_3 is 8, and so on through 2_6. Now we can use this number line to take values from. We can represent any base 10 number between 0 and 127 using these numbers. We can do that by adding different numbers together.

For example, 3 is a base 10 number that isn't on the number line. We can represent 3 by using only a 2 and a 1, and none of the other numbers on the number line. We can get 22 by using the 16 and the 4 and the 2, and none of the other numbers.

If we want to represent a base 10 number in binary, we take the numbers we need and leave the ones we don't. To take a number, we put a 1 under it; we put a 0 under numbers that we don't need to use.

So let's do that now. Let's represent 3_{10} as 3_2.

64	32	16	8	4	2	1	
0	0	0	0	0	1	1	(1 + 2 = 3)

If we take the 0s and 1s above as our number, we get 11. 3_{10} = 11_2. So our answer is 11.

Let's do 22 now.

64	32	16	8	4	2	1	
0	0	1	0	1	1	0	= 22_{10} (because 0 + 2 + 4 + 0 + 16 = 22)

So we represent 22 base 10 in base 2 as 10110. The 1 means that bit is on (or true) and 0 means that bit is off (false).

Now we can use this as the basis to perform our logical operations. Table 7-3 shows the outcome of boolean expressions.

7-3. Boolean Operator Expression Results

AND	true AND true = true	true AND false = false	false AND false = false

OR	true OR true = true	true OR false = true	false OR false = false
XOR	true XOR true = false	true XOR false = true	false XOR false = false

In that table, if we substitute 1 for every "true" and 0 for every "false," perhaps you can start to see how we could say something like 3 AND 22.

Let's take our base 2 representation of 3 and our base 2 representation of 22 and line them up on top of each other, like this:

64	32	16	8	4	2	1	
0	0	0	0	0	1	1	= 3 (1 + 2)
						AND	
0	0	1	0	1	1	0	= 22 (0 + 2 + 4 + 0 + 16)

Now we just need to draw a vertical line between each bit that makes the 3 and each bit that makes the 22, performing the logical operator AND on each one. Here we go:

1: 1 AND 0 = 0

2: 1 AND 1 = 1

4: 0 AND 1 = 0

Skip 8 because it isn't used.

16: 0 AND 1 = 0

So now we've got a 0 (not used) in every place but the 2. The 2 bit has a 1 (true). So, the answer to the expression 3 & 22 is 2.

Let's try another quick one, because it's so fun. What's 13 ^ 6 (13 eXclusiveOR 6)?

64	32	16	8	4	2	1	
0	0	0	1	1	0	1	= 13 (because 1 + 0 + 4 + 8 = 13)
						XOR	
0	0	0	0	1	1	0	= 6 (because 0 + 2 + 4 = 6)

We can do this quickly now. Remember that exclusive or means that each operand must be a different value.

1: 1 XOR 0 = 1

2: 0 XOR 1 = 1

4: 1 XOR 1 = 0

8: 1 XOR 0 = 1

The answer is 1 + 2 + 0 + 8 = 11. 13 ^ 6 = 11.

Here is some Java code that prints out these different values just to prove it.

BooleanOps.java

```
public class BooleanOps {

  public static void main(String[] args) {
    System.out.println(3 & 22); //2
    System.out.println(13 ^ 6); //11
    System.out.println(0 & (3-3)); //0
    System.out.println(45 | 16); //61
    System.out.println((2 & 5) | ((7^3) & 5 )); //4
  }
}
```

Now that we can see behind the base 10 and into the bits, we can use the bit shift operators to manipulate the individual bits of a number.

PREV < Day Day Up > NEXT

Shift Operators

There are three shift operators, used for changing the bit values in integral types. They aren't used frequently, except in programs that need to keep their memory usage to an absolute minimum. The shift operators are

1. The left shift, represented as <<
2. The signed right shift, represented as >>
3. The unsigned right shift, represented as >>>

The left-hand side of the operators is the value to be shifted, and the right-hand side is the number of places to shift it. The type of each operand has to be an integral type, or a compiler error will occur. The shift happens on the 2's complement integer representation of the value on the left.

When the value to be shifted is an int, only the last 5 digits of the right-hand operand are used in performing the shift. When the value to be shifted is a long, only the last 6 digits of the right-hand operand are used in performing the shift.

The shifts occur at runtime, on a bit-by-bit basis.

Let's show the bits for the number 13, which we learned how to do in the last section.

64	32	16	8	4	2	1	
0	0	0	1	1	0	1	= 13

If you shifted all of the bits that make up that number 1 to the left, what would you have? You'd have this:

64	32	16	8	4	2	1	
0	0	1	1	0	1	0	= 26 (because 2 + 8 + 16=26)

We have the left shift operator to do this work for us, and we can represent it in Java code like this: (13 << 1).

What if we shifted 13 over 1, but to the right?

64	32	16	8	4	2	1	
0	0	0	0	1	1	0	= 6

Wait a moment. What happened to the 1 value that we originally had sitting under the 1 position? It just dropped off the map. It is forgotten about completely, and we have only what's left. If we shift 13 over to the right 10 positions, or even 938 positions guess what we end up with? 0. That's because the regular right shift operator (>>) is signed. Use the unsigned right shift operator to move 13 over 1 place, and you get the same result (6).

So what if we use a negative number to start with? For example, (-13 >>> 2)? Should be a fairly similar, innocuous result, and then we can call it quits. I'm afraid that (-13 >>> 2) = 1073741820.

What in the world will become of us?

Well.

The value of num<<pos is num shifted to the left pos positions.

The value of num>>pos is num shifted to the right pos positions, with the sign extension retained. The result is num/2pos. For non-negative numbers, performing a right shift is equivalent to dividing the left-hand operand by 2 to the power of the right-hand operand.

The value of num>>>pos is num shifted right pos positions with 0 extension. For all positive nums, the result is the same as using the >> operator. The difference between >>> and >> is that >> fills in all of the left bits with zeros. That means that *the sign value can change.* For negative nums, the result is equivalent to num right-shifted by pos plus two left-shifted by the inverted value of the right-hand operand.

There are limited contexts in which you need to use the shift operators. Many of the Java APIs shield us from having to deal with low-level bit manipulation tasks, so we can probably quit talking about them now. Anyway, more talk at this point would just be more of the same.

Chapter 8. CONTROL STATEMENTS

DO OR DIE:

- To be the smooth jazz of Java chapters

Control statements are the things you write to control the flow of your program. They're pretty straightforward for the most part, and very similar to how these constructs are implemented in other languages.

So I am going to go out on a limb here. I am betting that you're going to be familiar with when and why to use stuff like if/else statements and for loops. Maybe we can spare ourselves the formality. If you're not familiar, that's okay. These are basic elements in any programming language, and there are hundreds of examples of their usage throughout the remainder of this book.

If/Else

The if/else statement is used to execute code when a test condition is true.

```
if (x > 10) {
   System.out.println("x is greater than 10");
} else {
   System.out.println("x is less than 10");
}
```

Switch/Case and Breaks

The following code shows how to use a switch/case construct with a char. But any of the following primitive types are legal to test against: char, byte, short, or int.

```
switch (test) {
case 'A' :
      System.out.print("Found X");
      break; //Print 'X' if test = 'X'
   case 'B' :
      System.out.print("Found Y");
      break; //Print 'Y' if test = 'Y'
   default :
      //This code runs if test does not equal 'X' or 'Y'
      System.out.print("Found Z");
}
```

If neither break were there, and if test equaled 'X', the code would print XYZ. Java switch/case constructs "fall through" until they encounter a break statement or the end of the switch.

The break keyword is used to stop processing within a case statement. Note that the keyword default is used to specify the statement to execute in the event that the passed number does not equal any of the explicit case values.

While Loop

The while loop performs its test against the truth value of the expression *before* the loop executes. That means that if the test expression is initially false, the loop will never run.

```
int x = 0;

while (x < 10) {

   System.out.println(" x is now : " + x);

   //increment x by 1

   x++;

}

System.out.println("x is not less than 10 anymore: " + x);
```

Note that we are only able to refer to the variable x outside the loop because it was declared outside the loop. If you declare a variable inside a loop, you won't be able to reference it outside the loop.

Do-While Loop

The do-while loop differs from the while loop in two ways. Most importantly, the code is guaranteed to run at least once, even if the test expression evaluates to false on the first test. That is the point of this loop. The second way it is different is that it is hardly ever used. I mean, hardly ever used at all. But it's easy.

```
int x = 0;

do {

   System.out.println(" x < 10 : " + x);

   x++;

} while (x < 10); // test not performed until after

 // code block is executed at least once

System.out.println("x is not less than 10 anymore: " + x);
```

For Loop

The for loop is perhaps the most popular of all loops. It comes in two versions: one for iterating against a test expression, not unlike the while loop, and a second for iterating once for each item in a collection (such as an array or Hashtable).

The first kind of for loop has the following basic structure:

```
for(initialize a variable; test the expression;

operation to perform after iteration of loop)

{

//statements to execute when loop is true

}
```

Here is a simple example:

```
for (int x = 0; x < 10; x++){

   System.out.println(" x < 10 : " + x);

}
```

Note that at this point in the code, we cannot refer to the current value of x (10) out here. That's because we didn't declare it outside of the for loop, but rather declared the variable as part of the loop construct itself.

You may optionally omit the curly braces used around the statement that executes in the event that the test condition returns true. However, you must keep in mind that only the *first* statement following the for loop will be executed; if you want to do more than one thing inside the for loop, you have to use the curly braces. Otherwise, that second line of code doesn't get executed until the loop exits entirely (the test condition returns false).

Complex For Loop

The for loop can also facilitate more complex statements. Taking the same basic structure illustrated in the preceding sections, you can add initializing statements and more operations to be performed after the loop iterates. The additional initializers and operations are separated by commas. Here is what I'm talking about:

```
//two initializers and two post-loop operations

for(x=0, y=0; x <10; x++, y++);
```

Here is an example:

```
private static void complexFor(){

  int iStop = 8;

  int jStop = 21;

  for (int i = 0, j = 0; ((i < iStop) &&

      (j <= jStop)) ;

                         i++, j=i * 3)

    {// \t is the tab character

    System.out.println("i: " + i + "\tj: " + j);

  }

}
```

Note that in Java a simple semicolon (;) usually used to mark the end of a statement is also considered a complete statement itself. That makes the following entirely legal:

```
int i = 0, j = 5; for ( ; i < j; i++, j++ ) { ; ; }
```

From the cool-and-weird-code-to-impress-your-friends department: ask them to write the simplest possible *legal* for loop that executes a statement. Here's the answer:

```
for ( ; ; ) ;
```

Then for a practical joke, have them run the code. Can you guess what happens? It's an infinite loop that runs until it crashes their virtual machine! Hilarious. No one ever tires of a really *good* joke….

FRIDGE

Note for Nerds

The for loop actually compiles into a while loop in Javaland. Because they do the same thing, the compiler finds it more efficient to just manage one of them. So in a sense, a for loop is syntactic sugar. Which is weird for such a mainstay kind of a statement. But this is not true of the foreach loop, which behaves differently.

Nested For Loop

You can nest these different constructs within each other. You can put a while loop inside a for loop, and so forth. You'll find that putting one for loop inside another is commonly necessary. To demonstrate, let's print a multiplication table through 10, making a new line after every 10 results.

```
private static void timesTable(){
  for (int i = 1; i <= 10; i++){
    for (int j = 1; j <= 10; j++){
      System.out.print(i * j + "\t");
    }
    System.out.println();//makes a new line
  }
}
```

Here, we use the \t character to separate each result with a tab.

Continue and Labels

The continue keyword is used to skip the current iteration of a loop if some test condition returns true. There are two ways to use it. The first is without a label.

```
//the continue statement must be used inside a loop,
//or your code won't compile.
private static void demoContinue(){
  for (int i=1; i <= 100; i++){
    if ((i % 4) != 0)
      continue;
    System.out.println(i);
  }
}
```

The preceding code will print out every multiple of 4 between 4 and 100, inclusive. It does so by testing: "If I divide the current number by 4 and do not get a remainder of zero, don't print out this number." The continue keyword causes program execution to jump into the next iteration of the loop.

The second way to use continue is with a label, which allows you to explicitly state the place in your code you want to jump to. These are sort of useful for inner loops. In fact, a label must be specified immediately prior to a loop definition. You can just put a label anywhere and jump to it with continue. Here's an example:

```
private static void demoContinueLabel(){
    //here outer: is a label indicating
    //a named reference point in the code
    outer:
  for (int i=0; i < 4; i++){
    for (int j = 0; j < 4; j++){
      if (i < 2)
        //go to the label
        continue outer;
      System.out.println(" i is " + i + " and
           j is " + j);
    }
  }
}
```

FRIDGE

There is no goto keyword in Java. It was determined a long time ago that goto is evil and does bad things to good programs. So, although it has no meaning in the Java language, you are not free to use it as an identifier (variable name), because goto is a *reserved* word.

This code produces the following result:

```
i is 2 and j is 0
i is 2 and j is 1
i is 2 and j is 2
i is 2 and j is 3
i is 3 and j is 0
i is 3 and j is 1
i is 3 and j is 2
i is 3 and j is 3
```

That's because the continue keyword is used to jump out to execute the next statement after the label we named outer in the event that i is less than 2. So, the print statement is only reached when i is equal to or greater than 2.

Labels are almost *never* used in Java. Neither is the continue keyword for that matter.

They're considered kind of bad practice. Although it works great in the preceding example, if you catch yourself starting to type continue inside a loop, stop, put your hands up, and step away from the keyboard. Consider if it is really necessary. You'll find it is perhaps never necessary, and you can simplify your code in more obvious ways that improve its readability and predictability.

FRIDGE

We won't talk about Collections until the sequel to this book, *More Java Garage*, but for now know that they are Sets, Hashtables, Vectors, ArrayLists, and other things that you might be familiar with. If none of that is ringing a bell, Java Collections (in a gross generalization) often act like resizable arrays. They're different classes that store lists of objects that you can iterate over.

Enhanced For Loop (for each loop)

The enhanced for loop, which acts like the *for each* loop that you might be familiar with from other languages such as VB.NET and C#, gives you way to cleanly iterate over a collection with less fuss and muss.

Here is the syntax of the *for each* loop:

for (ElementType element : expression) statement

That syntax translates to this in practice:

```
for (Object o : myCollection)

    { //do something with o }
```

Note that the expression must be an instance of a new interface, called java.lang.Iterable, or an array. The java.util.Collection interface has been retrofitted to extend Iterable. The exclusive job of this new interface is to offer the opportunity for implementing classes to be the target of the new *for each* loop.

FRIDGE

This example uses a complete class to demonstrate the for-each loop. Don't worry if this looks unfamiliar to you. There are more examples without the surrounding class code. We'll get to classes in a moment.

The authors of the Java language preferred the colon to adding a couple of keywords (foreach, and in), because adding keywords can leave a language vulnerable to a lot of retro-fitting difficulties. I sort of hated the colon syntax at first, but now I like it because it is very easy to spot in code. This kind of loop is often represented like this in other languages:

```
foreach (ElementType element in collection) statement //C#
```

The preceding syntax uses the foreach keyword, which I find very readable and intuitive. And it's not like the Java authors haven't added keywords to the language before (assert was added in 1.4). Maybe there is a ton of code out there using foreach and in as identifiers (!). Well, either way, I'm happy it's here.

Let's see it in action.

ForLoop.java

```
package net.javagarage.efor;

import java.util.*;

/*
demos how to use the enhanced for loop that is new
with Java 5.0. it acts like a foreach loop in other
languages such as VB.NET and C#.
*/

public class ForLoop {

public static void main(String[] ar){

    ForLoop test = new ForLoop();

    for (Kitty k : test.makeCollection())
    System.out.println(k);

    }

    public ArrayList<Kitty> makeCollection(){
      ArrayList<Kitty> kitties = new
ArrayList<Kitty>();
      kitties.add(new Kitty("Doodle"));
      kitties.add(new Kitty("Noodle"));
      kitties.add(new Kitty("Apache Tomcat"));

      return kitties;
    }
}

class Kitty {

   private String name;

   public Kitty(String name) {
      this.name = name;
   }

   public String toString(){
      return "This kitty is: " + name;
   }

}
```

The output is as expected:

```
This kitty is: Doodle
This kitty is: Noodle
This kitty is: Apache Tomcat
```

You can also use the enhanced for loop with arrays:

```
package net.javagarage.efor;

public class ArrayForLoop {
   public static void main(String[] ar){
   /*this says, 'for each string in the array
   returned by the call to the makeArray() method,
   execute the code' in this case println)*/
```

```
  for (String s : makeArray())
     System.out.println("Item " + s);

  public static String[] makeArray(){
     String[] a = new String[] {"a","b","c"};
     return a;
  }
}
```

This looping construct affects only the compiler, not the runtime, so you shouldn't notice any difference in speed. A for loop is compiled into a while loop anyway, so there is a lot of sharing going on under that particular hood.

The array form of the for : statement generates a compiler error in the event that the array element cannot be converted to the desired type via assignment conversion. Here is an example of using the foreach loop to loop over an array:

```
int sum(int[] a) {

    int result = 0;

    for (int i : a)
       result += i;

    return result;
}
```

This code says, "declare an int called 'result' with a value of 0 to start with." Then, for every int in the passed in array (called 'a'), call the current iteration value 'i' and add it to the result. Very simple, clean, elegant.

Notice that the new loop does not use the term *for each* like other programming languages (such as C#) do. Instead, it uses a :. The new syntax is cleaner and simpler. When you combine the simplicity of the for each loop with the new power of generics (introduced in Java 5.0, and discussed in the upcoming book *More Java Garage*), you can write very clear and clean code.

```
void processNewOrders(Collection<Order> c) {
for (Order order : c)
order.process():
}
```

This code is immediately understandable: you see what type you are using in the signature and in the for loop. It is shorter, and every line lets you know what kind of type you are dealing with. As you might guess, it works with nested loops too.

Okay. Nuff said.

PREV < Day Day Up > NEXT

Chapter 9. CLASSES

DO OR DIE:

- Keep your cool
- Act natural
- Maintain

So far we have looked at setting up your environment, and using primitives, operators, and statements. We've gotten the necessary groundwork out of the way, and now we're free to explore classes. It's been, well, I admit it, a little bit lame. I'd like to make it up to you.

From here on out, things will be a lot more interesting, exciting, and new. I promise. It will be like when Abbott and Costello met the Werewolf, or when the Harlem Globetrotters came into ScoobyDoo—just an unbeatable entertainment combo. So please turn your attention to the magnificent JumboTron. A world awaits. The Java class….

The class is the fundamental unit of expression in Java. In this topic, we will learn how to make classes, what they are for, how to modify them to suit your evil purposes, and how to design them.

What Is a Class?

A class typically represents a type of thing in the world of the application. You define a class, and in it write code that represents two things: the properties of that type of thing and how it behaves. You might think of classes as categories. For instance, Person is a category or type of thing in the world. To make a class, you simply open your editor, type the keywords `public class`, and give it a name. Save that file with a .java extension and compile it using the `javac` command that comes with the SDK.

We can make a class called Person like this:

```
public class Person { }
```

If you type the preceding code into a text file called Person.java, it will compile. However, it is obviously useless, because it doesn't hold any data, and it doesn't know how to do anything.

We'll get to how to make it do stuff in a moment.

Classes are blueprints for objects. The chief job of the Java programmer is to write classes that represent different aspects of your application. You write classes. That's all you do (pretty much—you also have to implement the ways that classes interact with each other (which you usually do in other classes)). After you have written a class, you have defined *the sort of thing that is possible in your system*.

After you write a class in a source file (a .java file), compiling the source will result in a new file being created on your file system: a .class file.

After you run your program, it will almost certainly create objects. An object is *a particular instance* of a class. A class is a definition of a thing, and an object is a specific example of that thing. Objects are what you use in your running program to get stuff done.

For example, a person is a class (or kind) of thing. You are a particular instance of that category. You are an object of type Person. Randy Johnson is a different object of type Person. Luis Gonzales is yet another particular instance of the general class of thing called Person. Blah blah blah. You get the idea.

To make a new object of a class, use the `new` keyword, like this:

```
Person eben = new Person();

Person pawel = new Person();
```

We now have two different objects of type Person, each capable of holding their own separate data. They will only be capable of doing the same kinds of things though (because the class defines what they are capable of).

The main things that classes do are hold data (member variables) and define functionality (methods or functions).

An object in Java should generally have some properties. Perhaps our Person has eye color, weight, height, gender, and so forth. Each person—that is, every object or instance of the Person class—will have these properties.

Every person can also do things. A person generally can walk, talk, smooch, drink, and so forth. Each of these actions could be defined in a method of our Person class.

Let's look at how to define these different components. First, we are going to talk about comments. Although they are not the most important things about a class, you will see them inside the class example code, and I want you to know why they are there.

Classes Have Comments

Comments are used to remind yourself or inform others of what is happening in your code. They are ignored by the compiler and should be used extensively.

There are two notations for writing regular comments in Java code. The first is a regular single-line comment that looks like this:

```
public class Person {
   //this is a single-line comment
}
```

The second is used for comments that span multiple lines.

```
public class Person {
   /*
    This is a multi-line comment, and
    anything inside this is hidden from the compiler.
   */
}
```

Comments are really important. Please comment your stuff. It is annoying to point out the obvious, but it is certainly a good idea to keep track of changes to the code, indicate what other code uses it or relies on this code, who wrote it and when, and other things that will be useful down the road. When you comment your code, first think, "What is not immediately apparent about why something is happening here? What would a stranger who is a good Java programmer want to know about this in case of a problem or the need to change it (because that is likely when they'll be looking at your code)?" Notice that the author's name and e-mail are often useful in this regard too. If you answer questions like these, everyone will appreciate it.

Somebody at JavaRanch.com wrote that he writes comments for the psychotic programmer who will read his code, and who knows where he lives. I like that rule. It's a good rule.

I have a template that I sometimes use to comment files. I don't always use it in this book because it takes up a lot of room and you aren't interested in maintaining this code. I use it every once in a while just to emphasize the importance of commenting.

```
/* File: Person.java
 * Purpose: Represents a human who is going to party.
 * Uses: Address

 * Used By: Some other class name

 * Author: E Hewitt

 * Date: 12.31.99

 */
```

You can legally place comments at the very beginning of the file, at the end of the file, and scattered throughout the middle of the file.

Notice that there are obvious exceptions to where you can place comments. You can't place them in the middle of an identifier or in such a way that the comment would prevent the code from compiling, like this:

```
public class Person { //this is a righteous class! }
```

Now, you can't do that, because the comment hides the curly brace that closes the class. Of course, if you move the curly brace to the next line, you're good to go. But, duh. You wouldn't try to do that. What was I thinking? I must need some more Smarties.

Java has an extensive facility for commenting your code called JavaDoc, which we discuss later in this book. This is enough for now I reckon.

Classes Have Data (Members)

A class' data means its variables. These are often called members or fields. I'll use these terms more or less interchangeably because that's how you'll hear them used in the world, and it's good to get used to how others will talk about them. The data are the properties of the type of thing your class represents. (Often the term properties is used to mean only the public member variables of your class—but we'll talk about visibility in a moment.)

It is not a requirement that a class has data. You can define a class that only contains methods, and has no variables at all. That might be a good idea, depending on what your application requires. For example, our application might specify that a Person is the sort of thing that has a name. Let's expand our class to include this functionality:

```
public class Person {

    public String name;

}
```

Typically, your class' members are made up of the nouns you can list about your object. You find out what data your class needs to have by asking yourself, "What properties does a Person have?"

So our Person might have a name, an age, and an address. These could all be properties we add to our class.

```
public class Person {

   public String name;

   public int age;

   public Address address;

}
```

FRIDGE

In software analysis, this is called a HAS-A type of relation. That is, a Person HAS-A name. Another type of relation is IS-A, and we'll talk about that in the Inheritance chapter.

Now, every time we make an object of class Person, we know that we can define each of these bits of data (name, age, and address) for every different Person we make.

Notice that classes can hold data that is of other class types, and these types can be built in to the Java API, or they can be other classes that you define yourself.

The name variable is a variable of type String. String is a class representing a character sequence that has already been written for you as part of the Java API. If Sun didn't provide the API, you would have to define the String type yourself. Because Strings are things that everyone needs, it is very convenient that they are predefined for us.

The age variable is an int primitive type. Because Java supports autoboxing and unboxing of primitives, it acts like an int primitive when that's what you need, and it acts like an Integer wrapper class when that is what you need.

The address variable is declared to be of type Address. Address is *not* a class defined in the Java API, so the assumption is that *you* have defined a class called Address that has properties of its own (such as street, zipCode, or whatever). If you don't make that class first, the Person class won't compile.

You generally can get the value of each member of an object using the dot operator. I write "generally" because you might not have direct access to data from other classes, which is a matter of *visibility*, which we'll deal with later. For now, it is enough to know that our data members in the Person class are all declared to have public visibility, which means that any other class can read or write their values.

Let's make an object, set the value of its members, and then get the value of each member and print it to the console. I'm going to leave the Address member out of this one for now, so we can compile it.

Person.java

```
//declare the class

public class Person {

//define our two variables

public String name;

public int age;
```

```
  //main method is required starting point

  //for Java programs

  public static void main(String[] args) {

    //make the object of type Person

    Person p = new Person();


    //we can now refer to this particular object

    //as "p" to do stuff with it


    //set the value of the name member

    p.name = "Eben Hewitt";

    //set the value of the age member.

    p.age = 32;


    //print out our values

    System.out.println("The person named " + p.name +

        " is " + p.age + " years old.");

  } //end of main method

} //end of class
```

The output of running the Person class is like this:

```
The person named Eben Hewitt is 32 years old.
```

Having compiled the class, you will have generated a second file on your disk called Person class. This is the executable bytecode that the Java runtime interprets. You can open that class file and view it in a text editor, though many of the characters will not be readable, and you must not edit it directly.

Let's make it a little more complicated, and define a second class in the same source file (the one that ends in .java). This will be our Address class. Now that we know what an Address is, we can specify that a Person has a member of type Address.

Hey, I know this code is starting to look a little long. But *a lot* of it is comments. I think it's important to read what is happening line by line; it helps put all of the pieces together, which is also why I'm showing the complete listing. So don't worry.

PersonWithAddress.java

```
public class PersonWithAddress {
  //define our two variables
    public String name;
    public int age;
    //added the gnarly new Address member!
    public Address address;

    //main method is required starting point
    //for Java programs
    public static void main(String[] args) {
      //make the object of type PersonWithAddress
      //PersonWithAddress person = new //PersonWithAddress();

    //we can now refer to this particular object
    //as "person" to do stuff with it

    //set the value of the name member
   person.name = "Bill Gates";
    //set the value of the age member.
   person.age = 32;
    /*
    set the address. wait! it's a class itself. so we have to make an instance of
➥   address, and set its members, and then refer to those vars
    */
   person.address = new Address();
   /*
   now the address field refers to an object of type address, and you can fill up its
➥   members also using the . operator
   */

   person.address.city = "Mann Avenue";
   person.address.city = "Muncie";
   person.address.state = "Indiana";
    //print out some values
    System.out.println(
    "The person named " + person.name
    + lives in " + person.address.city + ", " +
    person.address.state + ".");
    } //end main method

} //end of PersonWithAddress class


//Whoa—starting a different class in the same

//source file—ok, party on Garth...


class Address {

  //define a coupla public members,

  //just like Person!

  public String street;

  public String city;

  public String state;

} //end Address class
```

This outputs the following:

```
The person named Bill Gates lives in Muncie, Indiana.
```

Classes Have Behavior (Methods)

We use the word behavior colloquially to refer to actions that someone performs. If the actions are performed repeatedly, Aristotle would call that character. My wife told me today that she thinks that Aristotle is a "jerk," but that's not important now. To you, anyway. I will admit I was a bit nonplussed. But only a bit—the "unities" of time, place, and action are pretty goofy if you get right down to it. You define the behavior of a *class* in its *methods*.

Each method (sometimes called a function, or in VB, a subroutine) represents a different little bit of functionality. Here is what makes up a method.

A method does one job. Like variables, methods have *names* or *identifiers*. A method name can have the same name as a member variable, because the parenthetical argument list keeps them from being ambiguous in the eyes of the JVM.

FRIDGE

If you are having a hard time compiling and running your programs using the command-line tools (javac and java), you can get an IDE. An IDE can give you insight about what is legal code at a particular point and color-coding different parts of your programs to improve readability. I recommend Eclipse, available for free from www.eclipse.org. You can also try Sun ONE Studio or NetBeans or Forte (whatever they're calling it today), which is also available for free from http://java.sun.com. (My favorite is IntelliJ IDEA, but it's $)

They have a degree of *visibility*, which indicates what other classes (if any) can call them. Methods can have visibility of default (no visibility explicitly declared), public, protected, or private.

Methods also have a *return type* that indicates the one value that represents the result of performing the method. Methods can take zero or more *parameters*, or values that you pass into them. Parameters are declared using the type of variable you want to accept and an identifier for each variable for use inside the method body.

Methods might throw exceptions, but we talk about that in Chapter 21, "Handling Exceptions."

Methods have a *body*, which consists of zero or more statements in between curly braces that do the work of the method.

Here is an example:

```
public class Person {

  private int age;

  public int getAge() {

    return age;

  }

  public void setAge(int ageIn) {

    age = ageIn;

  }

}
```

This class has one member variable and two methods: getAge() and setAge(). The getAge() method declaration says it returns an int, and the body of the method indeed does nothing but return the value of the age member, which is of type int.

The getAge() method's visibility is declared to be public, so any other class is allowed to use the getAge() method. This method takes no arguments at all.

The second method is called setAge(), and it also has a visibility of public. But it has a return type of void, which means that the method does some stuff but does not return any result to the caller.

A constructor is a special type of method that gives you control over how you create your objects. We'll talk about those in a bit.

Return Types

It may seem obvious, but if you say that a method will return a value, it *must* return a value. You can't do this:

```
public int getAge() {

  int i = 7;

} //won't compile!
```

This method will not compile because you say you're going to do something (return a value) and you don't keep your promise. By the same token, the following is also illegal:

```
public int getAge() {
   return; //won't compile!
}
```

Here, we say we're returning an int, but we don't: returning nothing is not the same as returning something (any kind of int).

The following, however, *is* legal:

```
public void ohWellWhateverNeverMind() {
   return ;
}
```

You can do that because you are returning nothing, which is what you say you will return. That method is equivalent to this:

```
public void hahaha() { }
```

Which is also legal. You don't have to have anything in the body. Though that doesn't do much for you except help you compile your class if you aren't sure what code to put there yet. Both are equivalent to the following, which is also legal:

```
public void allYourBase(String areBelongToUs) {
   ; ; ;
   ; ;
}
```

Again, you can do it, but it's probably not too common. If you see that sort of thing a lot at your job, you might reconsider employers.

Methods can return the result of evaluating an expression that contains literals, like this:

```
public int getADumbOldNumber() {
  return 7 * 54;
}
```

Methods can return a literal, like this:

```
public String getMessage() {
  return "I love you, man...";
}
```

Methods can return the result of a call to another method, which is just another form of an expression.

```
public double getRandomNumber() {
  return Math.random();
}
```

If you think it would be a good idea to be very explicit about what you're doing to help your reader, or if your method consists of a number of statements, you can return a result that you hold in a variable, like this:

```
public double getRandomNumber() {
   double d = Math.random();
   return d;
}
```

Methods can return a type that can be converted at runtime into the declared return type. For example, a float is a floating point type that holds 32 bits of data. A double holds 64 bits of data. That means that any float can fit into a double container. That means that the following is perfectly legal, but confusing:

```
public double getIt() {
  float f = 16F;
  return f;
}
```

Although the number you create inside the method body is of type float, it is a double when the method returns. So, here's a quick quiz. Is the following legal?

```
public float getFloat() {
  return 16F;
}
//call it from somewhere:
double myNumber = getFloat();
```

Here, we call a method expecting a double to be returned, and the method returns a float. This code *will* compile. That is because the variable (myNumber) to which we assign the result of the getFloat() method can hold a double-sized number; the method returns a float, which is not a double, but which can fit into a double. The runtime automatically converts it for us. Everything is cool.

Starting Programs: The Main() Method

There is a method that is used as the starting place for all Java programs called the main method. It is so called in C++ and C# as well. When you start a Java program, you call the java command and pass it the name of the class containing the main method. The signature of this method is like this: public static void main(String[] args). Let's break that down.

The method is public so it can be called from anywhere. It is static, which means that the method can be called on the class itself instead of requiring a particular instance of the class on which to call it (which makes sense because at the start of the world you can't possibly have any objects yet). The return type of the method is void, which means that the method doesn't return anything (note that this is different in C++ where the main method returns an int). The method is called main and takes one parameter, which is an array of Strings. The array of Strings represents any arguments that you want to pass into your program when it starts up.

FRIDGE

It is possible to have more than one main method defined in your application. When you start the program using the java command, however, you must specify the class containing the main method you want to use as the entry point for this run of the application. It is not common to see multiple main methods within a single app.

Say you had a program that launched a rover to Mars. You might have numerous classes that make up the program, and have the main method that starts things off defined in a class called Launcher. Your program is designed to accept the names of the rover that you want to launch. You could invoke it like this on the command line: `java Launcher Spirit Opportunity`. The name of your program containing the main method is Launcher (so you know there is a public class called Launcher). Then, the `String` array that enters the main method as a parameter is an array of length 2, with "Spirit" as the value of the 0th cell and "Opportunity" as the value of the 1st cell.

Params.java

```
public class Params {

    public static void main(String[] args) {

        if (args.length == 2){
            String a = args[0];
            String b = args[1];

            System.out.println("Your first argument: " + a);
            System.out.println("Your second
            argument: " + b);
        } else {
            System.out.println("You must supply
            two arguments!");
        }

    }
}
```

The output of this class when called like this: `java Params Spirit Opportunity`, is

```
Your first argument: Spirit
Your second argument: Opportunity
```

The output when called like this: java Params (with no arguments), is

```
You must supply two arguments!
```

The main method does not explicitly throw any exceptions. If an exception occurs inside the main method, it is thrown out to the JVM, which then halts and prints a stack trace. The stack trace is the list of methods called up to the point where the exception was thrown, in descending order. You can put a try/catch block around the main method if you want, but it will have no effect.

Let's add a few regular old non-main methods to our Person class.

PersonWithMethods.java

```
public class PersonWithMethods {
 //define some members
 public String name;
 private boolean isJerk = false;

   //program starts running here
 public static void main(String[] args) {
        //create one person
      PersonWithMethods p1 = new
                  PersonWithMethods();
        //set the member's value with dot operator
      p1.name = "Aristotle";
        //call the method on the current object
      p1.setJerk(true);

        //create a different person object
      PersonWithMethods p2 = new
        //PersonWithMethods();
      p2.name = "Hegel";
        //pass in a boolean parameter to the method
      p2.setJerk(false);
```

```
          //find out what we know about them.

          //note the explicit call to the method //toString()

        System.out.println(p1.toString());

          //note the IMPLICIT call to toString!

          //both this and the previous statement

          //are equivalent

        System.out.println(p2);


  }


    //in this method, we set the value of the

    //isJerk member variable

  public void setJerk(boolean isJerk) {

    this.isJerk = isJerk;

  } //end method


  public String isJerk(){

    if (this.isJerk == true)

        return "a jerky jerk";

    else

        return "totally a non-jerk";

    }


  public String toString(){

    return name + " is " + isJerk();

  }

} //end class
```

The preceding code outputs the following:

```
Aristotle is a jerky jerk

Hegel is totally a non-jerk
```

We make two separate objects here, which we refer to as p1 and p2. Each object keeps its own data. They each have a different value for name and a different value for isJerk. The methods we call perform the same operation, but return different results based on the member data.

The toString() method is what is called an *override* of the method called toString() in the parent class of Person, which is java.lang.Object. See Chapter 12, "Inheritance," for more on that. I employ it here to demonstrate that the println() method can only print strings, so if a parameter passed to that method is not already a String type (such as the p2 object we pass to it here), the println method will implicitly convert it to a String. That is why we see that the same method is in fact called for System.out.print(p2) as for when we explicitly write it: System.out.println(p1.toString());.

Static Methods

A static method is one that is called on the class itself, as opposed to a method that is called on an instance of the class. The main method is a static method, which makes sense, cuz there cannot be any object instances before the program starts up and begins making objects! The way to make a static method is to declare it using the static keyword.

```
public static void something() { ... }
```

You can then call the method without needing a particular instance of the class. Static methods are used frequently in Java programming. They are used when they can always perform the same operation, regardless of the current program state. That's because static methods can't know anything about the current program state—they aren't called on object instances. They are said to be called on the class itself.

Consider this to help make the point: The methods of the java.lang.Math class are all static. It is easy to see why. They always do the exact same thing, regardless of what is happening in the world.

You can call a static method by using the dot operator after the class name, like this:

```
Math.random();
```

The preceding method returns a more-or-less random primitive double with a value greater than or equal to 0.0 and less than 1.0. You don't need an instance of an object to do that work, because the result will always be the same, no matter what the state of some hypothetical Math object might be. It's not even meaningful when you think about it. What would be the state of Math itself where it would make a difference what number the random method returned? There is no particular instance of the Math class that would make a difference to the sin method, which returns the trigonometric sine of an angle. That operation is always going to be done the same way. By the same reasoning, consider the following methods of the java.lang.Math class: abs() returns the absolute value of a number, atan() returns the arc tangent of an angle, asin() returns the arc sine of an angle, pow() returns the value of the first argument raised to the power of the second argument, and so on. There are a couple dozen utility methods in the Math class such as these. Check them out.

FRIDGE

You can read the Java 5.0 API documentation at http://java.sun.com/j2se/1.5.0/docs/api/. This is an invaluable resource that you will want to bookmark.

Let's now look at a few details regarding static methods.

Here's something that could throw you for a loop: you *can* call a static method on an object. However, doing so does not in any regard change its functionality. It is merely a convenience. It is preferable to call static methods on the name of the class to indicate that they are static methods. But that's a convention that will likely die as developers start adopting the new static import facility in Java 5.0 (discussed in Chapter 11, "Classes Reloaded"). Put a different way:

```
calling Person p = new Person(); p.myStaticMethod();
```

Is identical to

```
calling Person.myStaticMethod();
```

Here is a more complete example:

StaticStuff.java

```
public class StaticStuff {
   public static void main(String[] args) {
      System.out.println(myStaticMethod
          (Math.random()));
   }

   private static String myStaticMethod(double num){
      return "Your lucky lottery number is: " +
         (int)(num * 100);
   }
}
```

The output is something like:

```
Your lucky lottery number is: 63
```

The StaticStuff class features three static methods. This is a good example because it shows how to use static methods, but it also shows how you can nest method calls to simplify your program's look and streamline it.

FRIDGE

Stuff you can mark as static: methods, variables, and top-level nested classes.

It is certainly worth noting that static methods cannot access any non-static members or methods. That makes sense when you think about it. How could something defined on the blueprint level (the class level) know something about any particular instance of an object that later gets made with that blueprint? It can't.

The first static method, main(), is always static and starts the program. We don't pass any arguments into the program, so its String[] parameter is empty. We then call the method println() on the static member out of the System class. We call myStaticMethod(), passing in as a parameter the result of the third static method call—the one to Math.random(). We take the random result generated by the call to Math.random(), pass it into myStaticMethod(), which returns a String that contains the user's lucky number between 1 and 100. We cast the number, which comes in as a double to an int so that it chops off the decimal places, which we don't care about.

Final Methods

In Java, it is possible to extend the functionality of a class by subclassing. For example, we can decide that we want a more specific kind of Person, and then write a class that inherits Person variables and methods. Such a class might be called Programmer. A Programmer is a Person, with an address and a name and all, but a Programmer has other characteristics and things he knows how to do that are not common to all people. When you create a subclass, it is possible to override the methods in the superclass. That is, the Programmer class could choose to redefine a method that is implemented in the superclass. We did this earlier by having our implementation of the Person class override the toString method of the Object class.

If you don't want to allow a particular method to ever be redefined in a subclass, you declare it final. Consider the following method that might return the absolute value of an integer.

```
public final abs(int a);
```

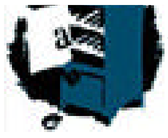
Here, we want to declare this method final, because it would be horrible if we allowed mathematical functions that should determine their results in an absolutely consistent manner to be overridden by a method in a subclass and change how an absolute is determined!

Note that classes, methods, and variables can all be declared final.

If a class is declared final, all of the methods within the class are implicitly final as well. Because the java.lang.Math class is declared final, none of the methods in it need to be declared final.

In the case of classes, it means that no other class can extend it. In the case of a method, as here, it means no subclass can override it. In the case of a variable, it means that its value cannot be reassigned after it has been assigned.

FRIDGE



There are certain things that you must always do when you declare a method. Methods must have a visibility level (default, public, protected, or private), a return type (such as void if the method returns nothing, or int, or Address, or String, and so on), a name, and a body. The body is the part in between the curly braces where you do your work.

You can declare that a method throws an exception in its signature using the throws keyword. If you care about that, it is discussed in Chapter 21.

Rules for Declaring Methods

We will discuss visibility modifiers in the next section. Classes, methods, and variables all have visibility. You can explicitly declare a method as having public, private, or protected visibility. If you do not explicitly indicate the visibility, the compiler will make your method (or class or member) to have default visibility. You don't write "default." For example, the following are all valid:

```
public void getAge() {}

void setAge() {}

private doWork(){}
```

Methods are defined by their signatures. A signature consists of

1. The name of the method
2. The number of parameters the method accepts
3. The type and order of each parameter

You cannot have two methods with the same signature in the same class; the runtime wouldn't know which one to call. You can have two methods with the same name—that is called overloading and is discussed later.

You cannot start a method name with a number or any weird characters like a carat or a % sign. Just don't be trying to do that. You can *start* a method name with an underscore, and the method name can contain numbers, they just can't come first.

Which of the following method declarations are valid or invalid?

```
public void _23() {}

void doinIt(String theThing) {}

int private HEY_SUCKER(){}

protected Address setaddress(theAddress) {}
```

The first is valid, because you can start a method name with an underscore.

The second is valid, because leaving out the visibility modifier is okay, and it will be set to "default" visibility.

The third method will not compile. Although the compiler doesn't care how you capitalize your methods, you cannot declare the return type (int) before the visibility modifier (private).

The last one will not compile because the parameter is not declared to have a type.

You can name the parameter that a method accepts according to the rules for naming any other variable.

In Java 5.0, you can also declare generic methods. These are an advanced topic, however, and if you're interested in using them, please check out the Generics topic in the companion book to this one, *More Java Garage*. Also, check out the Glossary in this book that contains some stuff about using generics.

That might be enough about those rules. Hmm. Yes, that's enough.

Conventions for Declaring Methods

Please follow these conventions for using methods. It will make your day go by with less unpleasantness. Methods are by convention named with an initial lowercase letter, and subsequent words capitalized. Like this:

```
public void someSortOfLongMethodName() {}
```

Method names should be descriptive. Although you are *allowed* to name your method x(), this is a real drag for everybody. Don't make the poor saps that inherit your code when you get rich and move to Tahiti have to deal with stuff like that. Have a heart for those poor souls left back at the office, hunched over their little P4s, in the near-dark, growing only slowly, like mushrooms.

Make your method names that are used just for returning a value get*ThingReturned*() like this: getAddress(). Such a method is called an accessor. If the value returned is a boolean value, you write is*Thing*() like this: isJerk(). Accessors are typically only this:

```
public int getAge() {

   return age;

}
```

Make your method names that are primarily used for setting the value of a member start with "set," like this: set*ThingToSet*().

A typical mutator looks like this:

```
public void setAge(int ageIn) {

  this.age = ageIn;

}
```

At the Java Academy for Super Nerds they brainwash you into saying "accessor" and "mutator" to refer to these kinds of methods. Nobody cool uses those terms though. All of the cool kids say "getter" and "setter." It somehow sounds less idiotic, which is why I think people probably use those instead.

The guidelines for naming parameters to methods is the same as for any regular variable name. Say you have a setter method. Call it setAge(), and it takes a parameter of type int like this: void setAge(int age). Typically, you will be setting the value of a class member. Name the method parameter after the class member, and name the method set*Member*(). So you have a method that ends up looking like this:

```
void setAge(int age) {

   this.age = age;

}
```

The keyword this is used to refer to the current instance. In VB, that's when you write "Me." It means "the current object," and in this context it helps us tell the difference between the member variable and the parameter. Some people slightly change the parameter name. My friend Vic does it by suffixing the word In to the variable.

```
void setAge(int ageIn) {

   age = ageIn;

}
```

Then, you don't have to use the this keyword. You can do that if you want. It might help to keep everything straight at first. We'll talk about this in a mo'.

Classes Have a Home (Package)

A package in Java is a namespace that allows classes to be identified. Packages allow you to have two different classes of the same name within the same application if the package name is different. The purpose of packages is to resolve naming conflicts.

To place your class in a package, use the package keyword. The package keyword acts as a logical namespace to separate different classes that go together. It allows you to put them in groups.

A package also corresponds to a physical location within a directory structure. This is different than in a language like C#, where the namespaces are purely virtual. In C#, you can put your class in any old directory and call the package whatever you want. Not so in Java. If you have a class called Kitty and you want to put it in a package called pets, the Java class file must physically reside in a folder called pets.

It follows that a class may reside in only one package.

If you explicitly put your class in a package, the package statement must be the first line of code in the source file that is not a comment. Comments can go first. But nothing else. Like this:

```
package net.javagarage;

public class MyClass {...}
```

The preceding statement means that there is a folder at the root of the application called net and it contains a folder named javagarage and that folder contains your class file. I can now write a class called MyClass and put it in the com.loveboat package, and the compiler and runtime will be able to tell them apart.

If you have more than one class in a source file, the package statement applies to every class in the source file.

If you do not explicitly place your class in a package, it will reside in the default package.

There is a convention for naming packages that just about everyone follows. It goes like this: Take your company's Web site and use its top-level domain as the top-level package. For example, in my case, my domain is called javagarage.net, so the TLD is net, so the top-level package is net. Then, put the domain name as the second highest-level package name. Don't put any classes at all in either of these packages. But now, after you're two deep, you usually name your application. Unless there is a subdomain that is relevant. Then, you can use the subdomain. For example, if I made an ecommerce site, I might put those classes in net.javagarage.store, and I could start defining the custom packages that I think I'll want in my app. Maybe products could have its own package, and checkout could have its own package. And like that.

For instance, I have classes in the net.javagarage.demo.inherit package that demonstrate using inheritance. I have classes in the net.javagarage.demo.classes package that demonstrate using classes. And so on. This is a good convention, based on the fact that a company's domain name is necessarily unique, so you know that your class will be unique. You just need to sort out your modules within each application after that, for organizational purposes.

It is important to take care with naming your packages, and organizing your classes within them. It is not uncommon for developers to revise and reallocate their package design multiple times throughout writing an application.

Classes Have Visibility

Up until now, we have put the keyword public in front of our class declaration, our method declarations, and our member declarations. We have mentioned different levels of visibility, but not what each of them means.

There are four levels of visibility in Java. They are listed here from least restrictive to most restrictive:

1. **public**. This is the most visible. It means that any class, in any package, can use that item. Classes, methods, and member variables can all be public. Classes declared public must be in a source file (the .java file) of the same name with matching case. Although you create multiple Java classes in the same source file, only one of them can be declared public, and that one must have the same name as the source file.

2. **protected**. Methods and variables that are declared protected are only visible to classes that are in the same package as this class, or are subclasses of this class. You cannot declare a class itself to be protected.

3. **default**. This is the level of access attributed automatically to a class, variable, or method that has no explicit access modifier on it. It means that this class can only be used by classes within the same package. The term default is not itself a keyword.

4. **private**. Methods and variables that are declared private are only visible inside the class in which they are declared. They cannot be seen (used) from any other class.

What level of access does the following method have?

```
int[] getMedicalRecords(Person p) { ... }
```

Default. Only classes in the same package can use this method.

If you try to access something that you aren't allowed to, the compiler will tell you with a message saying, "Can't access classname…". If you want to know more about classes, check out Chapter 11.

Chapter 10. FRIDGE: MMM-MMM LAMB CHOPS AND A MANHATTAN

I thought you might like my recipe for perfect lamb chops. It's a simple recipe that will make you very popular with whomever you're trying to be popular with.

Then we'll make a refreshing beverage to go with it.

When it comes to steaks, I have found that what really matters is the steaks you start with. With lamb chops, they all seem roughly the same. Which is just fine. Get enough to have four chops per person.

About half an hour before it's time to cook the chops, take them out of the fridge and put them into a ceramic dish. It doesn't need to be ceramic I guess, just not metal. Glass is good.

Drip extra virgin olive oil across each side. Generously mill rock salt and whole peppercorns over the chops.

Add a little finely crushed dill, which seems to be the key. Rub. Leave out at room temperature for 30 minutes or so.

Put the broiler on high. Cook for 4 minutes, then turn and cook for 3 minutes. Don't overcook them. This is assuming your chops are about 1 inch thick, which they should be. Serve over sticky basmati rice.

I know that this recipe is pretty simple. But sometimes, you know, that's the best thing.

Now the best thing to drink with lamb chops is a Manhattan. Everyone says so. So here is my recipe. I used to be a barkeep in New York City, which is the city where they force you to go to shows in the fake, sanitized Disneyland Times Square (ack, I'll take Shaft's 43rd and Broadway any day…) if you get their drinks wrong. So I know what I'm talking about.

Fill a rocks glass with ice. My grandmother only takes 5 solid cubes. You can do it that way if you like.

Add a jigger of bourbon. This is like the chops: getting good bourbon to start with is the key.

Add a solid dash of vermouth. Not too much now.

Now tease it with bitters.

Don't shake it or stir it or anything. None of those little cherries made by staunch men in lab coats bearing bottles of bleach. I heard when I was a kid that those things don't digest for seven years and I believe it. Anyway, that would be a bit frufru, don't you think? Just serve it with a warm smile and say, "Enjoy!" Just like my grandmother did. Believe me, your guests will just be delighted. You will make them really happy with this delicious, old-fashioned drink that you prepared lovingly.

Chapter 11. CLASSES RELOADED

DO OR DIE:

- Learn more about classes.
- Eat more, drink more, be merrier.

The Classes chapter in this book covered a lot of the basics about writing classes and making objects (instances of classes) and using variables and methods. We discussed what makes up a class, including a package, visibility, methods, and fields.

There was a lot of stuff in that last chapter that we had to cover before anything else could start making sense. But the Java class is the heart of the language. It is actually all that you ever do, write classes. So if we said everything about classes in one chapter, that chapter would be like one giant, super-long, book-length chapter, 900 hours long, like LoTR RotK, so action-packed that it doesn't even afford time to run for a refill on your mtn dew.

So, in this chapter we'll expand and deepen our understanding of classes, and demonstrate the true power of this space station. You might implement the things we cover here, such as enums, static imports, and constructors, depending on your design. We are going to do this in a kind of circular manner, coming back around like a boomerang and perhaps re-touching on things at a greater depth.

Using External Code in Your Programs: import

Almost every time you write a new class file, you will want to give that class the powers provided by some existing external file that lives in another package. You can do that in one of two ways:

1. Specify the fully qualified class name every time you reference the class.
2. Import the class or the package containing the class, and then simply reference the class using its name. You do this using the keyword import.

If you have used Microsoft .NET, you should be familiar with this concept. In C#, you might write the following:

```
using System;

class MyClass {

  public static void Main() {

    Console.WriteLine("smeagol was here");

    }

}
```

That's how you specify that you are going to use one or more of the classes in the System namespace in C#. C# namespace ~= Java package.

If you have used C#, you will find not just this section, but much of this book fairly easy going, as Microsoft has helped itself to very generous portions of Java's syntax in creating C#. You'll find that with a little tuck here and a little nip there, some of your Java code will compile directly to MSIL (Microsoft Intermediate Language used in .NET).

When writing code, the choices for using external classes boil down to the following options.

First, your code without import:

```
net.javagarage.monsters.Vampire vampire = new

             net.javagarage.monsters.Vampire();
```

Second, your code with import:

```
import net.javagarage.monsters.Vampire

...

Vampire vampire = new Vampire();
```

Note that you can import all of the classes in a package using the common wildcard character *. To import all the classes and interfaces in the monsters package, write

```
import net.javagarage.monsters.*;
```

That way, your code can do this all day long (assuming your "monsters" package has classes named Mummy and Werewolf in it):

```
Vampire vampire = new Vampire();

Mummy mommy = new Mummy();

Werewolf werewolf = new Werewolf();
```

Look, ma—no fully qualified class names!

Keep this in mind: trying to decide whether to import all of the classes in a package using * or importing only the class from that package that you need is not a very interesting dilemma. In fact, it is no dilemma at all. They both perform the same and result in the same class file size. When you import all of the classes in a package using *, the Java compiler does not do what you ask. Instead, it determines what classes in that package your code actually references, and imports only those. So don't worry about the efficiency of code that uses the * wildcard to import. My recommendation is to use the wildcard if you are lazy, or if you need more than one class from the same package. If you have a number of less-experienced developers on your team (or you yourself are easily confused), you might want to explicitly import each class, even when they're in the same package, to make the code more readable.

FRIDGE

Many IDEs will help you organize your imports so that you don't have to think about it too much. In Eclipse, for example, you can right-click anywhere on your source code file and choose Organize Imports—this kind of thing will help keep your code clear and clean.

Automatically Imported Classes

One last thing about packages. You might have noticed that you can just write, compile, and execute this code:

```
package com.monsters;

public class Vampire {

public static void main(String [] args) {

String music = "punk";

System.out.println("Tony Blair's fave music is "+ music);

}

}
```

A question arises here. Why don't we have to import String? That is a Java class made available in the API, and it is in the java.lang package. You don't have to import classes in java.lang package: all of the classes in that package are always automatically imported for you. That means that you could write this to get the same result as in the preceding:

```
java.lang.String music = "punk";
```

Static Imports

Static imports are new with Java 5.0. In previous examples, we have used the class name as a prefix to calling the static methods of the Math class, such as Math.random(). This is okay, but it is also unnecessarily verbose.

To facilitate ease of use in other classes with the static methods and static variables, you can add the static keyword to the import directive and import them all so that you don't have to use the class name as a prefix to calling them anymore.

So, you have a choice between writing this (which is how we used to have to do it)

```
public class MathClient {

   public void x() {

      double d = Math.random();

      //...blah blah

   }

}
```

and writing this:

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

public class MathClient {

   public void x() {

      double d = random(); //look! no "Math."!

      //...blah blah

   }

}
```

Using static imports is meant to help clear up code that becomes overburdened by numerous, repeated calls to constants that clutter up your code.

StaticTestConflict.java

```
package net.javagarage.statimp;


import static net.statimp.MyConstants.*;

import static net.statimp.MyOtherConstants.*;


public class StaticTestConflict {


   public static void main(String[] args){


      System.out.println(SOME_PROPERTY);


   }

}


class MyConstants {


   static String SOME_PROPERTY = "MyConstants value";

}


class MyOtherConstants {
```

```
class MyOtherConstants {

   static String SOME_PROPERTY = "MyOtherConstants value";

}
```

Note that it is okay to declare multiple classes in the same source file as long as only one of them is declared public. But you can't compile this code if you do not fully qualify the reference to the property with the class name as we have done. Here's what the compiler says about it.

```
reference to SOME_PROPERTY is ambiguous

both variable SOME_PROPERTY in net.statimp.MyConstants and

variable SOME_PROPERTY in net.statimp.MyOtherConstants match

System.out.println(SOME_PROPERTY);


1 error
```

You might think that you would not be able to use static imports to import two separate methods with the same name in different packages. For example, the Integer compareTo() method and the Byte compareTo() method both have the same names. However, they have different signatures: the compareTo() method from Integer accepts an Integer, and the method with the same name in Byte accepts a byte. Because of this, your namespaces stay separate, and the following lines of code are legal:

```
import static Integer.compareTo();

import static Byte.compareTo();
```

The runtime will be able to sort them out.

So. Yeah. That's all I want to say about imports right now. I hope that that answers all of your questions about this stuff. If, after this, you have more questions about packages and imports, you might consider getting a Zoloft script; as my cousin August says, "Better living through chemistry…" Otherwise, try taking up a hobby, like gardening, or hot rod racing.

Constructing Objects

When you define a class, you can define one or more constructors for it. A constructor provides a way of creating (constructing) an instance of its class.

When you use the new keyword to create an object, you are calling the constructor of that class whose argument list matches the arguments provided in the call.

The following code calls the default (no-arg) constructor for the JButton class:

```
JButton btnSave = new JButton();
```

Constructors are special methods that must match the names of the classes in which they are defined, and which have no return type. They are used to control how objects are created, and often, to initialize fields to the values specified in the argument list.

When you run the preceding code, the runtime determines if there is sufficient memory space to create the object and its data. If there is not enough space, the runtime tries to find any candidates for garbage collection. The garbage collector destroys items with no references, because they can no longer be used in a program's life. If there is still not enough space, an OutOfMemoryError is thrown.

```
package net.javagarage;

public Employee {

  private String name;


  public Employee(String nameIn) {

     this.name = nameIn;

  }

}
```

FRIDGE

We haven't discussed superclasses (a.k.a. parent classes) yet. But for now, just know that the java object tree is hierarchical. A class can extend another class, which means that it can inherit its non-private data and functionality. Every Java class in the world automatically inherits from java.lang.object. That means that any class you write will be able to do what Java objects do. It means that any class you write will inherit the methods defined in that class, such as equals, hashCode, wait, and notify.

The code in bold represents the constructor. It has no return type and its name matches exactly the name of the class.

It accepts one argument, a String. The field name is then initialized to the value of the parameter nameIn. The superclass constructor with a matching signature is called. Then the constructor exits and the object is ready for use. Let's see how this works.

Constructors.java

```
package net.javagarage.demo.classes;

/**
 * Demos use of constructors, and how
 * they interact with superclass constructors.
 * @author eben hewitt
 */
public class Constructors {

    public Constructors() {
        System.out.println("Superclass constructor!");
    }

    public static void main(String[] args) {
        //what does this print?
        //in what order?
        SubClass sa = new SubClass();
    }
}

class SubClass extends Constructors {
    //overriding the no-arg constructor
    public SubClass() {
        System.out.println("Subclass constructor!");
    }
}
```

So how many constructors are called in the preceding code? Here is a misleading clue: the code outputs the following:

```
Superclass constructor!

Subclass constructor!
```

The answer is three, because every class in Java is a subclass of java.lang.object—so the matching constructor in the object class gets called too. When you construct an object, all matching constructors are called down the hierarchy.

No-Arg Constructors and Overloaded Constructors

The constructor called in the preceding code is called the no-arg constructor. This means that it doesn't accept any arguments. You can explicitly write a no-arg constructor in your class definition. If you don't, a no-arg constructor is provided for you that does nothing but create an empty object of the specified type. That constructor is called the default constructor.

The default constructor is *not* provided if you provide any constructor definition of your own. If you want to provide a no-arg constructor, you have to define it yourself. Here's how:

```
package net.javagarage;

public Employee {

  private String name;

  public Employee() {}

  public Employee(String name) {

     this.name = name;

  }

}
```

The Employee class here defines a public, no-arg constructor. Because it provides a constructor that accepts a String argument, which it assigns to the name field, it must explicitly define a no-arg constructor if you want users of your Employee class to ever be able to create Employees without also initializing the name field concurrently.

After an object has been created, its variables must be initialized (remember, that's the second purpose of constructors).

Although you can create a JButton using the no-argument constructor, as we did earlier, the JButton class defines several other constructors (remember, classes can define as many constructors as they want—or none). One of the other constructors takes a String argument that initializes the field that represents the text printed on the button. So, calling the following constructor:

```
JButton btnSave = new JButton("Save");
```

kills two birds with one stone.

You don't have to call the setText(String s) method now, because the constructor that accepts a String argument has already done that for you. If there is not a constructor defined that matches the argument list that you try to use to create an object, a NoClassDefFoundError is thrown.

To overload a constructor, just make another one that has a different argument list than any other constructor in that class, just like overloading a regular method. There are many different overloaded constructors defined in the Java API. The String class, for example, has 14 constructors.

You can create a constructor with a variable-length argument list, as shown here.

VarArgConstructor.java

```
package net.javagarage.demo;

  //demonstrates ability to create

  //constructor with variable length arguments


public class VarArgConstructor {


  //vararg constructor

  public VarArgConstructor(Integer...ar){

    System.out.println("Object created!");


    for (Integer i : ar) {

        System.out.println(i);

    }

  }


  public VarArgConstructor(Integer i, Integer j){

    System.out.println("Constructor with two Ints

      called!");


  }


  public static void main(String...args) {

    VarArgConstructor va = new

            VarArgConstructor(1,2);

  }


`
}
```

Note that the correct constructor is called in the preceding class. If you replace the object creation statement with one that passes three or five ints into the constructor, the vararg constructor will be called. Because we pass in two ints, "Constructor with two Ints called!" is printed.

Private Constructors

You can define a private constructor as well. But just like in methods, the private access modifier indicates that that constructor can only be called from within the class itself. This is often used to purposefully disallow clients of the class to create instances of the class. For more about how and why to do this, visit the Singleton pattern, discussed in the "Design Patterns" chapter in *More Java Garage*.

Calling Other Constructors with this

The Java keyword this is used to refer to the current instance of a class. It operates like the VB Me keyword. But there is another use for this. Writing overloaded constructors could easily mean repeating the same code over and over. To save you from having to do that, you can call this as you would a method, and the invocation will refer to a constructor defined in your class that matches the parameter list.

If you invoke another constructor within the same class using this, it must be the very first thing you do in your constructor.

ThisConstructor.java

```
package net.javagarage.demo;

public class ThisConstructor {

  private boolean isInitialized;
  private String name;
  private int theID;

public ThisConstructor() {
  isInitialized = true;
  System.out.println("no-arg called");
}

public ThisConstructor(int theID) {
  super();
  this.theID = theID;

}
public ThisConstructor(int theID, String name) {
  //call constructor that takes int
  this(theID);
  //set the string value
  this.name = name;

  System.out.println("int, String arg called");
}

public static void main(String[] ar) {
  ThisConstructor ts = new ThisConstructor(555,
                         "Mr. Ed");

  }
}
```

The output is

```
no-arg called
int, String arg called
```

Static Constructors

This is kind of a misnomer. You don't get a static constructor by adding the static keyword to a regular constructor. You just write the keyword static and then curly braces. Inside the curly braces, write the code you want to execute.

Static constructors inside the class that contains the main method are executed even before the main method, when the class is initially loaded in the classloader.

You can define multiple static constructors. *They will be executed in the order in which they appear in source code*. The following code shows the order in which the constructors are executed.

Static Constructors.java

```
package net.javagarage.demo.classes;

/**

 * Demo order in which constructors are called.

 * @author eben hewitt

 */

public class StaticConstructor {


static {

  System.out.println("Inside static constructor ONE.");

}


//regular no-arg constructor

public StaticConstructor(){

  System.out.println("Inside regular constructor.");

}


public static void main(String[] args) {

  StaticConstructor c = new StaticConstructor();

  System.out.println("Inside main.");

}


static {

  System.out.println("Inside static constructor TWO.");

}

}
```

And here's the output:

```
Inside static constructor ONE.

Inside static constructor TWO.

Inside regular constructor.

Inside main.
```

Static constructors are rarely used in regular business application programming. They are used for loading native code, discussed in detail in *More Java Garage*.

Using Enums

FRIDGE

The code in this book was compiled and executed using the JDK 5.0 alpha and beta releases, and it is possible that there are changes between these and the final release. If you want to switch between different compilers, you can use the `- source 1.5` switch to the `javac` command.

Enums are new with Java 5.0. Enum is short for enumeration, and enums are useful when you want to define values that are meaningful to the user (in this case, other programmers) that can be switched against. The Enum is defined in the java.lang package.

When to Use an Enum

Programmers may be familiar with the enum, which programmers may be familiar with from C, C++, C#, or Pascal. A simple enum looks like this:

```
public enum Season { winter, spring, summer, fall }
```

An enum is an object that defines a list of acceptable values from which the caller can choose. It defines zero or more members, which are the named constants of the enum type. No two enum members can have the same name.

Java programmers have heretofore had to roll their own poor-man's enums by using constant- style names with `int` values, like this:

```
public interface Season {

  static winter = 0;

  static spring = 1; //etc..

}
```

Every field declaration in the body of an interface is implicitly public, static, and final. So the interface is used sometimes for constatnt declarations. If you want to make those constants available to your class, you implement the Season interface.

There are problems with having to do so, however. The first is that you need to write a lot of code to handle the situation, and rolling your own usually means that you are using an interface, as we do here, for something that it isn't intended for. This can cause clutter and confusion in your program.

Note that you can declare a public enum within a public class source file (remember that you can't declare more than one public class in the same source file).

You can also declare a public enum in its own .java file and compile it. Here is the enum replacement for our earlier interface:

Season.java

```
public enum Season { winter, spring, summer, fall }
```

That's all we have to do. The values for each of the seasons will be automatically assigned a numeric value, starting with 0, and incrementing one for each remaining enum value.

Let's look at the easiest way to implement the new enum construct.

EnumDemo.java

```
package net.javagarage.enums;

/*
We can loop over the values we put into the enum
using the values() method.
Note that the enum Seasons is compiled into a
separate unit, called EnumDemo$Seasons.class
*/
public class EnumDemo {

    /*declare the enum and add values to it. note that, like in C#, we don't use a ; to
➥   end this statement and we use commas to separate the values */

    private enum Seasons { winter, spring,
     summer, fall }

    //list the values
    public static void main(String[] args) {
        for (Seasons s : Seasons.values()){
            System.out.println(s);
        }
    }

}
```

Running the preceding code outputs the following, as you would expect:

```
winter
spring
summer
fall
```

Switching Against Enums

The following code demonstrates how to switch against the values in an enum, which is what enums are commonly used to do:

```
package net.javagarage.enums;
/*
File: EnumSwitch.java
Purpose: show how to switch against the values in an enum.
*/

public class EnumSwitch {

    private enum Color { red, blue, green }

    //list the values
    public static void main(String[] args) {
        //refer to the qualified value
        doIt(Color.red);

    }
    /*note that you switch against the UNQUALIFIED name. that is, "case Color.red:" is a
➥  compiler error */

    private static void doIt(Color c){

    switch (c) {
    case red:
        System.out.println("value is " + Color.red);
        break;
    case green:
        System.out.println("value is " + Color.green);
        break;
    case blue:
        System.out.println("value is : " + Color.blue);
        break;
    default :
        System.out.println("default");
    }
    }

}
```

Adding Fields and Methods to Enums

You can add fields and methods to enums, just like you would a regular class. The following shows that you can use an enum much like you would a class. You can define static or non-static fields and methods, and define constructors. You can define different levels of visibility, as with a class.

EnumDemo.java

```
package net.javagarage.enums;

/*
File: EnumDemo.java
Purpose: show how to use an enum that also defines its own fields and methods
*/
```

```
public class EnumWithMethods {

//declare the enum and add values to it.

public enum Season {

    winter, spring, summer, fall;

    private final static String location = "Phoenix";

    public static Season getBest(){
        if (location.equals("Phoenix"))
            return winter;
        else
            return summer;

    }

    public static void main(String[] args) {

    System.out.println(Season.getBest());
    }
}
```

Notice one limitation of enums that is not often present in other features of the Java language, placement of the values list is limited to immediately following the enum declaration. That is, you'd see a compilation error if you switched the following two lines of code in the preceding example:

```
private final static String location = "Phoenix"; //wrong!

winter, spring, summer, fall;
```

The enum of seasons comes following the field here, which is illegal. The compiler would let you know an identifier is expected.

Enum Constructors

You can create an enum with a constructor as well, just like you would in a regular class. The following code shows how.

```
package net.javagarage.enums;

public class EnumConstructor {

    public static void main(String[] a) {

        //call our enum using the values method
        for (Temp t : Temp.values())
            System.out.println(t + " is : " + t.getValue());
    }

    //make the enum
    public enum Temp {
        absoluteZero(-459), freezing(32),
        boiling(212), paperBurns(451);

    //constructor here
    Temp(int value) {
        this.value = value;
    }

    //regular field—but make it final,
    //since that is the point, to make constants
    private final int value;

    //regular get method
    public int getValue() {
    return value;
    }

    }
}
```

The output is the following:

```
absoluteZero is : -459

freezing is : 32

boiling is : 212

paperBurns is : 451
```

Now. Just because enums share the capability to define methods and fields and constructors like regular classes does not mean that defining and using all of those gadgets is a good idea. If you want that kind of functionality, you probably should write a class. Enums are very useful as a shortcut for initializing int variables in a list of constants when the values are meaningful only within the context of the other values. They're perfect for that. They're popular in many programming languages. But if you need a class, make a class, man, not an enum.

Methods Allow Variable-Length Parameter Lists

Varargs. Yes, that is what passes for a word these days (did you ever read Ray Bradbury's "The Sound of Thunder"?). The vararg is a thing that they have in C# and other languages and it means that you stomp your foot and say, "This method has a bunch of parameters and *I just don't know how many parameters a client might want to pass it, all right?*" It means "variable length argument list."

This is an easy concept to start working with. The following class defines a method that accepts a vararg, and invokes the same method passing it first several arguments and then fewer arguments. The arguments come in as an array.

```
package net.javagarage.varargs;

/*

Demos how to use variable length argument lists.

*/


public class DemoVarargs {


public static void main(String...ar) {


  DemoVarargs demo = new DemoVarargs();


  //call the method

  System.out.println(demo.min(9,6,7,4,6,5,2));

  System.out.println(demo.min(5,1,8,6));


}


/*define the method that can accept any number

of arguments. do so using the ... syntax.

Just calculates the smallest numeric value in the argument list.

*/

public int min(Integer...args) {

    boolean first = true;

    Integer min = args[0];


    for (Integer i : args) {

      if (first) {

        min = i;
```

```
            first = false;

         } else if (min.compareTo(i) > 0) {

            min = i;

         }

      }

   return min;

}

}
```

The output is

```
2

1
```

Varargs are shortcuts that prevent the programmer from having to define the method parameter as an array. You'll notice that because the main method that starts Java applications is defined to accept an array of String objects as arguments, it can also be called using this syntax, like this:

```
public static void main(String...args) {...}
```

But that is not because the main method wants an array because it just really enjoys having arrays around to hang out with. The String[] argument is required because we don't know at compile time how many arguments the user will pass into the java command.

The benefit to varargs is that your intentions are explicit (always good), and that you don't have to worry about falling off the end of the array. When combined with the for each loop as earlier, varargs are very easy to use.

So obviously, you want to accept an array as a parameter if you're going to be doing something with arrays. Use varargs only to indicate a variable length parameter list.

Note that varargs are new to SDK 5.0, so you need to compile for that version of Java to use them.

Wrapper Classes

You can't call a method on a Java primitive, such as an int or a double. And sometimes that's a problem because you would like to be able to. Sometimes, you need to hold a list of these kinds of values, and you want to use the classes in the Java Collections Framework; the problem here is that none of those data structures, such as ArrayList, Hashtable, or Vector, will hold primitive types. They only hold objects.

So what's a girl to do? Use the wrapper classes that come with the Java API. There is a class in the java.lang package that corresponds to each of the primitive types. Table 11-1 shows these classes.

The Primitive Wrapper Classes

Wrapper	Primitive
Boolean	boolean
Byte	byte
Short	short
Character	char
Integer	int
Long	long
Float	float
Double	double

The preceding wrapper classes allow their corresponding primitive types to be used as objects. They provide the following methods that make working with them easy and fun. The wrapper classes share these methods in general. Let's take one wrapper, Boolean, as an example, from which to extrapolate.

First, the constructors you can use to make Boolean wrappers:

- public Boolean(String b) Creates a Boolean object representing the value true if the argument is "true" (ignoring case).
- public Boolean(boolean b) Creates a Boolean object corresponding to the value of the passed Boolean primitive.
- boolean booleanValue() Call this method on an object of type Boolean when you want to get its value as a primitive. The corresponding methods in the other classes include intValue(), doubleValue(), and so forth.
- public int compareTo() Compares the values of two wrappers of the same type. It returns 0 if this object represents the same primitive value as the argument.
- public static boolean parseBoolean(String s) Takes a String argument and returns its Boolean value.
- static Boolean valueOf(boolean b) Returns an instance of the Boolean object corresponding to the boolean argument. This method is the sister of booleanValue(), which you use to get a primitive from a wrapper—the xValueOf() methods get a wrapper from a primitive. This method is overloaded to accept a String argument as well.

Boolean has three static fields: TRUE and FALSE, which represent their primitive counterparts, and TYPE, which represents the primitive type boolean.

I only show the Boolean wrapper class here as an example, because the other wrapper classes work very similarly, and share similar methods. The Integer wrapper class, for example, features an intValue method, and the Long wrapper class features a longValue method.

The wrappers are necessary when you need to use your primitive values with some API that only will accept working with objects. However, because that kind of functionality is so fequently needed in object-oriented programming, a new feature was added to Java 5.0; autoboxing, which we discuss next.

Autoboxing

As wonderful as the primitive wrappers are, they often mean we end up creating a lot of objects (say with valueOf(int i)), passing them into a list, and then putting them back into their primitive values again with intValue()) when we drag them out of the list.

Autoboxing is a feature new to Java 1.5 that takes care of this for us. Here's how it works in a nutshell. Create a primitive type. Pass it as an argument to a method that expects an Object. It magically works. The JVM converts the primitive to its corresponding wrapper on-the-fly for you.

Autoboxing wouldn't be complete without auto-unboxing. As you might guess, this means that when you ask for a primitive directly from an Object wrapper, ye shall receive.

Because this concept is fairly easy to understand and appreciate, let's go straight to the videotape.

AutoboxAssigment.java

```
package net.javagarage.autobox;


public class AutoboxAssignment {


  public static void main(String[] args) {

    //whoa! primitive reference to object

    int bottlesOfBeer = new Integer(99);


    //the other way 'round

    Float temperature = 98.6f;


  }

}
```

This simple test shows that we can move effortlessly between primitives and their wrapper classes.

FRIDGE

This is cool. It does not mean that you can start calling methods on 'primitives' as you can in some languages I won't mention (okay, C#). The old int.parse() trick won't play in this town….

This is not license to do whatever wacky thing you want, however. You can't call methods on a primitive and expect it to know what you're trying to do.

For example, say we try to call one of the Double wrapper class methods on a double primitive:

```
double d = 45D;
d.isNaN(); //Wrong!!!
```

The compiler sez

```
double cannot be dereferenced
```

Just so you know it when you see it, let's try another test, a little more complicated.

AutoboxDemo.java

```
package net.javagarage.autobox;

import java.util.*;

public class AutoboxDemo {

public static void main(String...args) {

   /**Make a map that holds String keys and Double values. Collection types can only
➥  hold objects, not primitives. */
    Map<String,Double> weatherForecast = new

TreeMap<String,Double>();

    weatherForecast.put("Monday", 65);
    weatherForecast.put("Tuesday", 68);
    weatherForecast.put("Wednesday", 70);


    System.out.println(weatherForecast);


}

}
```

This class creates a Map, which holds key/value pairs, and passes in double primitive values. Even though the Map only holds objects, they are converted on-the-fly and we don't have to deal with object creation for each one of those doubles ourselves.

This functionality is convenient. I could show you the old, yucky way we had to do this, but I will spare you the gory details. They're just too awful…

If you enjoy torture, please refer to Chapter 18, "Casting and Type Conversions," for a little more on this.

The Class Named Class

There is a class called Class that represents the possibility of making and loading classes in the Java Programming Language. You can make instances of the Class class for the classes and interfaces in a running Java application.

The primitive Java types and the keyword void are represented as Class types.

Class has no public constructor. That is, you cannot write this: `Class clazz = new Class();`. You can get a Class object for a given class in three different ways.

First, you can use the `getClass()` method of java.lang.Object. If you have an instance of `Object` called `obj`, you can write the following:

```
Class clazz = obj.getClass();
```

Second, you can use the class literal, like this:

```
Class clazz = java.lang.String.class;
```

Third, you can get it by passing a String containing the fully qualified name of the class of which you want a class instance, like this:

```
try {
    clazz = Class.forName("java.lang.String");
} catch (ClassNotFoundException e) {
}
```

Getting the Package of a Class

You can also get the package that a class is in by calling the `getPackage()` method on the Class object:

```
Class clazz = java.lang.String.class;

Package package = clazz.getPackage();

          //represents a package

String pName = package.getName();

          // java.lang
```

If you write a class without declaring a package for it, it will be created in the default package, which has no name. If you call getPackage() on the Class object of such a class, it returns null. It also returns null for a primitive type class, or an array. Like this:

```
Class clazz = SomeClass.class;

packg = clazz.getPackage();          // null


packg = char.class.getPackage();     // null

packg = char[].class.getPackage();   // null
```

Getting an Object's Superclass

You can get an object's superclass by calling the getSuperclass() method on the Class object.

```
Object obj = new String();

Class supr = obj.getClass().getSuperclass();

              // java.lang.Object

// The superclass of java.lang.Object is null

obj = new Object();

supr = obj.getClass().getSuperclass();     // null
```

Listing the Interfaces That a Class Implements

We will find out about interfaces later. For now, know that they are a contract containing methods that you can force a class to implement. Many classes in the Java API implement interfaces. You can list all of the interfaces implemented by a class with the following code:

```
Class cls = java.lang.String.class;

Class[] interfaces = cls.getInterfaces();
```

This stores the Class objects of all of the interfaces String implements in an array.

Note that if the reference type of your item is an interface, the call to

```
intfc.getClass.getSuperclass();
```

returns the object's superclass.

<shameless-plug>This is enough for now about the very cool java.lang.Reflect package. If you are interested in finding out about classloading, dynamic proxy creation, how to dynamically invoke a method, and even how to change the programmed visibility of a member using the reflection API, you'll have to get the sequel to this book, the inventively titled *More Java Garage*. </shameless-plug>

Now that we are on more solid ground with the fundamental structures in Java, it will get a lot easier now to start working with more code, seeing more useful examples, and starting to do some fairly exciting things.

The Java world gets markedly more complex at this point, and that sophistication will pay off for you in power.

We won't waste any time. The next chapter introduces one of the three cardinal precepts of Object-Oriented programming: for the sake of clarity, they are Faith, Hope, and Inheritance….

Okay, they are Encapsulation, Polymorphism, and Inheritance.

That somehow sounds less dramatic, though. Onwards and upwards.

PREV < Day Day Up > NEXT

Chapter 12. INHERITANCE

DO OR DIE:

- Learn about Inheritance.
- Have some food.

PsychoMan PsychoMan PsychoMan

Mr Hand (furious): Just what do you think you're doing?

Jeff: Learning about inheritance, and having some food.

The thing is this: classes can inherit functionality from other classes. It is kind of important. So we're going to talk about it for a while.

Getting Your Inheritance

I bet that "getting your inheritance" pun has been made a thousand times. I think I will leave it in anyway and try to pass it off to you like, "Yeah, we both know that that dumb joke has been made so many times. But it's not simply some somnambulist tech writer digging into the dregs of his already shallow literary toolbox and pulling out yet another overused bad pun. Rather, in my deft hands, it's another brilliant and hilarious example of a master of 'high camp' confidently practicing his craft."

I'm not sure I like where this is going. Let me start all over. Okay.

Java classes all participate in inheritance, whether they like it or not, and whether they know it or not.

No! Somehow that comes off as angry—too angry. I'm not *angry* about inheritance. Geez. Try again:

Inheritance in Java, as in life, is when you get something for nothing. [This is [Following is a Continuation Paragraph...] too preachy, too absent, and drippy. Note to self: Don't liken things to "life" as if "life" is somewhere else and you know something about it. This can be annoying, despite the incredible insight. Just tone it down a bit, okay? Is that so hard? Gawd.]

Every Java class has an implicit superclass: java.lang.Object. That is to say, the class called Object in the java.lang package (which you recall is automatically imported for you into every class) is the parent of every other class in Java that has ever been, or will ever be, made. Any person we make (that is, any object of the class Person we instantiate) is an object.

It makes sense that Person needs to know something other than just how to be a Person—it needs to know how to live inside Javaland. Every object of every class needs to know that information. It needs to be able to be constructed in the first place, and it needs to be able to participate in threads and such. That behavior is necessarily shared by every Java class. So for that reason, java.lang.Object is the superclass of every other Java class, which we can call the subclasses of java.lang.Object.

There are a few terms that we can use to mean the same thing:

Superclass = A class that is higher up in the hierarchical inheritance tree.

Parent class = Superclass.

Subclass = A class lower in the hierarchical inheritance tree than another class.

Child class = Subclass

So Person is a subclass of Object. Person is a child class of Object. Object is the superclass of Person. Object is the parent class of Person. We'll use these terms interchangeably to try to keep you awake.

If you want to, you can declare that your class will inherit the data (members) and functionality (methods) from one other class. You do so using the extends keyword.

But wait! When we declared the Person class, we did it like this:

```
public Person { ... }
```

Where's the part about Object? Where's this "extends" you speak of? They are implicit. The following code is equivalent to the preceding code:

```
public Person extends java.lang.Object { ... }
```

Classes can have only one immediate parent class. The following is *not legal*:

```
public Person extends LivingCreature, Homosapiens {

... }

//illegal! busted!
```

Multiple inheritance is a thing allowed in C++, and the Java language designers thought that it made code very confusing and difficult to debug and maintain. So they decided not to allow it at all. Internet people hanging around in forums complain on occasion that disallowing multiple inheritance makes it hard to do certain things in the language, and that it was a bad decision. I say, "Ah, hog wash." Java allows interfaces, which you can read about in the exciting chapter 17, intriguingly named, "Interfaces." You can implement multiple interfaces, and that kind of solves that problem, which I don't necessarily recognize as a problem, because in my experience, if you've got a good design, you won't feel any need to inherit from more than one class. In fact, I'll play this card: If you find yourself at your desk thinking, "Man, I can't do what I need to this way. I wish I could extend more than one class…," you should rethink your design. Really. The issue of only allowing single inheritance is a non-issue.

FRIDGE

Remember that children cannot see their parent's private members. In psychiatric circles such an episode is referred to as "The Freudian Scene," and can have long-lasting effects on the child. :)

To help you think about this kind of thing, make sure to look at the "IS-A Thing" and "HAS-A Thing" sections later in this chapter. Meanwhile, back at the ranch…Because every class extends Object, they each get to call all of the *public* methods that Object implements, as if they were Object itself.

Object has the following *public* methods:

```
toString(), equals(), hashCode(), getClass(),

notify(), notifyAll(), wait().
```

That means that you can call those methods on every Person object, even though they aren't defined in the Person class. You get that functionality for free.

Here are some things to remember about inheritance. If a member has protected visibility, any subclass of that member's class can access it because protected = package + children. A subclass can only see the protected members through inheritance. Private members are not visible to subclasses.

Methods Can Override Superclass Methods

Sometimes, what you get from a parent isn't exactly what you wanted or needed. Perhaps you can relate to that. :(. In that case, you might want to make a new implementation of the method you inherited from them. In Java, this is called overriding. In the real world I believe it's called therapy.

You can implement a method that is defined in a superclass however you want to. Nevertheless, the method signatures must match when you do this. If the signatures don't match, you aren't overriding anything; you're simply making a new method that has the same name, but different signature. If the signatures don't match, the compiler will be able to sort out which method you mean to call when. But if a subclass overrides a method that is implemented in a superclass, the runtime will call the subclass method on all instances of the subclass type. It just slides it right in there and replaces it. It makes the superclass method invisible to the subclass. Here's an easy example to demonstrate.

Let's take our Person class that we already have and extend it to make a Programmer class. We'll then extend Programmer with JavaProgrammer. The code, as always, is heavily commented so you can see what's happening line by line.

Programmer.java

```
package net.javagarage.demo.inherit;

/*

 * We'll just do some dumb printlns.

 * Notice that Person is in another package,

 * so we either need to import it or simply write

 * out the fully qualified name like this in the

 * extends statement.

 */

public class Programmer extends

               net.javagarage.demo.classes.Person {

     String onlineName;


     void writeCode(String someCode){

          System.out.println("The code: " +

               someCode);

     }


     void slackOff(long duration){

          System.out.println(

               "Now slacking in the hallways " +

          "for the next " + duration +

               " minutes.");

     }

     void screwAroundOnInternet(String site){

          System.out.println("Now screwing around

               at " + site);
          at " + site);
     }

     void portProgram(String fromPlatform, String
               toPlatform){
          System.out.println(
               "Spending the rest of the year " +
          "moving the app from " + fromPlatform +
               " to " + toPlatform);
     }
}
```

JavaProgrammer.java

```
package net.javagarage.demo.inherit;

/*
 Define a subclass of programmer that is a specific
type of programmer that can do all the same things
as Programmer, and other, more specific things.
 */
public class JavaProgrammer extends Programmer {

     //this method is the override of portProgram
     //defined in the superclass Programmer
     void portProgram(String from, String to){
          System.out.println("No port necessary: " +
           "the same Java code runs on both " +
          from + " and " + to);
     }

     //now define some new method that
     //only Java programmers do
     void goToJavaOneConference(){
          System.out.println("Going to JavaOne...");
```

```
    }
}
```

OverrideDemo.java

```
package net.javagarage.demo.inherit;

/**
 * All this does is make an object of type
 * JavaProgrammer and demos how the methods it
 * inherits (and the one it overrides) work
 */
public class OverrideDemo{

    public static void main(String[] args) {

        //create new instance of JavaProgrammer
        //the JavaProgrammer jp = new
                JavaProgrammer();
        //inherited member from Person
        jp.name = "Jane";
        //inherited from Programmer
        jp.onlineName = "Veruca Salt";

        //defined in Programmer
        jp.writeCode("<blink>Buy now!</blink>");

        //inherited method from Programmer
        jp.screwAroundOnInternet("javagarage.net");

        //overridden method defined in both
        //Programmer and JavaProgrammer
        //but this object's reference type is
        //JavaProgrammer, so guess which method
        //gets called...jp.portProgram

        //("Windows", "Linux");


        //also from Programmer

        jp.slackOff(147);


        //just a regular method defined only

        //in JavaProgrammer.

        //well, my work here is done...

        jp.goToJavaOneConference();

    }

}
```

Here is the output of running the class.

```
The code: <blink>Buy now!</blink>

Now screwing around at javagarage.net

No port necessary: the same Java code runs on both

Windows and Linux

Now slacking in the hallways for the next 147 minutes.

Going to JavaOne...
```

This code clearly shows how things work when one class extends another and overrides its methods. I hope it clearly shows that anyway, because I think that's enough talking about it.

IS-A Thing

This is more of a design issue, not something that is proper to the Java programming language. But design is important. I think that elsewhere in this book I am going to suggest that the design is the implementation. Think about that for a moment. It washes over you like a mantra: the design is the implementation….

Java code is written as a collection of classes. Some programs are 1 class, many are between 100 and 1000 classes, and some are more than that. When you start dealing with a lot of classes, you need to think carefully about how to divide up your classes. One way to do that is the subject of this topic: inheritance. If class B inherits from class A, we say that B "is an" A. Say you have a class called Vehicle, which is the superclass of Car. Car inherits from Vehicle. Car extends Vehicle (those mean the same thing). A Car is a Vehicle. Car is a specific kind of Vehicle. A Cat IS A Feline. A Feline IS A Animal (excuse my grammar). You know that Animal is a superclass that Feline extends, and Cat extends Feline.

Thinking about IS-A relationships will help you get your inheritance issues straightened out, help you communicate with your analysts and other programmers, and help you make a well-designed program. Don't change the channel before reading HAS-A Thing. They kind of go together.

HAS-A Thing

The other kind of relationships in Java dare I say programming is the HAS-A relationship. HAS-A means the properties of this class. A Car HAS-A engine. A Car HAS-A collection of tires. A Car has a maximumNumberOfPassengers. A Car HAS-A type, such as sedan, compact, sporty, or whatever. You get the idea.

When designing your applications, say to yourself, "What is this thing? What properties does it have?" After you know the answers to the IS-A and HAS-A questions, you should first make sure that some of the properties don't actually belong to a superclass, and then you have your basic class skeleton. Like this:

```
public class Car extends Vehicle {

    private Engine engine;

    private Tire[] tires;

    private short maxNumberOfPassengers;

    private String carType;

    //... getter and setter methods

}
```

Now all you need to do is figure out what a Car does in the context of your application, and you're off.

Classes Can Ensure That No Class Inherits from Them

FRIDGE

The only modifiers allowed in a class declaration are public, final, and abstract. A class cannot be both final and abstract.

The final keyword can be applied to classes, methods, and variables, and means roughly the same thing for each.

Sometimes, you want to make sure that no class ever will be able to inherit functionality from your class. Because of polymorphism, which you can read about in Chapter 16, "Abstract Classes," and others, situations may arise when you want to ensure that the improper substitution of one class for another does not break your application or cause a security problem.

In these cases, you can declare a class to be final. You do so in the class declaration, like this:

```
public final class String {...}
```

That's right. The class java.lang.String is a final class. The designers of the Java language were prescient enough to see that allowing Strings to be subclassed would lead to security risks. We'll examine this in greater depth in Chapter 13, "Strings," but for now, it is enough to understand the following possible scenario: you run an application that gains access to the local file system by substituting a subclass of String with a rewritten value (containing a path) that the user might not knowingly provide. Because Strings are immutable (cannot be modified), and because they are final, this sort of thing does not happen in Javaland.

Unless you are writing fairly low-level code that you anticipate causing a possible security breach, you may not find many occasions to declare your classes final.

In the following chapter we switch gears a bit, and discover how to use Strings in some depth. Strings are among the most common programming elements you will use, and there are deceptively subtle aspects to their behavior. So we switch out of talking about classes in general and how they behave, and move into talking about a specific and very popular class that comes with the Java API: that lovable sequence of characters known as java.lang.String.

PREV < Day Day Up > NEXT

Chapter 13. STRINGS

DO OR DIE:

- Don't tell me what to do

A String is a set of zero or more characters. A String can be empty, or its reference can be null, or it can contain the entire text of *King Lear*. Strings are objects in the java.lang package, and are used so frequently that they are the only class in Java with their own special constructor syntax. In this topic, we put the moves on a few String objects and see how they like it.

Creating String Objects

Strings are objects. Their superclass is java.lang.Object (they inherit all of the properties and methods defined in the Object class, and then define some of their own). As we know, everything in Java is an object (except for the primitive types such as int, boolean, and char), and they all have corresponding object types that are often called "wrappers".

If you want to be a big nerd and make everyone hate you, create a String like you would a regular object:

```
String s = new String("Sucker!!!");
```

If, for some reason, you do not want to be a big gigantic nerd and force everyone to hate you, use the following simpler, more common, and more efficient syntax to create a String:

```
String s = "All the cool kids do it";
```

If Java provides us with this different constructor syntax, what is the problem with creating it the longer and more familiar way?

The easy reason is that no one does it, so it is unexpected behavior. And although you are lovely and singular and creativity is your strong suit, the declaration and initialization of your Strings is not the place to make your artistic statement. In other words, predictability is very good. Be predictable. The poor schmoes who inherit our code will be just a little less poor. Let's look at the possibly more important technical reasons in the following sections.

One String for the Price of Two

The following is called a String literal:

```
"This is da creeda of Jacques Derreeda";
```

It's a bunch of characters between double quotes. The compiler will automatically create a new String object for every literal it stumbles across in your program. Sound weird? Think about what happens when the new keyword is invoked? Java does what it's told and creates a new object on the heap. But what does it do when you use the new keyword *and* give it a String literal in the same statement?

Consider. You're the virtual machine. You're supposed to create a new object when you see new and you're supposed to create a new object when you see a String literal. So, you create *two* objects, just as you're told to do.

As you might imagine, specifying only the String literal, instead of using the new keyword, is far more efficient, because only the object literal is created.

That begs the question, "Do I have two Strings then, each with the same value?" The answer, as you might not expect, is, "No, you don't." See the "String References and Immutability" section to find out why.

Converting a Char Array into a String

Despite the performance penalty described previously, sometimes it is necessary to create Strings using the new keyword. And that's just fine. I'm not getting all weird about it. I'm just saying don't do it if you don't have to. It's perfect, for instance, when you have an array of characters, and you need to get that array into a String object. In these cases, you must use the constructor. Here's what to do:

```
char[] myCharArray = {'g','a','r','a','g','e'};

String myString = new String(myCharArray);

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

Note that you cannot convert directly to a String. That is, you *cannot* do this:

```
String myString = myCharArray; //won't compile
```

(See the "Strings and Security" section to find out why.)

Note that the String class contains a couple of different ways to do this; the other way also takes an int letting you specify the offset.

Converting a ByteArray into a String

There is also a String constructor that accepts a byte array. Say you use the New IO library introduced in Java 1.4 to read in the contents of a file to a ByteBuffer, like this:

```
    Row row = null;

  rows = Collections.synchronizedList

(new ArrayList(numberOfRecords));


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

    row = new Row(metaData);


    Charset charset = Charset.forName("US-ASCII")
```

```
        CharsetDecoder decoder = charset.newDecoder();


        byte isDeleted = bb.get();

        row.setIsDeleted(isDeleted);


        ByteBuffer bIn =

            ByteBuffer.allocate(LENGTH_OF_LINE);

        CharBuffer cb =

            CharBuffer.allocate(LENGTH_OF_LINE);

        cb.rewind();

        charset.newDecoder().decode(bb,cb,false);

        cb.rewind();

        String s = cb.toString();


        synchronized(datarows){

          //put record into list that the app will use

              rows.add(convert(row, s));

         }

       }
```

There is obviously code here that we haven't discussed yet. I'm just providing you with a code snippet that will read in an ASCII file, which only requires seven bits to render a character (not 16 like Unicode) into an array of bytes so that you can make a String out of it. If I didn't include the code, then it might be hard to see where you would get a byte array in the first place. But the above code example reads each line in the file, calling a separate method called convert as it uses the data to create a Row object.

Once we have that byte array, we can covert each line into a String. Like this:

```
Column c = new Column();

c.setSomeShortData(bb.getShort());


byte[] arrFieldName = new byte[c.getLenFieldName()];

bb.get(arrFieldName, 0, arrFieldName.length);

try {

    String data = new String(arrFieldName,"US-ASCI");

    c.setData(data);
} catch (UnsupportedEncodingException e) {

    e.printStackTrace();

}
```

If this all seems like too much right now, definitely do not sweat it. Onto the next. But if later you discover you need some byte-array-readin-string-makin' code, there you have it.

String References and Immutability

Something that is immutable means that it cannot be changed. One immutable thing in this universe is a Java String, which means, after it is assigned a value, that value never changes—*ever*. To which you might reply, "Oh yeah? Well what about this?" and then whip out some ninja code like this:

```
String s = "kitty";

s = s.concat(" cat");

System.out.println(s); //prints kitty cat
```

The concat method of the String class takes the string value of the object on which you call the method and concatenates (adds to the end) the literal parameter you pass the method. So that looks like all kinds of mutability, right?

Here's what really happens:

When we call concat, the virtual machine takes the value of the String s, and hangs onto it. Because it can't make any other character string except for "kitty" be the value of s, *it creates a second, new string object, assigns it the value of " cat", and then creates a third String object with the value of "kitty cat" (both literals mushed together), and then reassigns the object reference of s to that new object!* Pretty shifty. In other words, there are now *three* String objects, and s refers to the third one. Technically (this is at least trying to be a technical book), s didn't change its string value; it only changed its object reference, which gives the appearance of changing that String object's value.

So how many references are there?

Only one. s.

But—hey! What happened to the "cat" object? Isn't it on the heap? Yes. But nothing refers to it, and so it is lost forever, with no way for anyone to access it.

In the preceding example, we took a String and changed its characters by calling the concat method on it. That code looks like this:

```
s = s.concat(" cat");
```

Notice that we assign s again here. That is, we have a String s, and then assign it to the String object created when the concat() method call returns, which is how we get the new value "kitty cat". Can you tell what this code will output?

```
String s = "kitty";

s.concat(" cat");

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

If you haven't fallen asleep yet, good show. If you were paying careful attention, you might have guessed that this code prints "kitty" because we never reassign the object reference of s to point to the result of the method call. So it points to the same value it did when it was initialized. This code makes a String object for "kitty", and the concat() method makes two more (" cat" and "kitty cat"). No one ever asks the JVM about it, and now no one ever can.

Let's look at another example that is a little more complicated.

```
1. String s1 = "first";

2. String s2 = "second";

3. s1 = s2;

4. s2 = "new value for s2";

5. System.out.println("s1: " + s1);

6. System.out.println("s2: " + s2);
```

What output will this code produce?

The first two lines are very straightforward: they create two separate String objects with different values. At line 3, the reference for s1 is reassigned to point to s2, and there's the trick. Will line 5 print "second" or "new value for s2"?

If Strings are immutable, and s2 is given a new value on line 4, what happens? A new object is created and s2 is reassigned to that new object's address. s1 still points to the previous address for s1, and "second" is printed. Line 6 prints "new value for s2" as you would expect.

Why go to all of the trouble to make Strings immutable? One benefit is speed. The runtime can reuse a literal from the pool of Strings without having to create a new one, and because object creation is one of the more expensive endeavors the runtime will undertake, it's good to avoid it if possible. There is another consideration. Security.

Strings and Security

A Java String inherits from Object, but no class inherits from String, or ever will. That's because String is marked final, which means that it is the last of its kind. No class can ever inherit from a class marked with the final keyword in its declaration. I'll prove it. Try typing this in Eclipse (or whatever IDE or vi or what have you):

```
public class StringSubclass extends String { }
```

Here's what the compiler tells you:

```
The type StringSubclass cannot subclass the final

    class String

StringSubclass.
```

So what does that have to do with security? Well, our inability to subclass String means that we cannot get a value from a host and rewrite that reference using our custom (and malicious) subclass, substituting the runtime type for ours (which otherwise would be legal).

Luckily, in day-to-day business programming, I have yet to find myself at my keyboard thinking, "Damn! It will never work! If only I could subclass String…" Of course, I'm not in the business of writing viruses…

There is the related issue of converting other objects to Strings. Some things that you think might directly convert to Strings (such as an array of characters) do not. You must use the Java keyword new, and pass it the char array. Now why on earth would that be necessary? All of these hoops they make us jump through! Let's take a step back.

What happens when you do this:

```
String x = "This guy";

String y = "Some other guy";

x = y;
```

Two String objects are created: x and y. Then, the value of the y String is assigned to x (meaning, now x has the value of y). Big deal. Well, in order to see the big deal, let's recall that the value of an object is its reference! Sure, it might look to us like the value of s String is the characters we write, but that is a human-centric view (which I learned in graduate school is indelicate). Think about it from the virtual machine's perspective. We just replaced the object reference for x to point to y's address in memory. If we could assign arbitrary references (or at least practically arbitrary types, such as byte arrays) to Strings, we could easily create a buffer overflow, ruin the integrity of the application, and gain access to local resources that we were not supposed to have access to. This small march of progress is the origin of many viruses that exploit less strongly typed languages. I won't mention any names.

String Assignments

The StringBuffer will change the object's value without using the assignment operator. This is slightly different than you might expect given how String works. Here's an example:

```
String s = "Elvis";

s + " Presley";

//s = Elvis


StringBuffer sb = "Elvis";

sb.append(" Presley");

//sb = Elvis Presley
```

To make the String in the preceding code operate like the StringBuffer, you need to assign s to the *result* of the operation, like this:

```
String s = "Elvis";

s = s + " Presley";
```

Recall that the shortcut for primitives works for Strings here as well:

```
String s = "Elvis";

s += " Presley"; //same difference
```

You can read about StringBuffers a little later in this topic.

Doing Stuff with Strings

Now that it is clearer how Strings assignments and concatenation work, we will turn our attention to performing miscellaneous but frequently useful String operations.

Convert Different Variables to Strings

In this section we'll look at a couple of ways to convert different kinds of things into Strings.

Using valueOf()

Often you need to convert certain types to Strings in order to pass them to a method that accepts only a String. You can do this using the valueOf() method of the String class.

```
System.out.println(String.valueOf(Math.PI));

//prints 3.141592653589793
```

There are several valueOf() methods defined for the String class, each of which takes different parameters (which means that the valueOf() method is *overloaded*). All of them are static. That means that they are called on the class, not on any particular instance of the class. That is, you don't need to make an object, you can just write: String.valueOf(Math.PI); as in the preceding code. There is a separate valueOf() method for each of the primitive types, for char arrays, and one for Object. Math.PI returns a primitive double in case you were curious.

FRIDGE

Psst. Remember this from Chapter 9? When you pass an object of any type into System.out.println(), that method automatically calls the toString() method of that object. So, if you don't override the toString() method, you might get something less than useful printed out.

Using toString()

A second way to convert objects to Strings is by calling the toString() method on the object. An obvious difference from the valueOf() method is that for toString() you need to have an object, whereas valueOf() is static. Here's an example.

```
Object o = new Object();

String s = o.toString();

String d = new Double(12).toString();

System.out.println("s: " + s + " D: " + d);

//prints s: java.lang.Object@194df86 D: 12.0
```

We create a generic object and call its toString() method. Then, we create an object of type Double, and pass a primitive double to its constructor. Its toString() method returns "12.0". What is the difference between the two?

Object's toString() method prints the name of the class, followed by the @ sign, followed by a hexadecimal representation of the hashcode for the object, which means it prints a value equivalent to the following:

```
getClass().getName() + '@' + Integer.toHexString(hashCode());
```

You can do something more meaningful by implementing toString() in your own way (overriding it). Like this:

```
public String toString() {

  return "Some unique-ish field:" + this.someField;

}
```

Character Data Encoding

All Java character Strings are rendered as 16-bit Unicode. Unicode is a standard specifically created for computer processing of character data. Its purpose is to provide a consistent manner in which to encode character data, so that users throughout the world, writing in multiple languages, can share a single system.

The problem that Unicode solves is the problem introduced by ASCII character encoding, which represents our Latin alphabet beautifully, but nothing else. This is no longer an acceptable mode for character data exchange in the Internet age. ASCII Latin characters can be represented by only 8 bits each, but have a very limited range; Unicode represents all of the characters from every major written language in the world. It also includes many mathematical symbols, ideographs, technical marks, and other items. Doing so requires two bytes per character. One of the features of Java that propelled it to popularity in the early 1990s was its enforcing of the use of Unicode throughout the language in order to embrace the global economy.

FRIDGE

Go to www.unicode.org and click on the link labeled Code Charts. This will let you select from all of the graphematic symbols that Unicode supports, including Greek, bi-directional Hebrew, currency and safety symbols, and more. You can read the symbol charts to find out what code you need to call to print certain characters.

Most of the time you don't have to worry about this. But if you need to present data written in Arabic, or present special symbols, you can write Unicode directly by specifying its symbol.

All Unicode characters map to a single hexadecimal number. You can specify a Unicode literal using the escape backslash followed by a u. For example, \u0123 prints a ! (bang). Here is an example using some currency symbols.

```
String s = new String("Euro: " + "\u20AC");

s = s.concat("\nDollar: " + "\u0024");

s = s.concat("\nYen: " + "\u00A5");

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

This outputs the following:

```
Euro: €

Dollar: $

Yen: ¥
```

Note that many Unicode characters may not be printable on your particular system. Note, too, that any given symbol is actually implemented by a font vendor, and so there may be great variation between its appearance between systems.

You can read more about character encoding and its use in I/O operations in Chapter 22, "File Input/Output."

Converting a Byte Array to a String

Java provides a few different constructors in the String class to accommodate creating String objects from byte arrays. This is particularly useful when interacting with legacy systems or native code that provide you only with byte arrays for character data.

The following code snippet converts text between Unicode and UTF-8 using a byte array:

```
try {
    //go from Unicode to UTF-8
    String string = "abc\u5639\u563b";
    byte[] utf8 = string.getBytes("UTF-8");

    //go from UTF-8 to Unicode
    string = new String(utf8, "UTF-8");
  } catch (UnsupportedEncodingException e) {
    //handle. we'll talk about exceptions later
}
```

Comparing Strings

The following code, which demos how to compare strings in different ways, is pretty straightforward. There is one weird thing, however. The compareTo() method of the String class returns an int representing the result of the operation: it returns a 0 if the strings compared are equal, a value less than 0 if the first string is lexigraphically less than the string argument, and a value greater than 0 if the first string is lexigraphically greater than the string argument.

CompareStrings.java

```
package net.javagarage.demo.String;

public class CompareStrings {

  public static void main(String[] args) {
     String littleName = "elvis";
     String bigName = "Elvis";

       //is the character content of the string
       //this returns false, because of
```

```
        //case-sensitivity
      boolean isIdentical = littleName.equals(bigName);

        //check without case-sensitivity:
        //returns true!
      boolean b = littleName.equalsIgnoreCase(bigName);

        //check order of two strings
        //lowercase FOLLOWS uppercase
      int i = littleName.compareTo(bigName);
      if (i < 0) {
        // big comes before little
      } else if (i > 0) {

      } else {
          //they are the same
       }
   }
}
```

This is the big thing to beware of when comparing Strings: don't use the == operator unless you really mean it! That operator compares the object references of two Strings—not whether they have the same characters in them. Sort of.

Remember that there is a String pool, and creation of a String that already exists should reuse that reference.

The == operator is used to compare primitive values. You can't use the equals method on primitives because you can't use any method on primitives. When used with reference variables (objects, like Strings), the result of x == y is a Boolean value that is the result of the following test: do these two reference variables refer to the same object—that is, the same space in memory? Said a different way, are the bit patterns of x and y identical?

You use the equals() method to compare two objects to determine if their meaning is equivalent. With Strings, it has been decided that their meanings are equivalent if the characters they store are the same—that is, "J.B. Lenoir" is meaningfully equivalent to "J.B. Lenoir". It has been decided because the String class is final. If you have a way you think is an improvement over this method of comparing String meanings, you're out of luck in Javaland. However, you can go nuts and enjoy overriding the equals() method for your own objects. This is a good idea, by the way.

To determine if two Strings are not equal, you can use s1 != s2 or (! s1.equals(s2)).

Here is something that will come in handy. Remember that you will get a runtime exception if you try to call a method on a null reference:

```
String s1 = null;
if (s1.equals("")) ...//no!
```

This code will blow up because you can't call a method on a null reference: the runtime will throw a NullPointerException. The way to fix it is to get in the habit of checking the other way around: pass the possibly null value into the equals()

method as its parameter. Like this:

```
String s1 = null;

if ("".equals(s1)) ....//okay
```

That's a good way to test if a String is empty.

PREV < Day Day Up > NEXT

Useful String Methods

In keeping with the habit I'm trying to develop, of focusing on code that you can run and change and mess around with, and get you in the habit of doing instead of reading, let's just plunk down some code that shows off some of the useful methods of the java.lang.String class.

StringDemo.java

```
package net.javagarage.demo.String;

/** <p>

 * Demonstrates how to use Strings to your advantage.

 * </p>

 * @author HewittE

 * @since 1.2

 * @version 1.0

 * @see

 **/

public class StringDemo {


private static void print(String s){

System.out.println(s);

}

private static void print(char c){

System.out.println(c);

}

private static void print(int i){

System.out.println(i);

}

private static void print(boolean b){

System.out.println(b);

}


public static void main(String[] args) {

StringDemo sd = new StringDemo();


//make a String to work with
```

```
String s1 = "Java Rockstar";

//return the character at a given position
print(s1.charAt(2)); //prints "v"

String s2 = "zzzzz";
print(s1.concat(s2)); //prints "Java Rockstarzzzzz"
print(s2.concat(s1)); //prints zzzzzJava Rockstar

print(s1.endsWith("r")); //prints "true"
print(s2.endsWith("y")); //prints "false"
print(s1.equals("Java Rockstar")); //prints "true"
print(s1.equalsIgnoreCase("java rockstar"));
//prints "true"
print(s1.equals(s2)); //prints "false"
print("muscle car".equals("cool")); //prints "false"

//prints java.lang.String
System.out.println(s2.getClass().getName());

//does string match this regular expression?
//this regex checks if String is any digit
print("7".matches("[0-9]")); //prints "true"
print("whoa!".matches("[0-9]")); //prints "false"

print(s2.length()); //prints "5"

double d = 21D;
//this is a static method, so we use the class name
//(String) and then call the method on the class
System.out.println(String.valueOf(d));
//prints "21.0"
//you can do this with floats, ints, chars, etc.

char[] chars = {'m','a','m','b','o'};
String charString = String.valueOf(chars);
print(charString); //prints "mambo"

//removes trailing and leading whitespace
```

```
s1 = " s p a m ";

print(s1); //prints "s p a m"



//check the pool

print(s2.intern());



}

}
```

The weirdo here is definitely intern(). That method checks the String pool and sees if there is already a String containing a meaningfully equivalent String, and if so, returns it. If not, the String is added to the pool and a new reference is returned.

Here is the output of running the preceding code:

```
v

Java Rockstarzzzzz

zzzzzJava Rockstar

true

false

true

true

false

false

java.lang.String

true

false

5

21.0

mambo

s p a m

zzzzz
```

So to recap, the JVM maintains a pool of unique Strings, which is empty when the JVM starts. As you create Strings, they are added to the pool. All Strings, including constant expressions with a String value, are interned. According to the Java 5.0 documentation, "for any two strings s and t, s.intern() == t.intern() is true if and only if s.equals(t) is true." You don't typically need to call intern yourself.

PREV < Day Day Up > NEXT

Better Efficiency with StringBuffers

The StringBuffer class is useful for making strings that need to be concatenated, appended to, or otherwise tangled with more than two or three times. Because regular String objects are immutable, changing a String actually creates a whole new String, which is an expensive operation. To make String-changing more efficient, we use StringBuffer. If you've done any work in C#, you'll recognize this as System.Text.StringBuilder.

There are generally only two things you do with a StringBuffer: append text to it, and then, when you're all done appending, call the toString() method to use it as a String. Like so:

```
package net.javagarage.demo.String;

public class StringBufferDemo {

public static void main(String[] args) {
  //do it manually
    StringBuffer sb = new StringBuffer("Sideshow");
    sb.append(" Bob");
    sb.append(" Frames Krusty");
    System.out.println(sb);

    System.out.println(doLoop());
}

    private static String doLoop(){
        StringBuffer sb = new StringBuffer();
        int i = 10;
        while(i > 0){
            sb.append(i + ",");
            i—;
        }
     return sb.toString();
    }
}
```

For this sort of operation, StringBuffers provide better efficiency. You often need to do this sort of thing when building SQL query strings to send to a database or when building file paths. It might be important to you to know that StringBuffer does not override the equals() method.

Calls to StringBuffer append operations can be chained. That is, the return type of a call to append is the newly appended StringBuffer object. So we could rewrite the code above in the following manner:

```
StringBuffer sb = new StringBuffer("Sideshow");

sb.append(" Bob").append(" Frames Krusty");

//...
```

This has the same result as the more verbose code in the previous example.

Chapter 14. ARRAYS

DO OR DIE:

- Learn how arrays work in Java. If you think you have a better objective for this topic (after all, a topic on arrays, duh), I'd be happy to hear about it (good luck, buddy [as if])

The array is a standard feature of programming languages. An array is a collection of objects or primitive values, each of which is of the same type, and each of which is associated with an ordinal place in the collection. This ordinal place is called an *index*. As a feature of the Java language, they offer quick storing and retrieval of data, in large part because their size is always known. That is, after you create an array to store 150 items, that particular array can never be resized.

Creating Arrays

You can declare a new array variable, just like you would any other member variable (like this)

```
String[] args;
```

The square brackets mean "array." An array in Java is an object, which means that you initialize an array with the keyword new. Like this:

```
String[] args = new String[10];
```

This creates a new array object, called args, with 10 buckets for putting Strings into. You can fill an array with whatever kind of thing you want: Object, int, boolean, Address, char, another array, and so on.

The first element of a Java array has an index of 0. To reference the element in the second bucket of an array named args, type this:

```
args[1]
```

FRIDGE

Remember that arrays are the size that you make them, no bigger, and that the first element's index of an array is 0. That means that if your array holds 10 elements, you can happily access elements with index 0 through 9 all day long. Those are the 10 you get. Try accessing the element at index 10, and you'll get an ArrayIndexOutOfBoundsException, meaning the element you tried to access doesn't exist. Try accessing the element at index -56. Same thing. This can be confusing, so try not to be confused by it. Remember that the first element of an array is 0, so the last element of an array is *array length minus 1*!

That's because the array starts at 0, dammit!

There is a shorthand to declaring and initializing and populating a Java array all with one line of code. You have to already know what all your values are, but this frequently comes in handy.

```
String[] cats = {"Noodle", "Doodle", "Little Mister"};
```

Now you have an array with three buckets, containing one each of the stated values. We can get a value out of an array by referencing the array name, followed by square brackets containing the index of the element you want. Like this:

```
cats[2]; //returns "Little Mister"
```

The following code demonstrates different ways of creating regular and multidimensional arrays in Java.

Creating Arrays.java

```
package net.javagarage.demo.arrays;

/**<p>

 * Demonstrates how to create arrays

 * and refer to their elements.

 * </p>

 * @author eben hewitt

 **/

public class CreatingArrays {


public static void main(String[] args) {

//1. instantiate new array with 5 cells

int[] myInts = new int[5];


//put a value into the 2nd cell

myInts[1] = 1;
```

```
System.out.println("1st cell: " + myInts[0] +

". 2nd cell: " + myInts[1]);

/*

 * prints 1st cell: 0. 2nd cell: 1

 * notice that means that the first cell

 * (the 0th cell was initialized to the default

 * value for its type (int)

 **/


//2. now show another way to make them

//notice that spaces don't matter

//populate array with the original members of //Motorhead

String [] myStringArray = {"Lemmy","Larry","Lucas"};


//this is one of the few times in Java you

//can use ; after }


//3.

//you can also do this (create without initializing)

Object[] objects;


//note that you can declare it by putting the [] in a

//different location. the following is LEGAL:

Object myThings[];


/* but no one does it that way, and i'd avoid it.

 * sometimes convention is just easier for everyone.

 * it is more natural to say "Object array called

 * myThings" as you read—it makes sure the complete

 * type is in the declaration, not the identifier.

 * After all, your value on the heap is an Array

 * object called 'myThings' that holds Objects,

 * not a "myThings array".

 */


 //try to access a value:

 //System.out.println(myThings[0]);
```

```
//COMPILER ERROR! Not initialized!


//create a 2-dimensional array with 10 elements

//in each dimension:

int[][] square = new int[10][10];


//create a "ragged" array, with each dimension

//holding varying elements

int[][][][][] ragged = new int[2][4][3][10][5];




}

}
```

One thing to note about creating arrays (and, perhaps more to the point, the loose language that is sometimes used, even by Java book authors, as in the preceding code) is that there is no such a thing as a real multidimensional array in Java. Every Java array has only one dimension. You can create an array that holds another array, however, thereby achieving two-dimensional array representational capability.

About array values: You can insert into an array any value that can be automatically promoted to the declared type. For example, say you have an array of ints. You can put a byte or a short in there, because an int is 32 bits, and those are smaller than 32 bits, and so can snugly fit inside an int. No problem. What you can't do is add a long to that array because a long is 64 bits, which is too big to squeeze in there. You can't put an object in there either.

PREV < Day Day Up > NEXT

Using Arrays

Even if your array is filled with primitive ints, every array in Java is an object, and is treated as such. Arrays are objects. So there are a number of methods that you can put to work to get value out of them. It also means that you should comply with object assignment and equivalency tests, as well as state and local laws.

You can think of two-dimensional arrays as holding a row and column number, in that order. For example, take Table 14-1.

A Two Dimensional Array

	0	1	2
0	**78**	—	46
1	**92**	78	92
2	**98**	81	66
3	**71**	56	89
4	**55**	43	45

This table represents a two-dimensional array that holds 5 rows and 3 columns. You would create such an array like this:

```
int[][] grades = new int[5][3];
```

The element at position [3][2] is 89. The element at position [0][0] is 78. The element at position [4][2] is 45. The element at [0][1] is empty. The element at position [4][3] doesn't exist, and trying to access it gets you an ArrayIndexOutOfBoundsException.

```
grades[0][1] = 90 ;

// puts 90 into row 0, column 1.

grades[2][ 2]++ ;

//increments value at row 2, column 2 by 1
```

Copying Array Elements

If you want to do this, you can simply assign a new array to the old array, and then you have your data in the new array, right? Well, yes, but this is shared data. If somebody updates that data, both arrays get the update, and this might not be desirable. So the API designers were thoughtful enough to allow us to do this via the aforementioned System.arraycopy() method.

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

//...

System.arraycopy(sourceArray, 0, destArray, 0,

        min(sourceArray.length, destArray.length));
```

Remember that length is a field of the Array class; it isn't length() or getLength(). The preceding code uses the static import feature new to Java 5.0, so it won't compile on earlier versions. But that's easily fixed.

The following class shows how to use two-dimensional arrays and how to use the System.arraycopy() method.

UsingArrays.java

```
package net.javagarage.arrays;


/**<p>

 * Shows how to use arrays after they are created.

 * </p>

 * @author eben hewitt

 * @see

 **/

   //import all static methods of the Math class as

   //convenience

import static java.lang.Math . *;


public class UsingArrays {


public static void main(String[] args) {

//create an instance (object) of this class

UsingArrays demo = new UsingArrays();


//now call the methods defined in this class

//on the object reference

demo.useSquareArray();

demo.copy();
```

```
          [illegible]

}

/**
 *Creates an array of 100 elements (10 * 10 = 100),
 *and populates them with values.
 */
private void useSquareArray(){
int[][] square = new int[10][10];

//loop over to populate.

for (int i = 0; i < 10; i++){
for (int j = 0; j < 10; j++){
square[i][j] = j+1;
System.out.print(square[i][j]);
}
System.out.println();
}
}



private void copy(){
char[] sourceArray = {'s','e','c','r','e','t'};
char[] destArray = new char[6];
//the arraycopy method copies the contents and not
//merely the references
System.arraycopy(sourceArray, 0, destArray, 0,
    min(sourceArray.length, destArray.length));

System.out.println(destArray[2]);
}



}
```

The result is as follows:

```
12345678910

12345678910

12345678910

12345678910

12345678910

12345678910

12345678910

12345678910

12345678910

12345678910

c
```

Shifting Array Elements

You can shift all of the elements in an array over to the right or the left with a little simple code. This uses the arraycopy() method, which we just saw.

```
//say you've got an array called array

//shift all elements right by one

System.arraycopy(array, 0, array, 1, array.length-1);


//shift all elements left by one

System.arraycopy(array, 1, array, 0, array.length-1);
```

The arraycopy() method (which should have been called arrayCopy to follow Sun's own naming conventions) takes the following parameters:

src	The source array
srcPos	Starting position in the source array
destination	The destination array
destStartPos	Starting position in the destination data
length	The number of array elements to be copied

Note that you have to know what you're doing when you use this method. It throws NullPointerException, IndexOutOfBounds-Exception, or ArrayStoreException, if the source or destination arrays are null, or if the system can't squeeze your big, honkin' data into that little array. Whew.

Using Stacks

Stacks are like Pez dispensers. Stacks are very similar to arrays, and so we'll discuss them here. Many internal data structures are stored as stacks within Java. The stack operates on a last in, first out basis. You can create your own stack, or you can use an instance of java.util.Stack.

The java.util.Stack class extends java.util.Vector, which is a type of collection. The two main operations you use with a stack are pop() and push(). You can also peek() to see the item on the top of the stack. Finally, you can search the stack for an item, and you can determine if the stack is empty.

UsingStacks.java

```
package net.javagarage.arrays;

import java.util.Stack;

/**
 * <p>Does stuff with java.util.Stack
 * @author eben hewitt
 */
public class UsingStacks {

public static void main(String[] args) {
Stack<Integer> stack = new Stack<Integer>();

stack.push(41);
stack.push(42);
stack.push(43);

//get first in line
System.out.println("First: " + stack.peek());
//prints 43

//is stack empty?
System.out.println("is empty? " + stack.empty()); //false
System.out.println("search: " + stack.search(42));

//search returns 2—this is 1-based!!!



}

}
```

Note that you need to compile this code with the 5.0 compiler because it takes advantage of the new generics autoboxing feature (we're adding int primitives to the stack, which is declared to accept only Integer objects—they get automatically wrapped).

PREV < Day Day Up > NEXT

ArrayLists and toArray

Say that you have an ArrayList (a java.util.ArrayList is a type of List that works like an array, but is automatically resizable). ArrayLists hold java.lang.Objects.

```
ArrayList myObjArrayList = new ArrayList();

//put stuff into myObjArrayList without generics,

//you can retrieve them only as type Object


//initialize a new array that has as many elements

//as the ArrayList has:

MyObject myObjArray[] = new MyObject[myObjArrayList.size()];


//instantly put all of the elements from

//the ArrayList into the array,

//and they are of the proper type:

myObjArray = myObjArrayList.toArray(myObjArray);
```

You now have a regular array that holds objects of type MyObject. You get this without having to loop over each item in the ArrayList and call its get method and perform a cast.

Neat.

Chapter 15. DOCUMENTING YOUR CODE WITH JAVADOC

Here is some stuff about documenting your code. I know this is something that you do regularly, and always make sure to keep up-to-date. Not.

The Java SDK comes with a tool that is easy to use (really) called Javadoc. It does an amazing amount of work for you by taking the API documentation for your applications off of your shoulders. This out-of-the-box functionality lets you write special comments directly in your code, and then it generates formatted HTML files to document your application's API.

Writing Javadoc Comments

Javadoc comments are an advanced form of comments. The purpose of regular source code comments is to indicate to readers of your source code files what you're up to. The limitation here is obvious.

What if I don't wanna distribute my source code files? Even if I did, it's inconvenient and difficult to sort through all of the source code just to find out what a class' contract is.

Then you have to distribute documentation files separately. Duh.

But it's a draaaag to make separate documentation files. They get out of synch with my source code, and then they're worse than useless.

True. You're starting to sound a little whiney, dear, but these are good points. Javadoc addresses your concerns by providing a facility for generating complete code documentation externally, in HTML. It's a terrific and easy documentation tool, because HTML is platform independent and anybody can view it with pleasure.

How to Do It

There are two requirements for generating documentation. First, you need to add the appropriate comments to your file. Second, you need to run the Javadoc command-line tool that creates the files.

The Javadoc tool generates documentation for the following items:

- Packages
- Public classes and interfaces
- Public and protected methods
- Public and protected fields

No private items are included unless you explicitly specify them.

Just write your special comments immediately above the items in your files that they comment (that is, right above your class, right above each field, and right above each method). All of your Javadoc comments must be outside a method body.

Start Javadoc comments with /** and end them with */. Since Java 1.4, the two asterisks are not required, but are convention.

Write a short sentence summarizing a method, or a short paragraph or two to summarize a class.

Within your summary text, you can use the inline tag {@link URL} to have Javadoc create a hyperlink for you. You can also use any HTML tags you want, though it is typically kept to basic items such as <P> and <CODE> to format your comments.

Using Javadoc Tags

You can use the following tags in your Javadoc comments. These get picked up by the tool to serve as the text for your documentation.

- @author Who wrote this class. Use this only with classes and interfaces, not methods. Use separate @author tags for each author if there's more than one.
- @version Identifies the version of the class. Use only with classes and interfaces. Use the following values:

```
/**
 * @author eben.hewitt
 * @version 1.0
 */
```

- @param Indicates a parameter to a method or a type parameter to a class. Use a separate @param tag for each parameter. For use in methods and constructors only. Don't include the param's type, just its name and description, as in the following:

```
/** for method:
 * @param salary The boss's current salary.
 * @param schmoozability How much this boss can
 * schmooze, represented on a scale of 900-1000
 */

/** for class type parameter:
 * @param <T> Some class parameter.
 */
```

- @docRoot Path to the root directory of the documentation, as in the following:

```
* This class is a member of the
* <a href="{@docRoot}/../guide/collections/index.html">
* Java Collections Framework</a>.
```

- @return Identifies a method's return value. Obviously, this is for use only with methods.

```
/**
 *@ return The boss's bonus
 */
```

- @throws Indicates exceptions thrown by this method. Should be used to describe under what circumstances this exception might be thrown. @exception is also acceptable in place of this, but I prefer the active verb.
- @see Points to other relevant classes with a hyperlink, as in the following example:

```
/**
 * @see packageName.ClassName#member text
 */
```

- @since Indicates the release of your software that first introduced this feature. Can be used with classes, methods, and fields.

```
/**
 * @since 1.2
 */
```

- @deprecated Indicates that a method or class is deprecated and shouldn't be used. Write it like this:

```
/**
 * @deprecated As of JDK 1.1, replaced by
   {@link #setBounds(int,int,int,int)}
 */
```

- {@value arg } Accepts the name of a program element and label. This way, it can be used in any doc comment, not just constant fields.

```
/**
 * @value package.class#member label
 */
```

- {@literal tag } Denotes literal text so that the Javadoc tool does not attempt to interpret enclosed text as containing HTML or Javadoc tag.
- {@link link} You use the @link tag inline in paragraph text to have the Javadoc tool automatically generate a hyperlink to the Java class or method you specify, as in the following example:

```
import ...
/**
 * Uses NIO to read in the datafile initially using a
 * @link java.nio.ByteBuffer} and a {@link
 * java.nio.CharBuffer}, and then stores the
 * complete file in a synchronized list.
**/
public class SomeClass { ... }
```

The above will generate a hyperlink to the Javadoc for the class indicated. Notice that you can add a # following a class name to indicate a method you want to link to directly. Doing so means that you have to supply the types of each parameter to the method, but not the parameter names, as in the following example of Javadoc for a public method:

```
/**
 * This client-facing method should be
 * preferred for clients over the {@link
 * mypackage.db.DataClass#updateRecord(long,
 * java.lang.String[], long)} method.
 */
```

Using the Javadoc Tool

Now that you are ready to generate your documentation, whip out a console and get ready to rumble.

Here's the usage for the Javadoc tool:

```
javadoc [options] [packagenames] [sourcefiles] [@files]
```

To see a full list of usage options, simply get a console and type javadoc –help and hit Enter. Here's a basic way to generate your docs:

1. At the console, navigate to the directory that contains the package you want to document.
2. Type javadoc -d DocOutputDirectory PackageName

To document the net.javagarage package and put it into the /eben/home/MyJavaDocs directory, I type this:

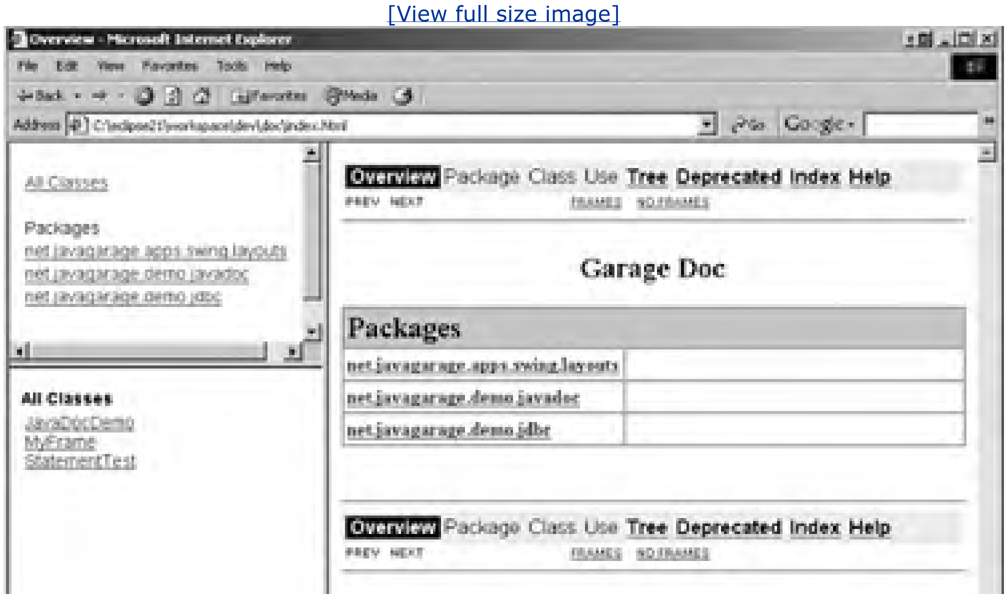
```
>javadoc -d /eben/home/MyJavaDocs net.javagarage
```

The utility runs outputting what it is doing along the way. It creates several directories and many files, and when it's done, you have a folder with all of your documentation. Click the index.html file to view it.

Note that you have several options. You can include all of the classes and packages in your application, or just select a few (see Figure 15.1).

Figure 15.1. The index page of the generated documentation.

[View full size image]


Overview - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Favorites Media
Address C:\eclipse21\workspace\dev\doc\index.html
Go Google
Overview Package Class Use Tree Deprecated Index Help
PREV NEXT FRAMES NO FRAMES

All Classes

Packages
net.javagarage.apps.swing.layouts
net.javagarage.demo.javadoc
net.javagarage.demo.jdbc

All Classes
JavaDocDemo
MyFrame
StatementTest

Garage Doc

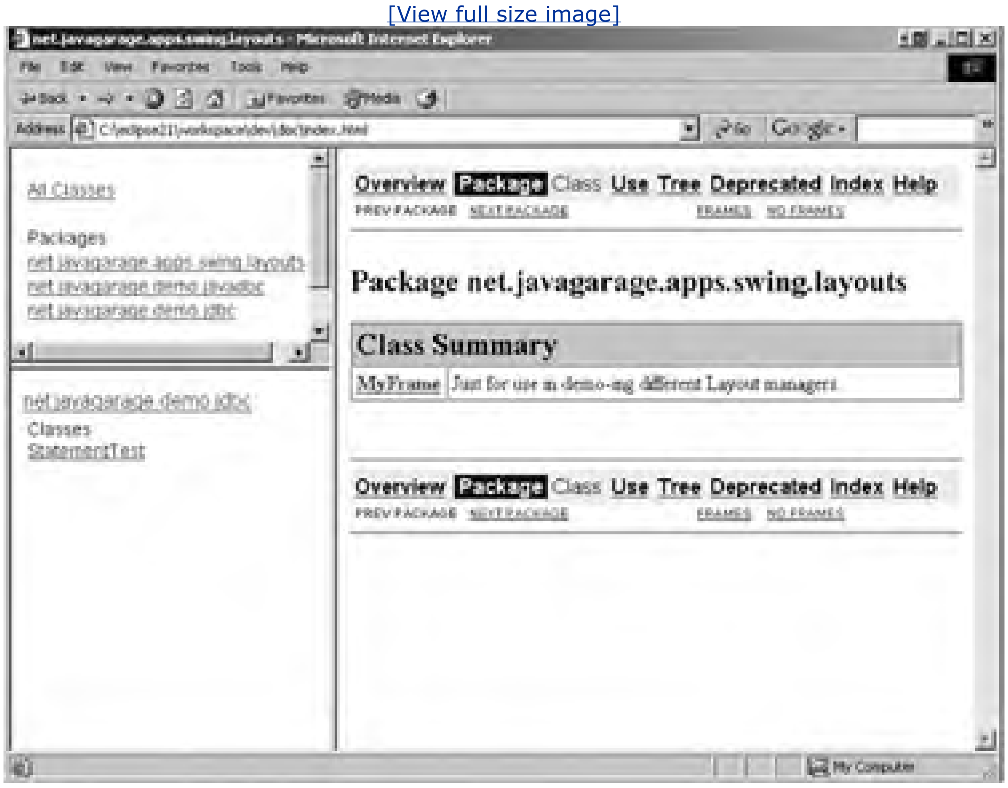
Packages	
net.javagarage.apps.swing.layouts	
net.javagarage.demo.javadoc	
net.javagarage.demo.jdbc	

As you can see, I have only included a few classes from the entire garage kit-and-kaboodle, for simplicity's sake. Notice that it looks exactly like the Java API documentation used online by Sun.

Let's click on the package name to get the class summary for the net.javagarage.apps.swing.layout package. It gives you a list of classes with their descriptions in the main frame (see Figure 15.2).

Figure 15.2. The package's class listing and description.

[View full size image]


net.javagarage.apps.swing.layouts - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Favorites Media
Address C:\eclipse21\workspace\dev\doc\index.html
Go Google
All Classes
Packages
net.javagarage.apps.swing.layouts
net.javagarage.demo.javadoc
net.javagarage.demo.jdbc
net.javagarage.demo.jdbc
Classes
StatementTest
Overview Package Class Use Tree Deprecated Index Help
PREV PACKAGE NEXT PACKAGE FRAMES NO FRAMES
My Computer

Package net.javagarage.apps.swing.layouts

Class Summary	
MyFrame	Just for use in demo-ing different Layout managers

Changing Javadoc Styles

There is a default stylesheet that Sun's Javadoc tool creates when you use it to generate Javadoc. It looks like this:

```
/* Javadoc style sheet */

/* Define colors, fonts and other style attributes

   here to override the defaults */

/* Page background color */

body { background-color: #FFFFFF }

/* Headings */

h1 { font-size: 145% }

/* Table colors */

.TableHeadingColor     { background: #CCCCFF } /*

 Dark mauve

*/

.TableSubHeadingColor  { background: #EEEEFF } /*

 Light mauve

*/

.TableRowColor    { background: #FFFFFF } /* White */


/* Font used in left-hand frame lists */

.FrameTitleFont     { font-size: 100%; font-family:

 Helvetica,

Arial, sans-serif   }

.FrameHeadingFont   { font-size: 90%; font-family:

 Helvetica,

Arial, sans-serif   }

.FrameItemFont      { font-size: 90%; font-family:

 Helvetica,

Arial, sans-serif   }


/* Navigation bar fonts and colors */

.NavBarCell1      { background-color:#EEEEFF;} /*

 Light mauve */

.NavBarCell1Rev   { background-color:#00008B;} /*
```

```
 Dark Blue */

.NavBarFont1      { font-family: Arial, Helvetica,

 sans-serif;

color:#000000;}

.NavBarFont1Rev   { font-family: Arial, Helvetica,

 sans-serif;

color:#FFFFFF;}


.NavBarCell2    { font-family: Arial, Helvetica,

 sans-serif;

background-color:#FFFFFF;}

.NavBarCell3    { font-family: Arial, Helvetica,

 sans-serif;

background-color:#FFFFFF;}
```

I include it here so that you can see the elements that you would need to change if you want to specify your own values.

To change the style your docs use, just use the Javadoc tool in the normal way, but also add the – stylesheet flag.

```
>javadoc -d /my/docs -stylesheet /home/styles/docs.css

     net.javagarage
```

Generating Javadoc in Eclipse

If you are using an IDE such as Eclipse to do your business, you can use a quick weezard to do your dirty work for you.

1. In Eclipse, click Project > Generate Javadoc…
2. Select the packages you want to include.
3. Specify the visibility level you want to include.
4. In the Destination text field, specify the location where you want the docs to go. Click Next.
5. Tweak the documentation options if you want—usually it is best to leave them all selected.
6. Specify a different stylesheet if you like, using the CSS classes as shown previously. Click Finish.

Generating Javadoc with Ant

You can also generate your Javadoc using the Ant utility. Ant is available as a free download from http://ant.apache.org. The current version, as of this publication, is 1.6.1. Ant is used like the make tool to automate the software build process. It offers an XML syntax featuring tags that, among other things, wrap common Java tools, including the javadoc command.

While in this section it is not appropriate to go into the Ant tool in detail, my assumption here is that this will serve as a useful—if quick—reference later when you want to generate Javadoc using Ant.

If you are using Eclipse (3.0 is the current version), you already have Ant installed with it. All you need to do is create a file (conventionally called build.xml), and add it to your project. You can then add Ant tags to that file, and then execute the script using the built-in Ant plugin.

Writing the Build Script

Here is a sample script that you can use in your projects to easily generate your Javadoc.

build.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="My Project Build" default="init">
<property name="src.dir" value="code/source"/>
<property name="docs.dir" value="./docs"/>
    <!—Create documentation. —>
    <target name="Generate JavaDoc" depends="">
        <javadoc packagenames="myrootpackage.*"
             sourcepath="${src.dir}"
             destdir="${docs.dir}/javadoc"
              access="private"
              windowtitle="My Window Title"
              verbose="true"
              author="true"
              version="true">

             <bottom>
                 <![CDATA[<b>Java Garage, Eben
                    Hewitt, 2004<b>]]>
              </bottom>
          </javadoc>
     </target>
```

```
</project>
```

This script will pick up all of the Javadoc tags you have written down to the "private" level. If you specify access="public", then only Javadoc comments on public methods will be written out to the Javadoc.

The text between the "bottom" tags is optional and represents some arbitrary text that you can place at the bottom of every page. It is appropriate for copyright.

When this script is executed, it creates all of the Javadoc for you, and places the files in the docs/javadoc directory—the value of the destdir attribute.

For extensive, helpful documentation on using the Ant tool to automate your builds, check out http://ant.apache.org/manual/.

Executing the Ant Script in Eclipse

While you don't need Eclipse to execute Ant scripts, Ant is often used as a plugin to an IDE, and comes built-in with Eclipse, and is the simplest way to demonstrate without going into detail here. Note that you need to have a directory structure that matches the one you specify in your build.xml file for the Javadoc destination.

Once you have the script modified to your liking, here is how to run it:

1. In the Eclipse IDE, choose Window from the main menu, and click Show View > Ant.
2. Click the three yellow plus sings to add the build.xml file to your project.
3. Once you have done so, click the green forward arrow in the Ant window to execute the script. The console window will open and show you the verbose output that Ant generates as it executes the script.

Ant is explored in more detail in the *More Java Garage* book.

That's it! Happy Javadoc-ing.

Chapter 16. ABSTRACT CLASSES

DO OR DIE:

- Clean my room.
- Take out the trash.
- Double RAM in server.
- Take over the world by ruling a race of mindless robots that I control through my Tivo.

An abstract class is mostly potential. A regular class must implement all of the methods it declares. An abstract class can implement some of the methods it declares, and leave some for a subclass to implement.

The abstract class can be a useful thing. In this topic, we'll take a look at what they are, how to write them, how they behave, and when to use them.

Dude, Where's My Implementation?

An abstract class is a class whose declaration includes the keyword abstract. Based on that definition, you should be able to write the simplest possible abstract class. Okay…geez…*I'll* do it. How about this:

```
public abstract class SimplestAbstractClass { }
```

That meets the definition, and so it compiles.

You might never write an empty class such as this. But there are occasions when you will come across such empty classes; such usage aims to provide a formal organization to an application for later possibilities for extensibility.

The following is also an abstract class:

```
public abstract SecondAbstractClass {

   public abstract void someMethod();

}
```

The preceding code declares an abstract class that includes an abstract method. The class should be subclassed, and when it is, an implementation for someMethod() must be provided. Notice that abstract methods don't have any curly braces (which is where the implementation would go). They just end in a semi-colon. If you end a method declaration in a semi-colon, you must also declare it abstract.

The following is also an abstract class:

```
public abstract class ThirdAbstractClass {

    int counter = 0;

    void printMessage(String msg) {

         System.out.println("Your message: " + msg);

    }

}
```

Notice how we have what looks to be a totally regular class. It has a member variable that is initialized to a value, and it has an implemented method. But it's declared abstract, and it compiles. What gives?

Abstract classes can consist of implementations, such as in the preceding ThirdAbstractClass, or they can *defer* their implementation entirely, or they can have a combination of stuff they implement and stuff they defer to let someone else implement.

PREV < Day Day Up > NEXT

What Is an Abstract Class?

You cannot instantiate an abstract class. That is, you cannot make an object of it. You can't write new SecondAbstractClass(). Ever. This is sensible, as it is the way the world works. The world has things in it, like balls and books and rocks. The world also has abstract concepts that help us organize actual things, such as rocks and people, and to organize other concepts. The economy is a good example. There is no such thing as "the economy." Anywhere. It doesn't exist. We talk about it enough. "Blah blah blah…how's the economy…. Ah it's horrible…. Ah well what're you gunna do…." Because "the economy" is a useful placeholder for a collection of other things—some of which are physical objects, and some of which are other concepts. You can't touch the economy, or hang it out the window, or kick it. It doesn't exist. You can't say new Economy(). You can only allocate 20 pounds British Sterling to Jill, and transfer 1,000 shares of Sun Microsystems stock to Phil, and collect $387.65 interest on your specific mutual fund run by a specific company containing specific stocks of specific companies that exist in the world. Those are the actual things of the economy. And the interest rate and whether the NASDAQ is generally bullish or bearish, these are abstract. So it is well and just that abstract classes should support both a little implementation (thing-ness) and a little idea (abstract-ness). Or neither. Or one. Or both.

That's why SimplestAbstractClass, SecondAbstractClass and ThirdAbstractClass above all make sense as possible representations of an abstract class within the Java programming language.

Some Guidelines About Abstract Classes

- When you want to declare a method to indicate something that should be done, but you want to defer the implementation to a subclass, you use an abstract class. That's what they're for. Any method that you want to let a subclass implement you declare using the abstract modifier. We can call this an abstract method.

- You can mix implementing methods and abstract methods in an abstract class. However, a class must be declared abstract itself if it contains *even one* abstract method.

- You cannot instantiate an abstract class. That is, you cannot make an object of it. You can't write new SecondAbstractClass(). Ever.

- A class cannot be both abstract and final at the same time. Think about it: abstract means that you must extend the class and implement it somewhere, and final means you are totally not allowed to ever extend this class. They are mutually exclusive. Note that when I write, "you must extend it…," I don't really mean it. That is, you can create an abstract class in your app and never have any other class extend it. It just wouldn't do anything then.

Using Exceptions with Abstract Classes

A common question regarding abstract classes is how to use exceptions with their method declarations. What exceptions can an inheriting method throw with respect to the inherited method? Consider the following class, whose single method throws one exception: IOException.

```
public abstract class AbstractPrinter {

abstract void printMessage(String msg) throws IOException;
}
```

The implementation of this abstract class is shown in PrinterImp1.java. To implement an abstract class, you simply extend it, just as you would extend any other class, using the extends keyword.

PrinterImpl.java

```
package net.javagarage.demo.abstractclasses;

import java.io.IOException;

public class PrinterImpl extends AbstractPrinter {
  void printMessage(String m) throws IOException {
    System.out.println("Your message:
         " + m);
  }
}
```

FRIDGE

This is different than in C#, in which classes are extended using the : operator. That is the same way that interfaces are implemented in that language. To implement an interface in Java, you use the keyword extends.

The preceding shows that the rules are the same as when you extend a regular class.

- You cannot declare that your implementation method throws an exception if the superclass method does not.
- If your abstract method declares that it throws an exception, your implementation method must throw that same exception, or any subclass of that exception.

Using an Abstract Class Polymorphically

A chief benefit of object-oriented programming is polymorphism. Abstract classes are one way that Java supports polymorphism. That is an issue best left for another topic all to itself. But here, let's look at what is perhaps the most common reason people make abstract classes in the first place: to get flexibility in their design.

Say you have a little Dungeons and Dragons type game where some hero fights monsters. In this adventure, it is possible for the Hero (we'll make him a class) can encounter Serpents or Goblins. Looking at our requirements, we might determine that a Serpent and a Goblin have certain traits in common. They are both monsters. Let's put a capital M on that and make it a class: Monster. Good. Now the Monster class will be the abstract superclass of both Serpent and Goblin, and they will be concrete subclasses. We don't allow any instances of Monster itself—you have to make some particular *kind* of Monster.

This is a pretty good design. Because now when we need to make a monster do something in the game, we don't have to clutter up our design worrying about making the Goblin move and fight and then practically doing the whole thing over again making the Serpent move and fight. They will both move and fight, but they will move and fight in different ways. The Goblin will just walk to move. The Snake will slither, which is a more specific kind of movement, different from how it will most frequently be implemented. The Goblin will fight by swinging a sword, and the Snake will fight by biting. Again, they both need to do it, but each does it in its own way.

Why not just make a Goblin class and a Snake class and forget about Monster? Who cares about Monster if the Hero will never actually encounter anything that is a Monster, but only ever subclasses of Monster? Here's why: There are certain things that the Hero does when he or she encounters a monster, regardless of what type of monster it is. These things probably include "fight." Consider what that means in terms of your design.

It might seem okay right now to make a method signature that looks like this: `heroInstance. fight(Goblin goblin);`. But you have to replicate most if not all of that functionality making another method called `heroInstance .fight(Serpent serpent);`. Now you're in Copy and Paste land. And that's a dark forest indeed. See where the Monster comes in handy?

Here's a rule: If you find yourself highlighting a method implementation and typing `Ctrl+C`, you probably need an abstract wedgie in there somewhere. Just don't do it—don't copy that code. Just release the keys and step away from the laptop. You don't have to live like that. You can be abstract.

Adventure.java

```
package net.javagarage.demo.abstractclasses.dungeon;


/**

 * <p>

 * @author eben hewitt

 */

public class Adventure {

    String name;


    //constructor that accepts a name for this adventure

    public Adventure(String nameIn){

        this.name = nameIn;

    }


    //program starts here
```

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

    //make a hero to go on adventure
    Hero harry = new Hero();
    //make a monster:
    //dig the polymorphism—
    //the reference type is the abstract superclass!
    Monster gizmo = new Goblin();
    //make a new adventure
    Adventure chamberOfHorrors
         = new Adventure("Chamber of Horrors");
    //start them off
    chamberOfHorrors.go(harry, gizmo);
}

//do the adventure
private void go(Hero hero, Monster mst){
    System.out.println("Welcome to The " + this.name);
    //blah blah blah
    System.out.println("You hear a
               horrible sound...");

    //the important part: call the private method
    //that represents the encounter
    doCombat(hero, mst);

        //all finished
    System.out.println("The " +
        this.name + " is done.");
}

//we can pass in any subclass of Monster,
//and it will act correctly.
private void doCombat(Hero hero,
                      Monster monster) {

     String monsterType =
            monster.getClass().getName();
        //dig the polymorphism...
        System.out.println("Hero sees a " + monsterType + "!");

        System.out.println("Will it approach you?");

        //dig the polymorphism again
        monster.move(12);

        //whoa! again!
        monster.fight();

        hero.fight();

        //okay, we get it now.
        if (hero.strength > monster.strength)
        {
        System.out.println("You defeated the
            " + monsterType + "!");
        } else {
        System.out.println("The " + monsterType + "
            defeated you!");
        }
    }
}
```

Hero.java

```
package net.javagarage.demo.abstractclasses.dungeon;
/**
 * <p>Some nerdy do-gooder who fights with Monster
 * subclasses in an Adventure
 *
 * @author eben hewitt
 * @see Adventure
 * @see Monster
 */
public class Hero {

    //used to determine who wins fights.
    int strength;

    //Constructor to make Heroes with.

    public Hero (){
        //randomly determine strength
        //so every hero object is different
        this.strength = (int)(Math.random() * 100);
    }

    //we'll pass a Monster to fight with into
    //this method just to show how polymorphism works
    //and keep this as simple as we can.
    public void fight() {
        System.out.println("The Hero attacks
        with a sword with
        strength " +
        this.strength + "!");
    }
}
```

Monster.java

```
package net.javagarage.demo.abstractclasses.dungeon;
/**
 * <p>Abstract class, represents an enemy of Hero.
 * @author eben hewitt
 * @see Hero
 */
public abstract class Monster {
    //we'll use this as a simple way to
    //determine winner of fights
    int strength = (int)(Math.random() * 100);

    //this will be different for most monster types
    String attack;


    //most monsters will walk, so we will provide
    //a default implementation for them to do that
    //so every walking monster in the world doesn't
    //need to implement this, since it would be
    //roughly the same.call getClass().getName()
    //to prove the runtime type
    public void move(int numSpaces){
        System.out.println(getClass()
                getName() + " walking " +
                numSpaces + " spaces.");
    }

    //abstract method. we don't know enough about how
    //each monster will do this, and it will probably
    //be at least a little different for each kind
    abstract public void fight();
}
```

Goblin.java

```
package net.javagarage.demo.abstractclasses.dungeon;
/**
 * <p>A subclass of Monster that is a terrible beast.
 * @author eben hewitt
 * @see Monster
 * @see Serpent
 * @see Adventure
 */
 ,
public class Goblin extends Monster {
    //constructor. We don't have a member
    //var called 'attack' but remember
    //that our superclass Monster does

    public Goblin() {
        this.attack = "club";
}

    //implement fight in some Goblin-specific way
    //for this example, that just means hard-coding
    //the String 'Goblin' in there

    public void fight() {
        System.out.println("The Goblin attacks
              with a " + attack
              + " with strength " + strength + "!");
    }
}
```

Serpent.java

```
package net.javagarage.demo.abstractclasses.dungeon;
/**
 * <p>
 * @author eben hewitt
 */
public class Serpent extends Monster {
    public Serpent(){
        this.attack = "bite";
    }

    public void fight() {
        System.out.println("The Serpent
       attacks with a " + attack
               + " with strength " + strength + "!");
    }

    //remember that Goblins just walk—they can
    //just inherit that functionality from Monster.
    //Serpents move a different way:

    public void move(int spaces){
        System.out.println("Slithering " +
               spaces + " spaces...");
    }
}
```

Here is the output when we make a new adventure and make the Hero fight a Serpent.

```
Welcome to The Chamber of Horrors
You hear a horrible sound...
Hero sees a net.javagarage.demo.abstractclasses. dungeon.Serpent!
Will it approach you?
Slithering 12 spaces...
The Serpent attacks with a bite with strength 28!
The Hero attacks with a sword with strength 61!
You defeated the net.javagarage.demo.abstractclasses.dungeon.Serpent!
The Chamber of Horrors is done.
```

The Hero wins! Here is the output when we make a Goblin monster.

```
Welcome to The Chamber of Horrors
You hear a horrible sound...
Hero sees a net.javagarage.demo.abstractclasses.dungeon.
```

```
Goblin!

Will it approach you?

net.javagarage.demo.abstractclasses.dungeon.Goblin walking 12 spaces.

The Goblin attacks with a club with strength 14!

The Hero attacks with a sword with strength 5!

The net.javagarage.demo.abstractclasses.dungeon.Goblin defeated you!

The Chamber of Horrors is done.
```

Oops. The Hero lost that time.

The important thing is to notice that we can make any kind of Monster subclass—Skeleton, Gelatinous Cube, Wraith, Necromancer, whatever we want—and it won't break any code. We can use Monsters of types we haven't thought of yet, or that someone else thinks of later, and our Adventure class methods can stay the same. Very cool.

Look for places in your application where you can make use of this kind of design. It is extra work up front, but you will see productivity and ease of maintenance benefits later. As my friend from Connecticut always says, "You're going to pay now, or you're going to pay later—and it's usually cheaper to pay now."

Chapter 17. INTERFACES

EL OBJECTIVOS:

- Learn It
- Live It
- Love It

The interface is Java's answer to multiple inheritance. It is a Java type that defines *what* should be done, but *not how* to do it. Interfaces are perhaps most useful when designing the API for your program. In this topic, we'll find out how to define an interface, how to implement one, and how to use it in program design.

Let's Get One Thing Straight

Let's get one thing straight. Right off the bat. There are really two ways that people in Javaland use the term interface. One is conceptual and one is concrete.

People sometimes talk about a program's interface, or even a class' interface. Remember that API is an acronym for Application Programming Interface. When used this way, it means *the thing with which we interact*. It's what is exposed to us that we can work with. It is the visible boundary of a class or a program or a programming language's libraries. This is the conceptual version of the term interface. It means the public (I really mean non-private) methods that we can call to do something.

On the other hand, an interface is a Java programming language construct, similar to an abstract class, that allows you to specify zero or more method signatures without providing the implementation of those methods. Remember the implementation is the code block that does the work. Let's look at the difference.

```
public void printMessage(int numberOfTimes);

// a method declaration in an interface.
```

The preceding code is an example of an interface method declaration. Notice how the signature ends in a semi-colon. There is no code that does the actual work of the printMessage() method.

An interface will consist of method signatures that look like this; there can be no implementation at all. You do the implementation in a class. An implementation of the printMessage() method might look like this.

```
public void printMessage(int numberOfTimes) {

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

    System.out.println("Current message number " + i);

  }

}
```

Or the implementation could be different. Maybe a different implementation uses a while loop, and has a different String message, or doesn't print the current iteration of the loop as represented by i.

Imagine an interface with some method signatures, and a class that will implement the interface.

> Interface: I have 5 method signatures.
>
> Class: I want to implement them.
>
> Interface. Okay. But then you have to implement all of them. You are not allowed to say that you implement me without implementing every single one of my methods.
>
> Class: It's a deal.

An interface is a contract. It is binding between the interface and the class that implements the interface.

But why in the world would you want to do that? A class can implement whatever methods it wants. Why not cut out the middleman, go ahead and implement the methods you want, and then forget about the interface altogether? Well, you could do that. But there are actually several reasons why interfaces are very cool.

If it helps, you can think of an electrical outlet.

An electrical outlet is a wonderful invention. It is really, really a cool thing, the electrical outlet. The interface for every electrical outlet is *exactly the same* (okay, they're different in the United States than other places, and sometimes you get three little holes to plug stuff into, and sometimes only two; work with me here. Geez). You know that you will be able to use the electricity you need for your laptop, PlayStation, hair drier, or electric dog polisher as long as each of them have standard cords that will plug into an outlet. We could imagine a hypothetical interface called Pluggable that means that it has a standard set of prongs that will plug into a standard outlet. A bowl does not implement the Pluggable interface. Can't plug it in. The book "The Complete William Shakespeare" doesn't implement the Pluggable interface. Can't plug it in. However, an e-book reader does implement the Pluggable interface. You could read "The Complete William Shakespeare" on e-book or in paper form. Same text. Different implementation.

So back to our story.

Some Reasons Why Interfaces Are Very Cool

1. **They encourage smart application design**. Say that you have an application like an e-commerce site. You have to access a database a lot for different reasons. On one occasion, you need to retrieve products and display the catalog; another time, you need to store customer information. Now, these operations aren't related much in terms of the domain. But they do the same thing: interact with the database. You might create an interface called DataAccessor that defines all of the operations that can be done with the database: select, insert, update, delete, and so forth. Then, you have one standard way that all of the programmers on your project can write to. All of your database access contracts, even if written by people working on unrelated parts of the app, will look the same.

2. **They promote readability**. I have seen enough people come and go in different jobs that I am a firm believer in doing what you can to make your code readable. Interfaces promote readability because readers of your code know what to expect of a class that implements an interface. Also, it gives readers a second, concise location to overview what the class does.

3. **They promote maintainability**. The reason that they promote maintainability is more complicated, and we'll get to it in a moment. In a nutshell, if a class implements an interface, you can use the interface type as the reference type of instances of that class. That means that later you can swap out the actual implementation without breaking any of your code. That ability is an instance of polymorphism, one of the pillars of object-oriented programming. More on this in a mo.

4. **They allow flexibility in your class definitions**. In Java, you are only allowed to explicitly extend, or inherit from, one class. This is different from languages like C++ and C# that allow you to extend multiple classes. By allowing Java classes to extend from one class and also implement an interface at the same time, we can in effect skirt the fact that we're not allowed multiple inheritance, because of polymorphism. In fact, Java programmers are often encouraged to implement an interface instead of inheriting from a class when faced with that choice. Programmers probably come to such a crossroads when creating a new thread. You can create a thread by extending the Thread class or by implementing the Runnable interface. They both get you a thread. But you can implement as many interfaces as you want—you can only inherit from one class. So the idea is to prefer interfaces over inheritance unless you really are extending the definition of what your superclass can do. That is, nine times out of ten we aren't actually extending the functionality of a thread when we write `extends Thread`. We're not making a more specific, more functional kind of thread; we're just wanting to spawn a new instance of a regular old thread so that we can execute some code separately from the main program thread. In this case, you should implement Runnable and get your thread that way. That leaves you with the ability to extend a different class if you want to, one whose functionality your class is closer to, or that you really need to inherit from.

How to Write an Interface

The Java API is full of terrific interface definitions. One commonly used interface is java.lang.Runnable, which can be implemented to create a thread. To write an interface, you can look at the many good examples in the Java API.

When writing your own, you just use the keyword interface instead of class, and then don't include the method implementations. Like this:

```
public interface MyInterface {

      public void someMethod(long someParam);

}
```

As you can see, you just use the interface keyword in place of class. And then you substitute a semi-colon for the curly braces holding the method implementation. That's all you have to do in a nutshell. In the following sections, we'll see the many twisting, cavernous corridors that lurk beneath the deceptively simple interface.

After you have an interface, you implement it. That means you declare a class that implements the interface, and then write the implementation for each method in the interface. And the method signatures must match exactly. Like this:

```
public MyClass implements MyInterface {

   //since I said "implements MyInterface",

   //I must include this method, or

   //this class won't compile

   public void someMethod(long someParam) {

       //this is my implementation.

   }

}
```

You can add methods and other stuff to your implementing class if you want to.

Interfaces Versus Abstract Classes

There are differences between an interface and an abstract class, though commonly the two are confused. Why do you need an interface? When should I use an interface and when an abstract class? Let's let the two duke it out for themselves, and you decide.

Monsters fight!

Round One: *An interface is a totally abstract class*. In an abstract class, you can define some methods that are abstract, and some methods that are fully implemented. Any class that extends that abstract class must implement the abstract methods, but it inherits the functionality as implemented in the abstract class. This is very cool if you need that kind of structure in your program, but can make your API a little snaky.

Round Two: In an interface, there can be no implementation whatsoever. You can say this:

```
List starWarsFigures = new ArrayList(500);
```

Then, your reference type is the interface type (List)! That means that later in your code you can change the implementation without breaking any of the code that does something with your starWarsFigures object. Like this:

```
List starWarsFigures = new Vector();
```

This is very cool if you need that kind of flexibility, which is often a good thing.

Round Three: An interface is less flexible in how its variables and methods are declared. Here are the rules about writing an interface.

Rules for Writing an Interface

- Declare an interface using the keyword interface.
- An interface may extend zero or more interfaces if it likes, but it cannot extend a class, because then it would inherit functionality, and interfaces cannot have functionality. They can only talk about it.
- Interface methods cannot be static.
- Interface methods are implicitly abstract. For that reason, you cannot mark them final (duh), synchronized, or native because all of these modifiers tell how you're going to implement the method, and you're voluntarily giving up that ability when you write the method in an interface.
- strictfp is okay on an interface. It is okay because you can evaluate compile-time constants using strictfp rules within an interface.
- All interface methods are implicitly abstract and public, *regardless of what you write in the interface definition!* It is true. The interface method declaration void biteIt(int times), despite its apparent access level of default, actually has public access. Try it. If you write a class in another package beyond the visibility of default access,

and include the seemingly legal implementation of `void biteIt(int times) { ; }`, the compiler will tell you that you cannot reduce the visibility of the method from `public` to default. They're all abstract; they're all `public`.

- An interface can define variables. But all variables defined in an interface must be declared `public`, `static`, and `final`. Many Java programmers have adopted the practice of defining only variables within an interface and putting constants in it. This works to get at shared constants, but it is a workaround and is no longer necessary if you're using J2SE SDK 5.0. It features a new static import facility that allows you to import constants just as you would a class or package.
- It should be obvious by now that an interface cannot implement another interface or a class.
- You may modify your methods using the keyword `abstract` if you like, but it will have no effect on compilation. Methods in an interface are already abstract, and the Java Language Specification says that its use in interfaces is obsolete.
- Likewise, the interface itself is already abstract. So you can do this if you want: `public abstract interface Chompable {}`. But there's no point; it's redundant.
- Interfaces have default access by default (!). So this is legal: `interface Chompable { }`. But if you want your interface to have public access, use that modifier in the interface declaration.
- You cannot declare an interface as having private access. It doesn't make any sense. No one could implement it. So `private interface Chompable { }` gets you a compiler error for your trouble.
- `public`, `static`, and `final` are implicit on all field declarations in an interface.

There are some weird things to keep in mind.

Interfaces can be declared `private` or `protected` if they are declared nested inside another class or interface. The following will compile, though its usefulness is dubious at best.

```
public class interface test {

  private interface myinterface{ }

}
```

Only an inner class of the class of which the interface is declared can implement the interface.

Is and Does

Remember that the job of an interface is to designate a role that a class will play in the society of the application. Whereas a class (especially an abstract class) defines what something is, an interface defines what it can do.

Which of the following would you make into an interface, and which would you make into an abstract class?

Person

Employee

Programmer

Skier

WildAnimal

DataAccessor

Swimmer

WestCoastChopper

You'll see a 3-dimensional pattern emerge if you relax your eyes and stare at it long enough. This sort of thing is important to keep in mind as you design your application. You can do all of your applications without ever using an interface. But again, they clarify your API and provide you with flexibility down the road.

Constants

Java programmers commonly define constants inside their interfaces, if it makes design sense. You can do so using variables in an interface because the values will be present instantly at runtime and their values shared among all classes implementing your interface, because they are static and final.

Here is how you do it.

```
public interface IDataAccessor {

  String DB_NAME = "Squishy";

}
```

FRIDGE

Like the public and abstract deal, interface variables are implicitly public, static, and final. That is, the following are equivalent within an interface: public static final String DB_NAME = "Squishy" and String DB_NAME = "Squishy". Likewise, you are not allowed to do this: private String x;

Note that it only makes sense to define interface constants if the variables are tightly related to your interface definition. There is a pattern that has been popular among developers where they use interfaces for the sole purpose of defining constants, so you might come across a lot of code to that effect. But J2SE 5.0 introduces other mechanisms that are more appropriate for this sort of thing—namely typesafe enums and static imports.

The reason that you don't want to use this old pattern anymore (assuming you're using it) is that it confuses the API. To access the constants defined in the interface, you must include implements InterfaceName in your class definition—and that isn't really true. You aren't implementing any functionality defined by such an interface; you just want the variables. It's a workaround, and now there are language features to handle that situation, so you don't have to resort to it.

Interface Inheritance

Yes, an interface can extend another interface. Just say that it does, like this:

```
public interface ISpy extends

     IInternationalManOfMystery {...

}
```

This is rarely used in practice. It means just what you would expect: Now the class that implements ISpy must also implement all of the methods from IInternationalManOfMystery.

Also, an interface is allowed to extend more than one interface. Just separate them by commas. Notice that this is different than regular classes, which are not allowed to extend more than one class.

Implementing Interfaces

You have to do two things to implement an interface. You have to announce that your class will implement the interface in your class declaration, like this:

```
public class SuperHero implements IFly {...}
```

Then, you have to implement each method defined in an interface. Most IDEs, including Eclipse, will tell you the names of the methods you need to implement as a convenience. Note that whether you have implemented each method you're supposed to is checked at compile time, not at runtime.

FRIDGE

It is a naming convention that you'll often see to prefix "I" before the name of an interface in order to identify it easily as an interface. If you've programmed in VB, you're probably used to Hungarian notation, which serves a similar purpose.

Remember that the purpose of an interface is to say that the implementing class must have methods that match the signatures. That doesn't say anything about the quality of the implementation. Say we have an interface called IFly (which we'll define next), and it has one method `void accelerate(int speed);`. The following is legal, and will compile:

```
public class SuperHero implements IFly {
    public void accelerate(int i) {}
}
```

This can be convenient if an interface has a method or two that you don't really need. But if you find yourself doing this sort of thing with any frequency, you might take a second look at whether you really ought to be implementing that interface, and if there isn't some other way to solve the problem. If you are the designer of the interface, you might want to reexamine your interface design to make sure that you are defining what users (in this context, programmers who want to implement your interface, including you) really need. By the same token, if you find in your code a number of methods that programmers need to define over and over again, consider moving them into the interface and making them part of the contract. This could have the effect of tightening and strengthening your abs. I mean your API.

A class can implement more than one interface. Separate each interface with a comma in the class declaration, like this:

```
public class RockStar implements IGuitarPicker,
    ILeatherPantsWearer { }
```

Abstract Class Implementation of Interfaces

This is something very weird. Actually, I'll pose it as a question and see what you think. Previously, we had our IFly interface, and our regular old class SuperHero implemented it.

(Deep, oily announcer voice) For 50 points, can an abstract class implement an interface? *(After slight pause, with greater significance)* Can an abstract class implement an interface?

Think about it (after all, it's worth 50 points). Can an abstract class implement *anything*? Actually, yes, it can. It can have both abstract and concrete methods. So the answer must be yes. Because you can do this:

```
public abstract class SuperHero implements IFly {
   public void accelerate(int speed) {
      // some code here to do accelerating...
   }
}
```

Any class that extends SuperHero will inherit this functionality, or alternatively, it can decide to override the accelerate() method if it wants to, and define a different implementation.

That was easy enough. But here's the killer: Is the following legal?

```
public abstract class SuperHero implements IFly { }
```

Notice that there is not only no mention of the accelerate() method, there is no code at all. How could an abstract class say it implements an interface that it doesn't implement? It must be wrong!

Actually[1], that code compiles without a problem. Because there is another contract at work when you define an abstract class, and that contract is that the interface methods must *have been implemented* by the first concrete subclass of the abstract class. The first concrete class is not required to implement all of them, just those that have not been previously implemented by some abstract class that implements the interface up the hierarchy. The following series of classes is legal and compiles.

[1] I hate it when people say, "actually." It sounds soooo snotty. As if there is some other life that you're living over there in happy fantasyland where everybody has all wrong ideas about stuff and you're so dumb that it can't be imagined how you even function.

IFly.java

```
public interface IFly {
    void accelerate(int speed);
    void slowDown(int speed);
}
```

Now an abstract class will say it implements the IFly interface.

SuperHero.java

```
public abstract class SuperHero implements IFly { }
```

JusticeLeagueSuperHero.java

```
public abstract class JusticeLeagueSuperHero
        extends SuperHero
{
    public void accelerate(int speed) {
      //some real implementation code!

      System.out.println("Accelerating to " + speed + "mph");
    }
}
```

Notice that JusticeLeagueSuperHero implements the accelerate () method, but not the slowDown() method, which is also required by the interface. The code compiles because it is also declared abstract, and it is expected that one day someone will want to make a concrete subclass of JusticeLeagueSuperHero, so that you can say new SuperHero() and get to work. Of course, if no one ever makes a concrete subclass that implements the slowDown() method, the interface is not really implemented, even though we said it was. But that couldn't matter less, because we can't ever make any objects.

So now let's look at the final class in our implementation of this interface, the concrete class SomeMoreSpecificSuperHero.

SomeMoreSpecificSuperHero.java

```
public class SomeMoreSpecificSuperHero

        extends JusticeLeagueSuperHero {


  public void slowDown(int speed) {

    //some real implementation code here!

    System.out.println("Slowing down

        to " + speed " mph");

  }

}
```

In our little hierarchy, we have an interface, an abstract class (SuperHero) that implements the interface but doesn't actually implement any methods, a second abstract class (JusticeLeagueSuperHero) that extends SuperHero and provides a concrete implementation of one method in the interface, and then our first concrete class (SomeMoreSpecificSuperHero) that extends JusticeLeagueSuperHero and implements the remaining method. By the time we get to the concrete class, everything has an implementation. Cool.

Well, I'm going to get a beer and make a pizza. Thanks for talking about interfaces with me. I'm feeling much better now.

PREV < Day Day Up > NEXT

Chapter 18. CASTING AND TYPE CONVERSIONS

DO OR DIE:

- Stop
- Drop
- Roll

Here we go, into the land of casting. This is where we tell you to empty the rusty old coffee can containing your meager savings. You'll need a bus ticket, since we're sending you to Hollywood to be in pictures.

It's going to be just terrific.

But in case our blockbuster career gunning down everything in sight alongside an ever-balder Bruce Willis doesn't work out, let's do what every out-of-work Hollywood actor does: get a day job by learning runtime type information and casting in Java. But just until we get our big break.

Casting

The purpose of casting is to allow conversions from one type to another.

To get your mind around casting, it is helpful to think of the primitive types as being buckets of bits. Some buckets are bigger than others: long is a bucket of 64 bits, float is a bucket of 32 bits like int, and so forth.

Here is an example using primitives:

```
short s = 16;

//this is okay without cast, since int

//is a bigger bucket (32 bits) than short (16 bits)

int i = s;
```

So we can do this, right?

```
short s2 = s + 26;

//Wrong!! Here's the output:


net\javagarage\casts\Casting.java:21:

    possible loss of precision

found : int

required: short

            short s2 = s + 26;

                     ^

1 error
```

Where does Java say the error is? At the + operator. And what exactly is wrong with trying to add two shorts together to make another short? Nothing at all. Except, despite appearances, that's *not* what we're doing.

When you perform an operation like this on integral primitive types, the two operands are automatically promoted to ints, even though the result would fit inside a short. That's what always happens when you perform mathematical operations on integral types. To be promoted in this context means that they get to be a bigger sized bucket. And you can't fit an int-sized bucket into a short-sized bucket, can you?

Well, no you can't. In this case, we know that our resulting number (16 + 26 = 42) is plenty small enough to really be stored in a short. So we want to tell the compiler that we know it will be okay. Little E.T. will get home in the end. The cast in Java is the programmer's way of saying, "I know that you don't want me to do this, and you're usually smarter than I am, especially about Java, but I just need to do this right now, so you're going to have to trust me. If it blows up at runtime, I'll take responsibility."

The cast operator is (…). In between the parens, you put the name of the type to which you want to cast.

So here's how we solve it:

```
short s2 = (short) (s + 26);
```

This gets us what we want: a short containing the result of 42. Notice that we have put parentheses around the addition operation. Why's that? Because if we didn't, the (short) operator would cast the variable s to a short (which it already is) and would then try to perform the addition, and what would happen? You'd be adding two shorts together again, and the operation would promote the result to an int implicitly, and the compiler would blow up. Well, it wouldn't exactly "blow up," but it would become momentarily unhappy, and take it out on you in the form of console complaints.

When you cast, you leave yourself vulnerable to blowing up at runtime. Why's that? Because you are taking the reins. And you might not know as much about bits at runtime as the JVM. Not to rain on your parade (and we were just getting so excited about casting), but can you guess what happens here?

```
//this is the most an int can hold:

//2147483647

int j = Integer.MAX_VALUE;


//knock it over the edge

int result = j + 1;


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

This is an error, right? No! It's perfectly legal. So what in the world does it print? An int can't, by definition, hold more than it can hold. It prints this:

```
-2147483648
```

Overflow is what happens when a result is too big for the type of variable that is meant to hold it (that is, the int's bucket o' bits isn't big enough to hold +2147483648). What we've got here is an overflow.

If an integer overflows, only the least significant bits are stored. So exceeding your bucket's limit causes an overflow that results in a negative number—in this case, the smallest possible value for an int. So you can guess what is printed here.

```
j = Integer.MIN_VALUE;

result = j - 1;
```

Yep: 2147483647. It's like those revolving fireplace doors in those old Vincent Price movies. It just comes back around.

Java handles this situation differently for integer types than for floating point variables. When a double or a float overflows, the result is positive infinity. When a double or float underflows—that is, when it is too small to be represented properly—the result is 0.

Illegal operations, such as dividing by 0, result in NaN (Not a Number). Otherwise, floating point expressions will not raise exceptions; they will simply default to one of these predefined modes.

Note: You cannot cast a boolean primitive to any other primitive type in Java. It is "special." Shhhhh. This means you cannot convert a boolean to a 0 for false or a 1 for true as in some languages. A boolean's values are the literals true and false, and those values aren't strings. They're booleans.

Also note: If what you need is to perform some hard-core mathematical operations including unbounded arithmetic, you can use the five classes in the java.Math package (not java.lang.Math) BigInteger and BigDecimal. These guys perform more slowly than their counterparts in primitiveland do, and so they aren't used unless necessary.

And now, back to our previously scheduled topic. We were talking about casting.

This code is a compiler-error-making bit of code:

```
int I = 37F; //No!
```

The compiler will get mad, saying that you stiffed it. You told it you were going to provide an int, but instead you gave it a float.

The compiler doesn't like the following either (sort of picky, this compiler…):

```
float f = 69.99;
```

Why in the world not? Remember how Java likes its integral types to be integers by default? It likes its floating point numbers to be doubles by default. So here, we said that we'd give f a float, but instead, we gave her a double.

You have to make it explicit then, using the F following the float. This fixes it:

```
float f = 69.99F;
```

You can also write D following doubles.

```
double d = 87.65D;//ok jose

double d = 122984.8604 //ok jose
```

Note that casting from decimal to integral types can result in the parts following the decimal simply getting lopped off, as with a guillotine.

```
double d = 1405.9876;

int i = (int) d; //gives 1405
```

Of course, hacking apart your floating point numbers in the manner of the chain saw may not prove aesthetically pleasing enough for you. In this case, you can try the java.lang.Math.round() method, which returns a passed float as an int.

Casting Between Primitives and Wrappers

Here are the rules about it:

A value of a primitive type can be cast to another primitive type by identity conversion, if the types are the same, or by a widening primitive conversion or a narrowing primitive conversion.

A primitive type value can be cast to a reference type by boxing conversion.

```
int i = 5; //5 is a primitive int

Integer iRef = i; // convert to wrapper

//reference type: ok
```

A reference type value can be cast to a primitive type by unboxing conversion.

```
Integer iRef = new Integer(5); //starts life as

reference type

int j = i; //ok, unboxed to primitive
```

See the topic on Autoboxing for more about moving seamlessly between primitive and wrapper class types.

Casting with Reference Types

This is very similar to casting with primitives. But it's different. And there are some rules, young lady. Deviation will not be tolerated.

Simple Casting

A reference of any object can be cast to a reference of type java.lang.Object.

You can cast up the class hierarchy, and down the class hierarchy. But you can't cast laterally. This means that a reference to any object O can be cast to a class reference that is a subclass of O if, when it was created, O was a subclass of that class.

You can cast up the class hierarchy automatically—that is without using the cast operator.

```
Object o = "some string";
```

Object is up the hierarchy because String extends Object. String is down the class hierarchy from Object. If you cast down the class hierarchy, you must use the cast operator.

```
Type t = (Type) (someObject);
```

A cast could compile and yet still fail at runtime if the cast turns out not to be legal. With the new generics facility in Java 5.0, some of this problem should be alleviated. (See the topic on Generics in *More Java Garage*.) If it turns out that a compiled class cannot cast at runtime as it hoped it could, a ClassCastException is thrown.

Subclass Casting

Let's say we have the following class hierarchy:

```
class A {}

class B extends A {}

class B1 extends A {}

class C extends B {}
```

Those definitions are legal, and you can type that into a source file. As long as one of them is public, and its name matches the source file name, it will compile. Here's what we can do:

```
A a = new B();//ok—B is subclass of A!

B b = a; //Error!! can't go up the
         //hierarchy w/o casting

B b = (B) a; //could possibly fail at runtime

b = (B1) b; //nope—lateral cast not allowed

A a1 = new C(); //ok—C is indirect subclass of A!

C c = new A(); //no—C is "narrower" than A

C c1 = (C) (new A()); //compiles and fails at runtime
```

Interface Casting

A reference to any object O can be cast to an interface reference if one of three things is true.

1. If O's class implements that interface
2. If O is a subinterface of that interface
3. If O is an array type and the interface is the interface Cloneable or Serializable

Let's say we have these classes and interfaces:

```
class A {}
class B extends A implements IInterface {}
class B1 extends B implements ISubInterface{}
class C extends B1 {}

interface IInterface {}
interface ISubInterface extends IInterface {}
```

Here's a class showing what we can do.

IntCast.java

```
public class IntCast {

public static void main(String[] args) {

IInterface b = new B();
//ok, B implements IInterface

//b = (B) new A();
//compiles, but fails at runtime
//since A doesn't implement IInterface

IInterface b1 = new B1();
//works great—B1 implements a subinterface
//of IInterface

//C c = new IInterface();
//NO! you can't instantiate an interface
//they're all abstract!

A a = new C();
//that's just fine.
System.out.println(a.getClass());
//prints C

}
}
```

Perhaps the most common version of using the interface reference type is with the Collection interface.

```
Collection c = new ArrayList();
```

Because the ArrayList class implements the Collection interface, you can do this. Then, you can make your methods accept Collection types; if later you decide you want to implement your shopping cart or whatever you're using the ArrayList for as a different kind of list, you can do that without breaking any contracts with clients of the method.

```
public void doIt(Collection c) {

//Now I can use an ArrayList here or a Stack or a
//LinkedList or...
}
```

Array Casting

You can cast an array, but the data they hold must be of compatible types. For example, this code will not compile:

```
double arr1[] = {1.5,3.14};
int   arr2[] = (int) arr1;   // compile-error
```

But arrays holding compatible types can be reassigned to a different reference type. This is all legal:

```
int[] i = {1,2,3};
int[] j = {3,4};
i = j;
java.io.Serializable s = i;
java.lang.Cloneable c = j;
```

Arrays cannot be converted to any class other than Object or String.

Arrays can be converted to interfaces only of type java.io.Serializable and Cloneable.

Primitive Widening Conversions

Different than casts, which the programmer must perform explicitly, the widening conversion happens automatically.

Widening conversions happen when one primitive type is promoted to another primitive type of greater capacity. For instance, an int can hold only 32 bits of data. A long, on the other hand, can hold 64 bits of data. That makes the following code legal:

```
int i = 10;

long lg = 20;


lg = i; //ok, lg now = 10
```

Likewise, primitive integer types are implicitly converted to floating point types, like this:

```
int i = 10;

float f = i;//ok, i = 10.0
```

If you want to squish a big bucket o' bits into a smaller bucket o' bits, you have to cast.

Every type can be converted to String, even parameterized types.

```
ArrayList<Integer> a = new ArrayList<Integer>();

a.add(9);


    //remember that passing any object to println

    //automatically calls its toString() method


System.out.println(a); //prints [9]
```

Note that in general, these conversion rules can be extended for parameterized types as well. The following, as you might guess, is illegal:

```
ArrayList<Integer> a = new ArrayList<Integer>();

a.add( 67L );//No! looking for int, found long
```

Reference Conversions

You can convert from any class, interface, or array (arrays are objects…) to an Object reference, or to null.

You can convert from any class to any interface that it implements.

You can convert from any interface to any of its superinterfaces.

For more info on array casting, see the Arrays chapter.

Narrowing Conversions

Narrowing conversions cause you to lose information about the primitive being converted. You can lose precision and magnitude data. So, the compiler requires that you explicitly cast in order to do this.

```
int i = 10;

float f = i;

i = f;//illegal without explicit cast
```

When you perform a narrowing conversion, you lose all but the lowest order bits.

You cannot convert any primitive type to null. The compiler complains of "incompatible types" if you try to do this.

You cannot convert boolean types to any other types.

I know I already said that. I don't mean to be a nag or anything. It's just, you know, maybe you're reading out of order, or didn't think that was the most fascinating thing you ever heard and needed a little reminder.

Wrapper Conversions

Any primitive to any reference is disallowed except for the following.

Primitive to String is always okay.

Primitive conversion to its corresponding wrapper class is okay, due to autoboxing, and a wrapper class to its corresponding primitive is okay (due to auto-unboxing). The following will not work:

```
Float f2 = 10;//No! 10 is an int

Long g = 20;//No, even though int is smaller than

           //long, this is the Long wrapper class
```

The int is the default integer type. The following doesn't work either:

```
Float f2 = 10.99;//No! 10.99 is a double
```

The double is the default floating point type. Both of the following work:

```
Float f2 = 10.99f;

Float f3 = 10.99F;
```

Cool.

Chapter 19. INNER CLASSES

STARRING...

- Inner Classes
- Method Local Inner Classes
- Anonymous Inner Classes
- And Special Guest Appearance by Static (Top-Level) Inner Classes as The Beav

Over-read on IM…

Zilly: can't you get him to come out? He never comes.

Elmo: he hates movies.

Zilly: well. So do I I still go :P

Elmo: yer not as internally motivated as MisterLister :)

Zilly: i'm not internally motivated period duh.

Elmo: since he got javagarage all dude wants to do is mope around his apt reading java books

Zilly: tell him theres hot chix there. then he'll come out with us.

Elmo: hot chix at de passion of de christ? uh.

Zilly: tell him hot chix with java books.

Elmo: lol…

This part is all about inner classes. Inner classes are not the kind of thing that you need to deal with much of the time. In Swing programming, where you make graphical user interfaces, you will likely need to use them. But if you go on to write Java on the server side using servlets and JSPs, or if you write a console application, you aren't likely to ever need them.

But they are important in GUIland, and that is a non-trivial land to us here. So let's check them out, see how to use them, and get on with our lives. My eyes are burning. Aren't yours?

Regular Inner Classes

Recall that you can write many different classes in one source file. You can also write a class inside of another class.

Why in the world would you want to do that? Well, they sort of allow you to scope your classes. The inner class becomes a member of the outer class. That is, the inner class acts as a member of the outer class, just like its methods and fields.

Important thing: Objects in an inner class have access to even the private variables in instances of the outer class.

You typically use an inner class to define a helper—that is, a class with a very specific purpose. In GUI development, you need to create listeners that handle events such as mouse clicks. Often, these events are handled using inner classes.

There are four kinds of inner classes that are tailored for different purposes. The first is the easiest.

It is simple business, defining an inner class. Try something like this:

```
class MyOuterClass {

  class MyInnerClass {

  }

}
```

To instantiate an inner class, you must first have a reference to the outer class. In general, an object of an inner class is created by the outer class itself.

A class can define as many inner classes as it feels like putting up with.

An advantage to using inner classes is that they can be hidden from other classes in the same package, so you can restrict access to their functionality.

Inner classes can be declared private. This is different than regular classes, which must be default or public.

Using Method Local Inner Classes

The second type of inner class is the method local inner class (MLIC). The name sez it all. We're talking about a class defined inside of a method body.

The primary difference between a regular inner class and an mlic is that the MLIC cannot use the local variables of the method that contains the inner class definition. Only in the event that local variables or arguments are declared final can an object of an inner class access them.

OuterMethodLocal.java

```
package net.javagarage.inner;

/**<p>

 * Demos Method Local Inner Class

 * </p>

 **/



public class OuterMethodLocal {

public final int x = 5;

public int outY = 10;


//the method

int someMethod() {

final int inX = 5;

int inY = 10;


class InnerClass {

//do some wicked stuff

int getLucky() {

//return inY; //NO! inY is not final!

//return outY; //OK

return inX;//OK —it's final!

}

}


//instantiate method local inner class

InnerClass inner = new InnerClass();
//call a method while you're at it

return inner.getLucky();


}//END OF someMethod


//I can't do this here—this class

//is only available inside the method!

//InnerClass inner = new InnerClass();


public static void main(String[] ar){

OuterMethodLocal out = new OuterMethodLocal();

System.out.println(out.someMethod());

}

}
```

Using Anonymous Inner Classes

Method local inner classes define a class inside the body of a method. But you can also define a class as a method parameter. The anonymous inner class is called anonymous because it is never given a name.

The key distinguishing characteristic of anonymous inner classes is that they are declared and instantiated at the same time.

The most common place to see anonymous inner classes is in GUI event handlers. The idea is to keep the code that handles an event in the same place where the control is defined. You create them when you need a class that you only need to call from one place, when one specific thing happens. Because they're defined as part of a method, they must adhere to the same restrictions that method local inner classes do.

Here is an example:

```
myButton.addActionListener(
    new ActionListener() {
        public void actionPerformed(
            ActionEvent e) {
        //do the work here!

        } //end actionPerformed method
     }//end anonymous inner class
); //end addActionListener method call
```

The addActionListener() method takes only one argument, of type ActionListener. ActionListener is an interface that listens for action events. You process action events in Swing by writing classes that implement this interface. After an object of that class is created, it is registered with a component using the addActionListener() method. That method is the same one that any class that implements the ActionListener interface would need to implement. So that is a little weird: it seems that you're making an instance of an interface, which you normally can't do. You can also do the same thing with abstract event adapter classes as well, such as WindowAdapter or MouseAdapter. It's the only place you're allowed to use the new operator on interfaces and abstract classes. That's because your anonymous class automatically becomes a subclass of that class.

The class can be anonymous because you'll never use this class again. It is okay—perhaps even desirable—that it has no name or definition outside of this context. The clicking of the myButton control is the only conceivable time you would want to invoke this code. Any other GUI control on your page is going to have to define its own action handler.

Static Inner Class

Inner classes can be static, meaning that they have access only to the static member variables of the outer class. The reason is that they don't share much of a relationship with the outer class—they're tied to the name of the outer class, but not an instance of the outer class. A static nested class can be instantiated without an instance of the outer class. You can access it just like any static member variable. These are also known as top-level nested classes because they really boil down to a way to control namespace.

For example, in C#, you can define arbitrary namespaces that have no correspondence to actual directory structure (not so in Java), and you can nest namespaces within a class or a bit of code almost willy-nilly (again, not so in Java). So, the static nested class lets you get a similar level of namespace control. To which I say, "Big deal."

So, the static nested class has a life all its own, without an outer object enclosing it, and is finally free to pursue its amateur ham radio license. I have to say, though, that these things ain't used much.

StaticNestedClass.java

```
package net.javagarage.inner;

/**<p>

 * Demos static inner class usage.

 **/

public class StaticOuter {


    //an instance of inner object cannot

    //directly reference this, nor can its methods

    int outerVar;


    //an instance of inner object cannot

    //directly reference static members—

    //BUT the members

    //(methods and fields) defined inside the

    //StaticInner class CAN

    public static int outerStaticVar;


    //Here is the Static inner class, defined

    //like a member of the StaticOuter class

    public static class StaticInner {


   int innerVar;


    //The StaticInner class methods can
    //access only the static members of
    //the enclosing class
    public int getOuterStaticVar() {
        return outerStaticVar;
    }
}//end StaticInner

public static void main(String[] args) {

    //this statement creates a new instance
    //of the StaticInner class ONLY
    StaticOuter.StaticInner inner =
                  new StaticOuter.StaticInner();

    inner.innerVar = 5;

    //Instantiating the outer class does
    //NOT give us any instance of the
    //StaticInner class
    StaticOuter outer = new StaticOuter();

    //yeah, yeah outer can reference its
    //OWN members per usual...
    outer.outerVar = 10;
    StaticOuter.outerStaticVar = 20;

    //Wrong! I can only access this guy
    //from WITHIN the methods of
    //the StaticInner class
    //inner.outerStaticVar = 20; //wrong!!

    //prints 20.
    System.out.println(inner.getOuterStaticVar());

  }
}
```

The methods of a static inner class can access only the static members of the outer class.

So that ends our adventure through the fascinating land of inner classes. They are often useful when applied in conventional ways, such as implementing an actionPerformed method for a JButton. Just be careful not to overuse them and possibly introduce unnecessary complexity and confusion in your code. As my friend Erin from Indiana used to say, "eschew obfuscation."

PREV < Day Day Up > NEXT

Chapter 20. BLOG: INNER CLASSES AND EVENT HANDLERS

Beware when using inner classes, anonymous inner classes in particular. They are the subjects of much debate within the Java community. Developers who don't like their use say they break encapsulation—one of the key principles of object-oriented programming—and that they're hard to read and difficult for beginners to understand.

Developers who promote their use during late night local television shows point to their convenience and elegance. You need to know about them to read and write Swing application code. But in general, you don't need them, and your code is probably clearer and more flexible without them.

My rule is this: Use anonymous inner classes to handle actions and events. Don't use the other kind of inner classes. I think that's where the true confusion can come in, given how inner and outer classes share their members.

But anonymous inner classes solve the problem of the Monolithic Switch Block—an antipattern familiar to coders throughout the land. Come on. I know you've done it. The first step to getting help is admitting you have a problem….

Whenever you see code like this, where the frame itself implements the ActionListener interface….

```
public class MyGUI extends JFrame implements

          ActionListener {

   protected JButton btnSave;

protected JButton btnCancel;

//and so on

//have to implement this guy because he

//showed up with the ActionListener interface

   public void actionPerformed(ActionEvent event) {

String command = event.getActionCommand();

//use giant if/else block to determine

//who fired this event

      if (command.equals("Save...")) {

         // do something:

handleSave();


      } else if(command.equals("Cancel")) {

         //and so on...


private void handleSave() {

//do work here

}
```

…you know there's too much Liquid Drano in the code designer's latté. This is not good object-oriented design. In general, this kind of construct can grow fairly large, and changes in one place in the code often require changes in another part of the code. Moreover, if/else blocks require the runtime to test the conditions, and this means that your app could slow down the larger it gets.

Why run a bunch of tests you don't have to?

The same deal implemented with an anonymous inner class would look like this:

```
//look ma! no "implements ActionListener"

//on the class!

public class MyGUI extends JFrame {

createGUI() {

//create the buttons

   JButton btnSave = new JButton("Save...");

JButton btnCancel = new JButton("Cancel");


//add action handlers

btnSave.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

//do something for save

}

});


btnCancel.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

//do something for cancel

}

});

//...
```

Anonymous inner classes solve some problems: The code is always where it should be, and you'll get better performance. It is probably polite to seriously limit the amount of code you put into an anonymous inner class if you can. Of course, you gotta do what you gotta do. But if you are going to put a lot of code into an event handler, you should just collect the data in the ActionListener, and then start a new thread inside it to do your work. Otherwise, your GUI will be frozen waiting for your ActionListener to get done.

Following is an example of how to start a new Thread in your ActionListener, which makes your GUI more responsive. It also delegates the actual work of the action to a handler, and does not attempt to do all of the work in the GUI class itself.

```
JButton btnSearch = new JButton("Search");

btnSearch.addActionListener(new ActionListener() {

    public void actionPerformed(ActionEvent e) {

      Runnable runner = new Runnable() {

        public void run() {

                  try {


ArrayList results = searchController.search();

    createAndShowResultsPanel(results);

           } catch (BandSException bse) {

             //handle

           } catch (Exception ex) {

             ex.printStackTrace();

             //handle

           }

         }

       };


       Thread t = new Thread(runner);

       t.setName("Search Thread");

       t.start();

     }

  });
```

We call a controller (which I assume above we already have a reference to) as a delegate to do the work. This is the way you will usually want to handle button clicks and such—it separates the GUI (code that creates the user view) from the business logic.

Chapter 21. HANDLING EXCEPTIONS

DO OR DIE:

- Find out about exceptions.
- This is all about exceptions.

Exceptions

The two lean men sat coiled, perched across from one another, their boots stained with calves' blood. Their hands were rough, like raw leather. The man facing the bar's door dug his knife further into the raw pine table. The man with the moustache eyed the twisting knife only half afraid.

Moustache: Tell me again.

Knife: There's two kinds.

Moustache (shoving the chair out from under him): I know that! Don't you think I know that already?

Knife (stretching back his arms in a peacock display of relaxation—just enough to reveal the six shooter under his arm): Two kinds. Checked. And unchecked.

Moustache: And it is something that happens—

Knife: An *event*.

Moustache: Yes, an event that can occur during the execution of code.

Knife: It's not just any event. It's the worst kind of event. It could mean the end…

Moustache pales in horror. The men slug their near-frozen beers.

Moustache (eyes suddenly alight with fire, he pounds both fists on the table): You will tell me about them! Tell me about these exceptional events…….

Toolkit: A Custom Exception

ApplicationException.java

```
package net.javagarage.demo.exceptions;

/**<p>

 * This class is a subclass of java.lang.Exception,

 * and can be reused to represent any application-

 * specific exceptional state. For instance, if you

 * are using a database, it is not a good idea to

 * allow database exceptions to bubble up to the

 * user, or to require handling by more than one

 * layer in your application.

 * <p>

 * For these reasons, you typically want to make your

 * own exception wrapper that will serve to represent

 * application-level exceptional states. For example,

 * an online store might have a CartException that

 * gets thrown explicitly if the user tries to check

 * out with no items in their cart, or tries to

 * specify a quantity of less than 0.

 * </p>

 * <p>

 * Note that it is good practice to actually extend

 * the functionality of a class that you extend. That

 * is, you should add a method or two that is

 * specific to your situation.

 *

 * @author eben hewitt

 * @see java.lang.Throwable

 * @see java.lang.Exception

 **/

public class ApplicationException extends Exception {

private int status;
```

```
/**
 * constructor no-args
 */
public ApplicationException() {
super();
}

/**
 * Overloaded constructor Exception wrapper—anything
 * that implements the Throwable interface
 */
public ApplicationException(Throwable t) {
super(t);
}
/**
* Overloaded constructor with message describing the
* exceptional state
*/
public ApplicationException(String message) {
super("ApplicationException: " + message);
}

/**
 * Overloaded constructor with message and
 * wrapped root cause
 */
public ApplicationException(String message,
      Throwable cause) {
super(message);
initCause(cause);
}

/**
 * Overloaded Constructor with int message if your
 * underlying layer (such as a database) returns some
 * meaningful integer status such as -1.
 */
```

```
public ApplicationException(String message,
    int status) {
this(message);
this.status = status;
}

/**
 * Sets the status message
 */
public void setStatus(int status) {
this.status = status;
}

/**
 * Gets the status int.
 * @return int The status number
 * representing some state
 */
public int getStatus() {
return status;
}

/**
 * Gets the message from the root cause
 * @return String The root cause message
 */
public String getCauseMessage() {
if(getCause() != null) {
return getCause().getMessage();
} else {
return "Cause unknown";
}
}

}
```

This custom exception features a number of overloaded constructors. It is nice to provide a few overloaded constructors, because it gives the user of your API choices about how to interact with it.

Now that we've seen how to define a custom exception, let's look at how to use one. Check out ExceptionClient.java. His job is to run some code that will cause the custom exception to be thrown.

ExceptionClient.java

```
package net.javagarage.demo.exceptions;

/**<p>

 * ExceptionClient is simply for testing. It is a

 * main method that calls two static methods to

 * demonstrate how to use custom exceptions. The

 * custom exception we use is ApplicationException.

 * </p>

 * @author eben hewitt

 * @see

 * net.javagarage.demo.exceptions.ApplicationException

 **/

public class ExceptionClient {

/**

 * In your business method, just throw exception

 * wrappers. That way, you can keep your code clean,

 * flexible, and easy to maintain.

 * @throws ApplicationException

 */

private static void testWrapper() throws ApplicationException {

      try {

            //do something that will generate an exception

            //you don't want to proliferate

            int[] nums = {1,2,3};

            int x = nums[5]; // ERROR! out of bounds!

      } catch(ArrayIndexOutOfBoundsException obe){

            throw new ApplicationException(obe);

      }

}


/**

 * Test explicitly throwing an exception when

 * something bad happens. Say we have some business
 * rule where you aren't allowed to take more than 10
 * consecutive holidays. We want to generate an app
 * exception if someone tried to.
 */private static void testExplicit(int x)
throws ApplicationException {
    System.out.println("You are asking for " + x +
    " days off.");
if (x > 10){
    throw new ApplicationException(x +
    " is TOO MANY DAYS!\nYou will be beaten.");
}
//only prints if the exception is not thrown.
//if it is thrown, execution jumps out of the method
//and up to where it is caught in main()
System.out.println("Have a nice time!");
}

/**
 * Shows two different uses of your custom exception.
 * Obviously, we cannot test both of these at once,
 * because once an exception is printed,
 * execution stops.
 */
public static void main(String[] args) {
//to demonstrate usage
try {
//1. test wrapping API exceptions
//testWrapper();

/*
 * calling (1) produced this output:
 * Message: java.lang.ArrayIndexOutOfBoundsException: 5
 */

//2. IF user does something bad, you want
//to send customized notice of it
testExplicit(5);//okay

testExplicit(100);//EXCEPTION!
} catch(ApplicationException ae){

System.out.println("Message: " + ae.getMessage());

}

}

}
```

Here is the output result of running ExceptionClient.java as is:

```
You are asking for 5 days off.

Have a nice time!

You are asking for 100 days off.

Message: ApplicationException: 100 is TOO MANY DAYS!

You will be beaten.
```

Looking at the code, you'll notice that there is a second method defined, called testWrapper(), that catches an ArrayIndexOutOfBoundsException and simply takes that entire exception and passes it as the constructor of our custom exception. We can do that because ArrayIndexOutOfBoundsException extends IndexOutOfBoundsException, which extends RuntimeException, which extends Throwable. Whew.

Exceptions and Inheritance

Look. Let's be honest. I don't like this anymore than you. I just want to go go go. Maybe get a truck and drive. Out on the open road. I look out at the pool and it looks fantastic. I know that it's ice cold. I know it is a burning freeze of water. But it looks so serene and inviting. Maybe I could like being a giant block of ice. Drop in and freeze and then bob up and down, forever, like a cat dangling its tail on a fence, lazy, slow, back and forth. This clock wouldn't tick anymore. In a meaningful way, for me, anyway.

To:=?big5?q?JavaGarage=A4=A4=A4=E5=A5=CE=?=<java-mid.ea-@lists.javagarage.dude>

Subject: excpetion inheritance

From: "Shoeyong" <shoeyong@sonuvagun.com>

Date: Wed, 15 Dec 2003 22:47:49 +0800

Message-id: <002941c18965$4cdf7666$0100a8c0@uyong>

Old-return-path: shoeyong@sonuvagun.com

Sender: Totally Fokd <sauroman@evilempire.org>

I have come to an impasse in my evil plan to rule the world. I need to know about inheritance and exceptions.

OK. What's the name of your box?

"Boomdeay". What difference does that make?

None. I just like to know the names of guy's boxes. What is el problem?

How does it work, this thing you call method overriding and the exceptions methods throw?

Are you on something dude? An overriding method cannot be declared to throw checked exceptions EXCEPT those that are declared in the superclass method OR subclasses of those checked exceptions.

Example pls?

Say you have a super class that defines

```
void myMethod() throws FileNotFoundException...
```

Then all of these alternatives in the subclass are ok:

```
void myMethod() throws FileNotFoundException
```

the following are all NOT ok:

void myMethod() throws PsychoticBreakException //not ok: this is not in the IOException hierarchy

void myMethod() throws FileNotFoundException, EOFException //not ok: the overriding method cannot throw methods that aren't thrown by the overidden method

void myMethod() throws IOException //not ok! IOException is higher up the inheritance hierarchy than FileNotFoundException, which the superclass throws

Explain it to me. This is very good…

IOException is the mama exception up the hierarchy tree of exceptions having to do with input/output ops. EOFException (End of File), and FileNotFoundException, for example, extend IOException. FileOutputStream constructors throw FileNotFoundException.

Let's say it again in a slightly different way, since it can be confusing at first.

So say you have this method "someMethod" in superclass "ClassA".

someMethod() throws IOException.

"ClassB" extends ClassA.

Now you override someMethod() in ClassB. What can it do? What can it not do? What must it do?

It can declare that it throws IOException, just like the superclass method.

It CANNOT throw *more* exceptions like this: someMethod() throws IOException, PsychoticBreakException

It CAN throw *fewer* exceptions, like this: someMethod() { } //no 'throws'.

Step right up and test your deductive reasoning skills!

True or False: If a superclass method throws no exceptions at all, an overriding method in a subclass may not throw any exceptions either.

True!

Now if you are familiar with C#, exceptions are different in Java. You are not forced in C# to handle exceptions that you aren't interested in. Java forces you to do that. So it adds a little more code, but it keeps your API exposed and documented as it should be, and makes your program more easily readable.

Cheers,

TF

This message was re-posted from java-chinese-big5@lists.javagarage.dude and converted from gb231 to big5 by an automatic gateway. Ha ha ha ha.

Exception Handling Pretty Good Practices

This is a list of things that are a good idea to do when writing code in the real world (he writes, as if there is any other world). I don't know if they are *the best* practices ever. That's pretty hard to determine concretely.

- **Do not squash exceptions—do something useful with them**. It is common to see code that catches an exception, and then does nothing. The following will compile:

  ```
  ...catch(Exception e) {}.
  ```

  Usually it isn't quite that angry, and you see a call to System.out.println(). If your application is a console application, this might be appropriate, but would it be better to save that message to a log file along with a timestamp, class name, and username? You bet. If you are writing a desktop application using the Java Swing API, you could still do that, but in addition, trying passing the exception.get-Message() or exception.getCause() to a JOptionPane.showMessageDialog(). It takes two parameters: the parent component (it can be null) and the String message to display. This is like a MsgBox in VB or a JavaScript alert.

- **Catch exceptions with the narrowest type match practical**. That is, design your program keeping in mind that one person's program is another person's API. If it matters to make a distinction between FileNotFoundException and IOException, make that distinction and handle them differently in different catch blocks. More specific is usually better. It's more complete and explicit, and easy to understand.

- **Don't proliferate checked exceptions into your API**. That means be careful if you find yourself writing loads of methods that throw everything. The reason is this: It makes your API contract more brittle, and you might find your client code becoming overburdened if the client can't do something meaningful with the exception. You don't have checked exceptions in languages like C++ and C#, so programmers coming from those languages may find it a bit of a hassle.

- **Don't throw an exception when you can perform a test**. They are meant for exceptional situations, and in all likelihood will cause your code to execute more slowly. A lot of times you can perform a test that saves you from the trouble, time, and reduction in readability that adding exception handling code causes. For example, say you have a form allowing a user to log in. If the password is left blank, you don't need to throw an exception. This isn't really exceptional. Users do dumb things all the time, kind of at a breath-taking pace actually; it's like there's a race with a big important prize if you can be the dumbest user. But if you can do if (password != null)…that's way better (simpler, clearer, faster) than throwing an exception about that little oversight.

- **Reasonably organize your try blocks with related statements**. That means that you shouldn't make a separate try/catch block for each item that throws an exception. Many situations do not call for atomic transactions. Typically, you don't just do one thing in a vacuum, and then do something else totally unrelated to what you just did. You open the file because you want to write to it. You build the URL because you want to connect to it. And then you probably want to read the stream, and do something with that. Sheesh, there's a lot of stuff to do in the world. These things go together. If you didn't build the URL, you don't have much chance of opening that connection, do you? It is okay (good) to put these statements together in *one* try block, and then deal with whatever problem arises in a catch or two. The lesson: do not write one try block per one potentially-checked-exception-throwing-method if it makes sense to group them together.

- **Keep your code encapsulated**. As mentioned earlier, you will hopefully design your applications in tiers, your user interface tier separated from your business logic tier, separated from your persistence tier, and so forth. Do not allow exceptions that are specific to the implementation in one tier bubble up to another. That is, do not allow a SQLException up in your client. Because then your client has to deal with it, and it means that it has been thrown up through each of the proceeding tiers. If you change your persistence layer to an XML file or some financial system or something other than a SQL-compliant database, you won't be throwing SQLException anymore, and yet you will have broken the encapsulation that is the very purpose of layering your code in tiers in the first place. To make matters worse, the API is outdated and clogged up with unnecessary code. The solution is easy: Wrap exceptions in code that makes sense for the tier on which it will actually get handled. Do not let this confuse you, though. It is just fine to propagate exceptions, because often a higher layer is the one you want to present the problem to the user. Just do it smart.

And so ends our business with exceptions. I hope that you have enjoyed it as much as I have. See you next time.

What an Exception Is

An exception is some bad news that you get told about during the execution of your program. There are many reasons for this. Your database could be down, and then the application that is trying to connect to it would not be able to do what it wants. Network problems are common causes of exceptions. Your program could rely on a file that cannot be found, or a user could enter something that is just ridiculous. You have to deal with these problems because they happen.

An exception is one result of poorly written code. If you write quick and dirty code, you should expect to have problems later. A good example of this is, oh, anything written in VB (which explains the comparatively large number of VB programming jobs on the planet).

An exception is different from an error. An error is something that you can't do much about. That is, your program probably would not be able to recover from an error at all. This is something like you're completely out of memory. In some circles, it is considered pedantic to make a distinction between exceptions and errors, and you'll commonly see the two terms used interchangeably. In the interest of preserving what little is left of our collective sanity, let's just call it six of one, half dozen of the other.

An exception is something that you can create to take care of an ailing situation, and reroute the flow of the application.

An exception is something that you *throw*, or that the runtime itself throws. Throwing means that you make it available again to another part of the program, make it bubble up the call stack—it means you.

An exception can be part of the method signature. You can write, "I know that I'm doing potentially dangerous stuff in this method, but I do not want to deal with anything bad that happens. I am going to pass the buck back up to the method who called me." Here's how you do that:

```
  public void someMethod(int someID) throws
      SQLException { ...
}
```

An exception is an instance of a java.lang.RuntimeException or java.lang.Error, or an instance of a class that extends one of them. Runtime exceptions are also called *unchecked exceptions*.

An exception that you create is an instance of a java.lang.Exception, which implements the java.lang.Throwable interface.

When an exception occurs, you should deal with it if you can.

Dealing with Exceptions

At nearly any point in your code, you could just do the following:

```
throw new RuntimeException();
```

And off your program goes, to the next block of code in your app that is prepared to handle such a statement. That is definitely changing the flow of your application abruptly.

What do you do when an exception crops up? You can do one of two things: you can deal with it or you can make someone else deal with it.

To deal with it, catch the exception in a catch block and do some real work to handle the situation. Perhaps this just means printing a user-friendly message to the screen so that the user knows what is happening and can notify someone. It could mean logging the problem and sending an e-mail—whatever is appropriate for your application. You can do that like this:

```
try {

// code that might throw an exception

} catch (Exception e) {

// handle the exception here

}
```

My dad used to tell me that if I didn't take care of my own business, somebody else would. I think that's true. I know it's true in Javaland. If an exception is never caught by you, it is caught by something called the default handler. Just like, you don't want somebody else taking care of your own business, you don't want the default handler handling your exceptions. If it does, it will halt execution, shut down the VM, and print a giant error message and stack trace to the screen.

But we can cause an exception in one method, ignore it, and then the exception will bubble up to the calling method. This is the behavior all the way back to the main() method that started your program.

Try not to let yourself get confused by the name RuntimeException, despite the fact that it does not distinguish this method from any other. That is, any exception that you would deal with inside a try/catch block would happen at runtime.

Built-in Exceptions

The Java programming language comes with many exceptions built into the language. These cover many different types of common, unfortunate situations. For example, ArrayIndex-OutOfBoundsException is popular. It is thrown to indicate that you have attempted to access an index that does not exist in an array. The system will throw this exception if you have an array of 4 elements, and you type x = myArray[100] (no 100th element). Or if you type x = myArray[-1] (cannot be negative value). Or if you type x = myArray[4] (there are only elements 0-3 in a four element array). This is a good example of the kind of thing that can happen easily, but from which it also should be easy to recover.

UncheckedExceptions

The ArrayIndexOutOfBoundsException is not something that the compiler will force you to plan for when you are writing your application. If you write some code that works with an array, the assumption on the part of the language designers is that you will be careful enough about what you are doing that you should not need to write a complete try/catch block every time you access an array. If you *do* get an ArrayIndexOutOfBoundsException, it is almost certainly because you have some poorly written code in there that needs permanent fixing.

Because the compiler doesn't check if you are handling the possible occurrence of this kind of exception, it is called an unchecked exception.

It means that you can compile the following code:

```
int x = myArray[i];

//look, ma—no try catch statement !
```

NullPointerException is another unchecked exception that you run into during development. Try typing the following:

```
Object obj = null;

obj.toString();
```

You can't call the dot operator on null. Your program blows up. The language designers know that you know this, and they don't want you having to clutter your code with tons of handling statements. So, NullPointerException is unchecked.

If you try to divide integers by zero, the virtual machine will generate an ArithmeticException. Again, you are trusted not to do this, and therefore spared the red tape and inconvenience of planning for it.

You are not required to handle any exception that is a RuntimeException, or one of its subclasses.

CheckedExceptions

Checked exceptions, on the other hand, must be dealt with. The compiler checks to see that you have at least something that looks like it will handle exceptions any time you invoke an operation that could generate a checked exception.

An IOException, a MalformedURLException, and a SQLException are all good examples of checked exceptions.

You cannot compile a class that contains this code on its own:

```
File f = new File("s");

f.getCanonicalFile();
```

If you try to compile the preceding code, your compiler complains that you have an unhandled exception lurking in there. Something inside that code block declares that it throws an exception of type IOException, and so you need to deal with it. The following will fix it:

```
public void someMethod () {

    File f = new File("s");

    try {

        f.getCanonicalFile();

    } catch (IOException ioe) {

       //do something here

}

}
```

The getCanonicalFile() method of the File class might throw an IOException because it is possible that the call would require reading the file system. The creation of the new File object does not need to be wrapped inside the try/catch block because it just creates an object of type File—it doesn't actually do anything on the file system. So, what you're doing is no more potentially problematic than the following:

```
String s = "well, hello Mr. Fancy Pants";
```

The compiler cannot judge the quality of your plans to handle the potential IOException. In fact, your catch block can be empty—you can write no code at all between the curly braces. But you have been forced to acknowledge a potential danger.

Throwing Exceptions

Your method might not be the very best place to handle an exception. It might make more sense to let the exception pass through your method unhandled, and make some other guy deal with it. You can do that. But you have to say that that is what you're doing. You do so with the throws keyword. The following modified method will also compile and run:

```
public void someMethod () throws IOException {

    File f = new File("s");

    f.getCanonicalFile();

}
```

Do not declare that your method throws unchecked exceptions that inherit from Runtime-Exception. Do not write the following:

```
void do(int i) throws NullPointerException

   //BAD DOG! NO!

{

   //blah blah blah

}
```

Why should you never do that? Sometimes the truth is tough. If you are that worried about your code generating a NullPointerException, you probably have an idea of where the problem is, and you should do something to deal with it at that point. Remember that because unchecked exceptions mean that you are trusted, don't advertise to the world with a great big throws sign, "Hey! Look at me! I wrote some crappy code! I might divide integers by zero!" It's just unseemly. Alternatively, sometimes unchecked exceptions occur because of things that are totally out of your control. I say, "If there's nothing you can do about it, there's no point in talking about it."

One other note: You can throw multiple exceptions by separating them with a comma when you declare them. Like this:

```
public void myMethod() throws FileNotFoundException,

    EOFException
```

Of course, it is legal to write something like the following:

```
private void myMethod() throws IOException,

FileNotFoundException...//Don't do this!
```

But don't do that. Because it is redundant. IOException is bigger (higher up the inheritance hierarchy than FileNotFoundException), and you might as well just deal with what you should deal with, and stay on track.

You can also throw an exception using the new keyword if you want to reroute the flow of your program. You do so like this:

```
public void calculateBillTotal(float bill,

                    float tax,

                    float tip)

          throws CheapskateException {

if (tip < bill * .15)

    throw new CheapskateException(

        "Fork it over, cheapskate");

return bill + tax + tip;

}
```

In the preceding code, if the tip is less than 15% of the bill, the exception is thrown up to the caller. The return statement that adds the parts of the bill together will never be executed.

Note one other thing. Sometimes you have to say stuff like this, just to make sure. There's a nut job in every crowd: *you can't throw anything that ain't throwable*. That is, don't try the following, smart guy:

```
public static int myMethod throws String

//you know better
```

How would you like it if people came to *your* house and threw a JFrame? Somebody could get hurt on one of those things.

> *"Is plastic okay? Maybe. Maybe nobody can say for sure…"*
>
> *—J. Benton, 1.16.04*

Catching Exceptions

In the preceding examples, all of the methods throw an exception up to the caller. But you catch exceptions to handle them in your code.

You can declare one or more catch blocks for each try block. There can be no code between the end of the try block and the beginning of the catch block.

Inside a try block, if your code generates an exception, processing stops immediately, and the runtime checks each subsequent catch block to match the exception type declared with the exception type thrown. Upon finding a match, it enters the catch block and executes whatever handling code is in there. Like this:

```
void someMethod() {

   throw new UnknownHostException();

   System.out.println("Uh-oh!");

   //this line will never be printed.

} catch (UnknownHostException uhe) {

     System.out.println("I will be printed"); //ok

}
```

Code inside a catch block may throw exceptions. Like this:

```
void someMethod(){

try {

    //connect to a database here

    //a SQLException happens. Oh no.


    throw new SQLException("Things are not

              what they seem");

} catch (SQLException sqle) {

    //code here to tell the user about the problem

    //with their query


    //Since there was a problem with the database,

    //let's shut down the connection to it.

    //But guess what—closing the connection throws a
    //SQLException too!

    //so we have to do it again...

    try {

        connection.close();

    } catch (SQLException se) {

        //handle the potential

        //connection.close() exception

    } // end inner catch

} // end outer catch

} // end method
```

Code inside a catch block can be well-employed to clean up any resources such as database connections or file streams that should be closed if an exception is thrown. Consequently, catch blocks can also contain try/catch blocks.

If you define a try/catch block, and none of the code in the try block generates an exception, the code inside all catch blocks is completely ignored.

You can rethrow an exception after you catch it. The following is okay:

```
try {

    throw new KittyException();

} catch (KittyException ke) {

    throw new DogException("My dog message. " +

ke.getMessage());

}
```

Using Finally

The finally keyword is used after a try/catch block to indicate that the code inside the finally block should be executed whether the code inside the try block generates an exception or not. No matter what happens (unless someone pulls the plug on your box or externally kills your JVM process), the code in your finally block will run.

Do it like this:

```
try {

...

} catch (SomeEx se) {

 ...

} finally {

 //put the code you want to run to matter what here.

}
```

Note that although the preceding code is how you will see it used 99% of the time, you can use the finally block *without* a catch block.

The finally statement is very useful, especially for doing complex operations involving files, a database, or a network connection, because there can be a lot to clean up if anything goes wrong (or even if it goes right!).

However, be careful of one subtle use with the use of finally. If your code throws an exception *other* than the one you are catching, and enters a finally block, that original exception will be totally lost if you throw a new, different exception inside your finally block. That is a mouthful. Or a mindful. Or whatever. Look at this:

```
InputStream in;

try {

   //do something with in

} catch (IOException ioe) {

   JOptionPane.showMessageDialog(null, "Error! " +

ioe.getMessage());

} finally {

     try {

         in.close(); //could throw

     }catch (IOException ioe) {

         // this hides the original exception

         }
}
```

Note that the finally clause will not execute if the catch block calls System.exit();.

So a finally clause is a good thing, and can help keep you from duplicating code.

Different Ways of Handling Exceptions

Let's look at a short demo class that allows us to see everything in one place. This example shows different ways of working with code that could generate exceptions. You might not be familiar with working with Java's networking libraries or database libraries. It doesn't matter here. What matters is noticing the different ways that exceptions can be used in your code.

```
package net.javagarage.demo.exceptions;


import java.io.IOException;

import java.io.InputStream;

import java.net.MalformedURLException;

import java.net.URL;

import java.sql.*;

/**

 * This class demonstrates different ways to write

 * your methods when you need to deal with exceptions

 * in them. Any one of them could be most appropriate,

 * depending on the situation.

 *

 * @author eben hewitt

 */

public class DemoExceptionDeclarations {


  //ONE THROWS. this method does an network

  //operation, but doesn't want to deal with it.

  //so pass the buck...

public static InputStream getDocumentAsInputStream(URL url)

throws IOException {

InputStream in = url.openStream();

return in;

   }

  //NO THROWS, MULTIPLE CATCH. doesn't throw

  //anything. it deals with the exception itself.

  //(This method must be the first born.)

  private void myNetMethod() {

  URL url = null;
  try {
//if this throws an exception, we jump down to
//MalformedURLException
  url = new URL("http://javagarage.net:80/index.html");

  //if this throws an exception, say, because of
  //network problems, or because there is no resource
  //at the URL, then processing jumps to the IOException
  url.openStream();

  } catch (MalformedURLException mue) {
  //getMessage() tells you details about what happened
System.err.println("MalformedURLException: " +
mue.getMessage());
//prints the stack trace to the standard error stream
mue.printStackTrace();
  } catch (IOException ioe) {
  System.err.println("Could not open stream to
             URL " + url);
  }
  }

  //CATCH NESTED IN FINALLY. this method does
  //operations that make her catch
  //multiple exceptions. that's okay, just add
  //multiple catch blocks. Make sure that your catch
  //blocks go from narrowest to widest
public void databaseSelect(int idIn) {
ResultSet rs = null;
Connection connection = null;//get connection

try {
PreparedStatement getStuff;
String s = "SELECT itemName, price FROM Products
            WHERE id = ?";
getStuff = connection.prepareStatement(s);
getStuff.setInt(1, idIn);
rs = getStuff.executeQuery();
//do something with result set here.

//if the above code generates an exception of type
//SQLException, this catch block runs
} catch (SQLException sqle){
System.err.println("Could not execute query:\n" +
sqle.getMessage());

//NESTED TRY/CATCH
try {
rs.close();
} catch(SQLException se){
System.err.println("Could not close result set.\n" +
se.getCause());
}

//if the code above in the first try block of the
//method generates any other type of exception,
//this will catch it, since Exception is the parent
} catch (Exception e){
e.printStackTrace();

//whether or not any exceptions were thrown above,
//this code will execute no matter what
} finally {
try {
//clean up after ourselves—but this also throws
//SQLException, so we will just deal with it here.
connection.close();
} catch(SQLException se){
System.err.println("Could not close connection to DB.\n" +
se.getCause());
}
}
}
}
```

Each method in the preceding class shows a different way of dealing with exceptions, and should serve as a reasonable guide as you write your own exception handling code.

Wrapping an Exception

Well-designed applications are often designed in tiers. Perhaps you have a data tier, and all of the objects of a similar type in a similar package are dedicated to accessing the database, and returning the result of the operation up to another tier of the application.

The purpose of the tiers is to separate different functions. N-Tier development dictates that you don't mix your user interface code with your workflow code or your data access code. It stands to reason that you do not mix an exception generated in the data tier with client tier code.

But you *do* need to handle the exception. So it's a predicament. Well, the thing to do is wrap up your SQLException object in an exception object that hangs out with the client tier crowd.

Let's take a look at an example of a custom exception wrapper for doing this very kind of thing.

Chapter 22. FILE INPUT/OUTPUT

STUFF WE'LL DO:

- Get ahold of the java.io package
- Learn how to perform useful File and Directory operations
- Read data from files
- Write data to files

Files and Directories

The first thing to note when dealing with files and directories in Java is that both files and directories are objects of type file. Don't look for java.io.Directory, cuz it doesn't exist.

The other confusing thing about working with files in Java is that when you create a new object of type java.io.File, you do not actually create a physical file on the file system. You have only created an object that can be *used* as a file. There are many classes and methods useful for doing this, which we'll look at in a moment, after we get used to dealing with File objects.

Fields

There are a number of fields in the java.io.File class that are useful when working with files and directories:

- static String pathSeparator The system-dependent path-separator character, represented as a string for convenience. It's : on Unix and ; on Windows. This corresponds to System.getProperty("path.separator");.
- static char pathSeparatorChar The system-dependent path-separator as a character.
- static String separator The system-dependent default name-separator character, represented as a string for convenience. This is \ on Windows and / on UNIX.
- static char separatorChar The system-dependent default name-separator character.

The following command gets the location in the file system from where the java command was invoked:

```
String pwd = System.getProperty("user.dir");
```

Creating a Directory

Using the mkdir() method will create a directory for you. You must use the mkdirs() method (with an "s") if you need to create all non-existing ancestor directories automatically.

```
//create dir without ancestor dirs
    boolean isCreated = (new File("directoryName")).mkdir();
    if (!isCreated) {
        // dir not get created
    }
```

Note that this manner of creating a directory does not create any necessary ancestor directories. That is, if you try to specify a directory like /usr/eben/downloads/tomcat/ and the downloads directory does not exist, the mkdir method will not create it for you, even though it is a requirement for creating the tomcat directory. So it does nothing but return false.

The following code shows how to remedy that situation:

```
//create dir and all ancestor dirs

   boolean isCreated = (new File("directoryName")).mkdirs();

   if (!isCreated) {

       //dir not created

}
```

Deleting a Directory

Deleting an *empty* directory is done with the delete() method called on the File object representing the directory you want to delete. It returns a boolean indicating whether the directory could be deleted.

```
//this will delete an empty directory

boolean isDeleted = (new File("directoryName")).delete();

    if (! isDeleted) {

        // could not delete dir

    }
```

If there are files or directories in the directory you want to delete, you must recursively delete each one.

```
/**

 * Deletes a directory recursively. If the directory

 * isn't empty, you have to first delete all

 * subdirectories and files underneath it.

 * <p>

 * Note that this guy takes a file object instead of

 * String because he calls himself (that's the

 * recursive part). It prints out what it is doing
 * as it does it.
 * <p>
 * Just call <code>new File("dirName")).delete();</code>
 * if you know it is empty.
 * @param directory The directory to delete.
 */
public static boolean deleteDir(File directory){
    if (directory.isDirectory()) {
        //get array of strings of contained
        //files and directories
        String[] subdirs = directory.list();

        //loop over entire array
        for (int i=0; i < subdirs.length; i++) {
          //call this same method, but create the
          //instance with the name of the subdir
            File f = new File(directory,
            subdirs[i]);
            deleteDir(f);
            log("Deleted " + f);
        }
    }
}
```

Files

The following class demonstrates several different methods of the java.io.File class. These include getting the size of a file, renaming a file, and moving a file to a different directory. It also gives you some utility methods to do things you might commonly need to do, such as get the file size in bytes.

CommonFileTasks.java

```
package net.javagarage.demo.io.files;

import java.io.File;
import java.io.IOException;
import java.text.DecimalFormat;
import java.text.NumberFormat;

/**<p>
 * This class demonstrates a number of
 * typical things you need to do when working
 * with files.
 * </p>
 * @author eben hewitt
 * @see File, NumberFormat
 **/
public class CommonFileTasks {

private static final String ps = File.separator;

//enum new in 5.0 classname
//the values will be automatically populated in order
//starting with 0 if you don't specify them

public enum Size {

BYTES, KILOBYTES, MEGABYTES;
}
```

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

String aFile = "C:\\mysql\\bin\\mysqld-nt.exe";

log(getFileSize(aFile, Size.BYTES));
log(getFileSize(aFile, Size.KILOBYTES));
log(getFileSize(aFile, Size.MEGABYTES));

String bFile = "C:\\test.txt";

createFile(bFile);
fileExists(bFile);
renameFile(bFile, "C:\\test2.txt");
deleteFile(bFile);
moveFile("C:\\test2.txt", "C:\\backups\\");
}

/**
 * Creates a file if it doesn't already exist.
 * @param fileName Name of the file you want to create
 */
public static void createFile(String fileName){
try{
File file = new File(fileName);

//creates file
boolean success = file.createNewFile();
if (success) {
log("Created new File");
} else {
log("File existed already. Not re-created.");
}
} catch (IOException e) {
log(e.getMessage());
}
}

/**
 * Renames a file. File will not be renamed if a file
 * of the new name already exists in that directory.
 * @param oldFile File to be renamed.
 * @param newFile New name for the file.
 */
public static void renameFile(String oldFile,
      String newFile){
boolean success = new File(oldFile).renameTo(new
File(newFile));
if (success) {
log("File " + oldFile + " renamed to " + newFile);
} else {
log("File " + oldFile + " could not be renamed.");
}
}

/**
 * Moves a file to a different directory. Note that
 * the default behavior is to fail if the directory
 * to which you are moving the file does not already
 * exist. Remember
 * that creating a new File <i>object</i> does not
 * create a new file <i>on disk</i>.
 * <p>
 * So this creates the directory if it doesn't exist.
 * @param fileName Name of the file to be moved.
 * @param destinationDir New location and name
 * for the file.
 */
public static void moveFile(String fileName, String
destinationDir){

File f = new File(fileName);
File dir = new File(destinationDir);
if (! dir.exists()){
//creates any needed directories that don't exist
dir.mkdirs();
}

boolean success = f.renameTo(new File(
new File(destinationDir), f.getName()));
   if (success) {
   log("File " + fileName + " moved to " +
           destinationDir);
   } else {
log("File " + fileName + " could not be moved.");
   }
}

/**
 * Deletes a local file.
 * @param fileName File to delete
 */
public static void deleteFile(String fileName){
boolean success = (new File(fileName)).delete();
if (success) {
log(fileName + " deleted.");
} else{
log(fileName + " could not be deleted.");
}

}

/**
 * Tests whether the file or directory exists already.
 * @param fileName The file to check.
 */
public static void fileExists(String fileName){
boolean exists = (new File(fileName)).exists();
if (exists) {
log("File " + fileName + " exists.");
} else {
log("File " + fileName + " does not exist.");
}
}

/**
 * file.length() returns a long representing the
 * file size in bytes. This makes it a little more
 * convenient to work with
```

```
    convenient to work with.

 */

public static String getFileSize(String

          fileName, Size s){


File file = new File(fileName);


    switch (s) {

    default:

    case BYTES:

       return file.length() + " bytes";

    case KILOBYTES:

       return file.length() / 1024 + " KB";

case MEGABYTES:

    double size = new Double(file.length() /

                           1024).doubleValue() /1024;

//create a display format for the number

//make it have two decimal places

NumberFormat formatter = new DecimalFormat("#.##");

return formatter.format(size) + "MB";

}

    }


private static void log(String msg){

System.out.println("—>" + msg);

}


}
```

The output is like this:

```
—>2248704 bytes

—>2196 KB

—>2.14MB

—>Created new File

—>File C:\test.txt exists.

—>File C:\test.txt renamed to C:\test2.txt

—>C:\test.txt could not be deleted.

—>File C:\test2.txt could not be moved.
```

Useful Methods

The java.io.File class includes a number of methods that make it easy to work with files. For the examples that follow, assume I've got a file on my hard drive at C:\\backups\\test2.txt that we'll use to compare the results of the following methods.

Here they are:

- boolean delete() Deletes a file
- boolean exists() Returns whether the file denoted by this pathname exists
- File getAbsoluteFile() Gets the file name in absolute form: C:\backups\test2.txt
- File getCanonicalFile() Gets the file name in canonical form: C:\backups\test2.txt
- String getParent() Gets the pathname of the parent directory of this file: C:\backups
- boolean isDirectory() Whether the current file is a directory
- boolean isFile() Whether the current file is a file
- boolean isHidden() Whether the current file is marked hidden
- boolean renameTo(File newName) Renames the file on which the method is called to the newName
- boolean setLastModified(long time) Sets the date and time of when this file was last modified
- URI toURI() Makes a URI object of this file: file:/C:/backups/test2.txt
- URL toURL() Makes a URL object of this file: file:/C:/backups/test2.txt

Note that use of some of the preceding methods (in particular toURL() and getCanonical-File() require you to handle the checked exception IOException).

Traversing Directories and Files

This is a class you can easily transfer directly into your projects. It features a method for recursively visiting a directory tree and all of the files in it. There is a method called execute that simply writes out directory names in all caps and indents file names under the directory in which they live. You can easily modify that method to do whatever kind of processing you want (for instance, make it all read only, set the last modified date, whatever). Here it is.

TraverseDirs.java

```
/*
This class lists all of the directories and files
under a specified start directory.
The method "execute" here simply prints out info
about the dir. You could change the execute method
to do something more exciting.
*/
package net.javagarage.demo.io.dir;

import java.io.File;

public class TraverseDirs {

private static final String startDir =
"C:\\eclipse21\\workspace\\dev\\WEBINF\\
src\\net\\javagarage\\demo\\io";

public static void main(String [] arg){
//specify the dir to start in
//and list everything under it
listEachDirAndFile(new File(startDir));
}

 //iterate all files and directories under dir
   public static void listEachDirAndFile(File dir) {
     execute(dir);

     if (dir.isDirectory()) {
       String[] sub = dir.list();
       for (int i=0; i< sub.length; i++) {
         listEachDirAndFile(new File(dir, sub[i]));
       }
   }
 }
```

```
  //iterate only directories under dir (recursive call)
  public static void listDirs(File dir) {
    if (dir.isDirectory()) {
      execute(dir);

      String[] subDirs = dir.list();
      for (int i=0; i < subDirs.length; i++) {
        listDirs(new File(dir, subDirs[i]));
      }
    }
  }

  //iterate files in each dir (recursive call)
  public static void listFiles(File dir) {
    if (dir.isDirectory()) {
      String[] sub = dir.list();
      for (int i=0; i < sub.length; i++) {
        listFiles(new File(dir, sub[i]));
      }
    } else {
      execute(dir);
    }

  }

//this is just where you do whatever work
//you want to do with each file or dir
public static void execute(File f){
if (f.isFile()){
System.out.println("\t" + f.getName());
} else {
System.out.println("Dir: " + f.getName().toUpperCase());
}
}
}
```

For me, the output is like this:

```
Dir: IO

Dir: DIR

TraverseDirs.class

TraverseDirs.java

Dir: FILES

CommonDirectoryTasks.java

CommonFileTasks$Size.class

CommonFileTasks.class

CommonFileTasks.java

WriteBytes.java
```

Notice that the CommonFileTasks$Size.class represents the enum Size, and the IO directory is empty.

File IO: Reading and Writing Stream Data

File IO in Java can be very confusing at first. The reason is because there are a number of abstract classes, and a large number of IO classes that implement specific aspects of IO behavior or requirements, which results in the programmer wrapping two or often three different File class constructors in order to get the desired implementation.

The classes we refer to here are in the java.io package.

Input and Output Streams

Java's implementation of input and output (I/O) services is based on streams. Streams are used to read and write byte data. The data may come from a file, a network socket, a remote object, a serialized object, input at the command line, or somewhere else. Streams can be filtered, allowing limited encryption, serialization of object data, compression using the zip or other algorithm, translation, and so on.

InputStream and OutputStream are the two abstract parent classes that other, more specialized stream types subclass to do the actual work of reading and writing byte data.

InputStream

Depending on what the source of your bytes is, you use a different subclass. Dig: FileInputStream gets bytes from a file. Meant for reading raw bytes. For reading character data, try FileReader.

ByteArrayInputStream

Contains an internal buffer that keeps track of the next byte to be supplied.

PipedInputStream

Gets bytes from an object of type PipedOutputStream. This is intended for use in multithreaded programming, where data is read from a PipedInputStream object by one thread, and the data is written to the PipedOutputStream on another thread.

OutputStream

Depending on where you want to write your bytes, you use one of the following implementing subclasses:

FileOutputStream

Writes the bytes directly to a File object or a FileDescriptor. Use this one if you have byte data like images to write. It's not the best choice for character data—for that consider FileWriter. Note that some platforms may allow a file to be written to by only one stream at a time.

ByteArrayOutputStream

Implements an output stream that writes data to an array of bytes, allowing the buffer to automatically grow. Get the data written to this object using toString() or toByteArray().

PipedOutputStream

This is the sender of data in a piped relationship.

Characters and PrintStream

There are two reasons to refrain from using byte input and output streams for reading and writing character data.

The first is that Java uses Unicode to represent every character with 2 bytes. The purpose of doing so was to ease and encourage internationalization and localization with languages whose characters (such as Korean and Chinese) require 2 bytes. Many plain text files from which you'll read data in, however, use only one byte (8 bits) to represent each character in ASCII text.

The second is that when you work with files containing text, you need to be able to read and write line by line; basic stream I/O doesn't have the concept.

However, you can work with character data in streams by using the PrintStream class. PrintStream provides a number of print and println methods that mimic system calls that are familiar to programmers of C and C++. A chief benefit is that it automatically converts characters, which allows you to pass Strings, chars, and other primitives as arguments. Although it may sound foreign at this point, PrintStream is actually the IO implementation that you're most familiar with; System.out is a print stream, wired for standard output. System.err is a PrintStream as well.

An advantage to working with a PrintStream is that it can be set to automatically flush the buffer after a byte array is written, one of the print methods is called, or if a newline character ('\n') is written to the stream. This accounts for the behavior of System.in, which is implemented as an InputStream.

The unusual thing about working with PrintStream is that it does not throw IOException. Instead, a flag is set internally that can be checked using the checkError() method.

Note that when you use a PrintStream, it converts all characters to bytes, using the current platform's underlying character encoding. If you need to write characters instead of bytes, use a PrintWriter.

Readers and Writers

Java provides a separate set of parent classes for doing character I/O, as opposed to byte IO that is implemented with streams.

Reader is an abstract class, very similar to InputStream. You don't use a reader directly, but rather a different subclass, depending on the source of the characters you need to read.

Writer is also an abstract class, very similar to OutputStream, and its implementing subclasses are used for writing out character data.

Character data comes encoded as a certain character set representation. So let's take a quick detour here before getting into Readers and Writers.

Character Encoding

FileReader and FileWriter read and write double-byte characters. But many native file systems are based on single-byte characters. These streams will utilize the default character encoding scheme of the file system.

To get the default character encoding, check the following property:

```
System.getProperty("file.encoding")
```

You can specify an encoding other than the default. To do this, create an InputStreamReader on a FileInputStream (to read) or OutputStreamWriter on a FileOutputStream (to write) and specify the encoding you want to use. Let's do an example of how to read ISO-Latin-1 or UTF-8 Encoded Data.

```
//String enc = "8859_1"; //ISO-Latin-1

String enc = "UTF8"; //UTF 8


try {

BufferedReader in = new BufferedReader(

          new InputStreamReader(new

FileInputStream("myFile"), enc));

      String line = in.readLine();

   } catch (UnsupportedEncodingException e) {

//handle

   } catch (IOException e) {

//handle

   }
```

Note that the constructor for FileInputStream is overloaded to accept a String containing the charset name, or a charset object, or a CharsetDecoder object. After we construct the FileInputStream object, we pass it as an argument to the InputStreamReader, and then pass that to the BufferedReader constructor, and work with the buffered reader.

You can also call the getEncoding() method to determine what charset is being used by this stream.

Charset

A charset is a combination of a defined set of character codes, and a scheme for encoding the characters into the codes. As we have seen, you can use various charsets when working with file input/output.

Every JVM implementation must support the following charsets:

US-ASCII Seven-bit ASCII, the Basic Latin block of the Unicode character set

ISO-8859-1, ISO-LATIN-1

UTF-8 Eight-bit UCS Transformation Format

UTF-16BE Sixteen-bit UCS Transformation Format, big-endian byte order

UTF-16LE Sixteen-bit UCS Transformation Format, little-endian byte order

UTF-16 Sixteen-bit UCS Transformation Format, byte order identified by an optional byte-order mark

Reader

The implementing subclasses of reader that you might use are illuminated in the following sections.

BufferedReader

This guy is perhaps the most popular reader, and for good reason. He's fiendishly handsome, plays a surprisingly smoking clarinet, and isn't afraid to kegstand character data when pressed to. There are other reasons too. To begin with, BufferedReader provides access to each individual line in a file. The other readers don't. So if you're reading data from a log file, a csv file, or other such source, this is probably your best choice.

Also impressive is that the BufferedReader is buffered. That means that it provides very efficient reading of characters, arrays, and lines. You can specify the buffer size you prefer if you don't like the default, which is usually large enough.

Here is a typical use of BufferedReader:

```
BufferedReader input

  = new BufferedReader(new FileReader("myInputFile.txt "));
```

Here's what's happening. You construct a FileReader object, passing it the String name of the file you want to read. You pass this newly minted FileReader object to the BufferedReader constructor, and call it "input." You can then read the data in one of three ways.

Call input.readLine() to access a single line of character data. A line is considered to be terminated by either a line feed (\n), a carriage return (\r), or a carriage return followed immediately by a line feed. This method will return null after it reaches the end of the stream. So you can read every line of a file like this:

```
String line = "";

while ((line = in.readLine()) != null)

    //do something with line

}

//eof
```

The loop looks a little busy because we're doing two things: checking if we reached the end of the file, and then assigning the current line of character data to the "line" string variable so that we can do something with it inside the loop. That line will be set to null when the end of the file is reached.

Call input.read() to read a single character.

Call input.read(char[] cbuf, int off, int len) to read characters into a portion of an array. This method repeatedly invokes the read() method of the underlying stream until one of the following happens:

1. The number of characters specified in the len parameter have been read.
2. The underlying read() method returns –1, which indicates the end of file has been reached.
3. The ready() method returns false, indicating any further calls to read() would block.

CharArrayReader and StringReader

These classes are useful when the character data source is an array or String.

FileReader

Useful when the character data source is a File. Wrap it in a BufferedReader for freshest taste.

InputStreamReader

Use this class when the character source is an InputStream. It allows any input stream type to be used with characters instead of bytes. Here's the basic recipe for reading lines of characters from an input stream:

1. Create your input stream type based on the byte source.
2. Create InputStreamReader object using that input source.
3. Create BufferedReader object from that input stream reader.
4. Call readLine() method. Returns a non-terminated String.
5. Process yer data.
6. Repeat until readLine() returns null.
7. Take the day off and go to a yachting festival.

Writer

Writer subclasses work the same way as reader subclasses do.

BufferedWriter

Efficiently writes character text to an output stream using a buffer. The BufferedWriter provides a newLine() method, which uses the system-dependent character for a new line as defined by the system property line separator.

Because many Writers send their data directly to the underlying stream, it is a good idea to wrap a BufferedWriter class around a FileWriter or OutputStreamWriter, much as we did with BufferedReader.

```
try {
      BufferedWriter out = new BufferedWriter(new

FileWriter("outFile.txt"));
      out.write("some data");
      out.close();
   } catch (IOException e) {
     //handle
   }
```

In the preceding example, the file outFile.txt will be created if it does not already exist.

If you want to append character data to the end of a file that already exists, pass a Boolean true as a second argument to the FileWriter constructor, as in the following example:

```
BufferedWriter out = new BufferedWriter(new

FileWriter("outFile.txt", true));
```

Note that you can call the flush() method to flush the buffer should it be necessary to manually do so.

OutputStreamWriter

This class serves as a bridge between character and byte stream classes. Its purpose is to encode the characters written to this stream as bytes of the specified charset.

Each time you call the write() method, the charset encoder is invoked on the given characters, and are sent to a buffer. The data is then written to the underlying output stream.

PipedWriter

This class is the companion to PipedReader, mentioned previously. A typical constructor specifies the name of the PipedReader to synch to.

Note that while there is a no-arg constructor for this class, it must be connected to a PipedReader before it can be used. Do so using the connect(PipedReader pr) method. If the PipedWriter is already connected to a PipedReader, an IOException is thrown.

Let's look at some code that puts this to work.

Reading Text from a File

Here is a class that you can likely use with some frequency. It reads in a text file line by line and calls a processing method as it does so. For this example, the processing method simply sends all of the characters to uppercase and prints it to the console.

ReadingTextFromFile.java

```
package net.javagarage.demo.io.files;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

/**<p>
 * This code reads in a file line by line
 * with BufferedReader and processes each
 * line.
 * @author Eben Hewitt
 **/
public class ReadingTextFromFile {

public static void main(String[] args) {
read("C:\\readtest.txt");
}

//convenience method to encapsulate
//the work of reading data.
public static void read(String fileName) {
    try {
        BufferedReader in = new BufferedReader(
        new FileReader(fileName));
        String line;
        int count = 0;
        while ((line = in.readLine()) != null) {
          //our own method to do something
          handle(line);
          //count each line
          count++;
        }
        in.close();

        //show total at the end of file
        log("Lines read: " + count);

    } catch (IOException e) {
        log(e.getMessage());
    }
}

//does the work on every line as it is
//read in by our read method
private static void handle(String line){
    //just send characters to upper case
    //and print to System.out
    log(line.toUpperCase());
}

//convenience to save typing, keep focus
private static void log(String msg){
    System.out.println("—>" + msg);
}
}
```

The output from the file is a result that has all characters in uppercase text, a ? string pointing to every line—whether is has data on it or not—and then the total number of lines read in, as in the following:

```
//blah blah blah
—>EXECUTING ACTIONS
—>
—>EXECUTING ACTION: INSTALLDRVRMGR
—> ARG1: '(NULL)'
—> ARG2: '(NULL)'
—> ARGS: ''
—>INSTALL DRIVER MANAGER SUCCEEDED
```

```
—> RETURN HR: 0X0

—>

—>Lines read: 31
```

Writing Text to a File

This class writes character data to a file. If the file does not exist, it is created. If it does exist and has data in it, any new data is appended to the end of the file.

WritingTextToFile.java

```
package net.javagarage.demo.io.files;


import java.io.BufferedWriter;

import java.io.FileWriter;

import java.io.IOException;

import java.io.UnsupportedEncodingException;

import java.util.Date;


/**<p>

 * This code reads in a file line by line

 * with BufferedReader and processes each

 * line.

 * @author Eben Hewitt

 **/

public class WritingTextToFile {


public static void main(String[] args) {

write("C:\\writetest.log");

log("All done.");

}


//convenience method to encapsulate

//the work of reading data.

public static void write(String fileName) {

try {

BufferedWriter out = new BufferedWriter(
new FileWriter(fileName, true));

out.write("GMT Date:");

out.newLine();

out.write(new Date().toGMTString());


out.close();


} catch (UnsupportedEncodingException uee) {

log(uee.getMessage());

} catch (IOException ioe) {

log(ioe.getMessage());

}

}


//convenience to save typing, keep focus

private static void log(String msg){

System.out.println("—>" + msg);

}

}
```

The result simply prints

```
—> All done.
```

And when you open the written file, it contains data similar to the following:

```
GMT Date:

27 Mar 2004 23:27:16 GMT
```

Reading and Writing Image Data

To find out how to read and write Image data, you can do two things. Check out the following classes in the 1.5.0 API.

The second thing is to look in the example code I've written in the toolbox section of the garage. The GaragePad app lets you draw freehand onto the canvas and then save the pixel data into an image.

Okay. That sounds kind of lame. So here is a sneak preview:

```
Image image = canvas.createImage(rect.width, rect.height);

Graphics g = image.getGraphics();

canvas.paint(g);

ImageIO.write((RenderedImage)image, "jpg", outFile);
```

This code uses the write method of the javax.imageio.ImageIO class to write out a new file with JPEG encoding, and it will be called whatever the value of outFile is. To do so, it creates a java.awt.Image object from a java.awt.Canvas object, which captures the drawn pixels.

Chapter 23. FRIDGE: GUACAMOLE

I think it was pretty hard, doing all of that work in the last topic. Where I come from, programmers can get hungry just like anybody else. And when that happens, I think there's nothing better than a relaxing break, with a cool side of fresh guacamole dip. Don't you agree?

Now. I'm going to divulge here my very own recipe for making guacamole. In my view, this one page right here is worth the entire price of the book. That's really what I think. I think you'll like it that much. Please try it out and tell me what you think. This recipe serves four people. Two, if you live in my family.

Most importantly, you have to choose good avocados. This can be fun. First, you don't want the green ones; those are not ripe enough. You also don't want the totally black ones that mush a little when you pick them up, because those are overripe and will have undesirable black dots inside. Choose avocados that are a mix of green and black on the outside, and that feel firm but also give a little when you squeeze them.

Now here we go:

Get a ceramic mixing bowl and empty into it the fruit of four avocados.

Mash the avocado with a big masher.

Cut a lime in quarters. Squeeze the juice of the lime wedges onto the avocado.

Take two tomatoes, still on the vine, and dice them. Make the sections large, though, like the size of a dime. Drop those into the bowl.

Get a big white onion and cut it in half. Save one half for something else later; we don't need it. Cut the other half into dime-sized pieces just like the tomato. Not lame little tiny pieces. Good sized pieces. I might call them chunks.

Generously add fresh ground salt and pepper. Stir all of this together.

Here is the killer. Dice a chile pepper. One of those long, thin dark green ones. They're called serranos. If you can't find that kind in your store, you might want to consider moving to the desert like me. We have both Java and serranos here.

Stir all this up together and serve with tortilla chips. It should be fluffy, fresh, cool, and perfectly delicious.

It's a meal in itself. It takes about 35 minutes to make. This is one of my favorite things in the world, probably.

Chapter 24. USING REGULAR EXPRESSIONS

DO OR DIE:

- Straight Cs
- Blow this taco stand the minute I graduate
- Stay sweet

Regular expressions are fiendish things. However, sometimes you need 'em. Luckily, the Java API introduced in 1.4 made working with them a good deal easier. In this topic, we will do a brief overview of the regex for the uninitiated (thank you sir; may I have another). Next, we'll see the facility that Java makes available for using them. Then we'll use them. Then we'll be done so we'll stop.

Purpose of Regex

If you're already Mr. Big Deal Regex Guy[1] and just want to know how Java does it, skip this section. Else, continue.

[1] I would like to take this opportunity to acknowledge our female readers within the programming community. Please note that throughout this book, for each reference to "x guy", where x refers to some technology, let "guy" refer to a member of either gender. I hope that this is accepted usage here, as such use colloquially seems generally accepted. For example, "Hey, are you guys going to lunch with us?" spoken cheerfully to one's male and female coworkers, or, "Uh, when did you guys get back from Reno—I thought you weren't gunna be home til Sunday," when referring directly to one's own parents.

A regular expression is a description of a textual pattern that enables string matching. You create a string of characters with special meaning and compile them into a regular expression pattern, and then use that pattern to find strings that match it.

The most common example of a regex type pattern is *, where the asterisk matches everything. In SQL, you might type `%ant` and because the `%` in SQL matches zero or more of any character, your expression will match "Fancy pants", "supplant", and so forth.

See Table 24-1 for a reference of metacharacter shortcuts you can use as you write regular expressions.

Regex Metacharacters

If You Want...	Use
Any digit 0 – 9	\d
Any non-digit	\D
Letters, numbers, and underscores	\w
Any character that isn't a letter, number, or underscore	\W
A whitespace character	\s
Any non-whitespace character	\S
Any one character	-
Allow nothing before this character	^
Allow nothing after this character	$
Zero or one character	?
Zero or more characters	*
One or more characters	+

The metacharacters are those that stand in for character types or are a placeholder for the number of characters they specify. You use metacharacters in combination with regular character sequences. Although that sounds straightforward enough, it can be tricky to use them in meaningful combinations of any complexity with characters.

Maybe you feel comfortable with regex now after that short introduction. That's really about all there is to metacharacters. A moment to learn, way too long to master.

As the great blues artist Willie Dixon sang, "I ain't supestitious, but a black cat crossed my trail...." So just in case you *do* have some reservations, let's take a look at some examples in Table 24-2.

Matching Regular Expressions with Character Sequences

This Regex	Matches These Strings
A*B	B,A,AAB, AAAB, etc.
A+B	AB,AAB,AAAAB, etc.
A?B	B or AB only
[XYZ]C	XC,YC,ZC only
[A-C]B	AB,BB or CB only

[2-4]D	2D,3D,4D only
Deal\s\d	Deal 8, Deal 999, but not Deal3 or DealA
(X|Z)Y	XY or ZY
(cat\s) { 2}	cat cat
(cat\s){ 1-3}	cat, cat cat, or cat cat cat

I hate saying this, but regular expressions are the subject of a number of complete books and it's probably best if I refer you to one of them at this point instead of going further with them outside the scope of their use in Java. Anyway, the preceding tables and the following sample code really should cover most of the situations you will need at first. Before you do anything crazy, check the API documentation for the java.util.Pattern class. It features a huge number of commonly used regular expression patterns, and if you're stuck, that might help you out. Which would be great. I have a really cool car.

Regex in Java

Ah, regex in Java. It's no big thing. Although Java can't do much to make our time-honored regular expressions any less obscure, they can make them easier to work with.

The relevant package here is java.util.regex. It contains two classes: Pattern and Matcher. Here's how it works:

1. Create a String containing your regular expression.
2. Compile that String into an instance of the Pattern class. You do this using the static `Pattern.compile(String regex)` method.
3. Create an instance of the Matcher class to match arbitrary character sequences against your pattern.

Those steps look like the following in code:

```
Pattern pattern = Pattern.compile("hi\s {1-3}");

Matcher matcher = p.matcher("hi hi hi");

boolean isMatch = m.matches(); //true
```

You can also do all of this in one step (very cool).

```
boolean isMatch = Pattern.matches("hi\s {1-3}", "hi hi hi");
```

It is easy to use the metacharacters in your regular expressions to determine if a String matches or the regex not. For example, say you need to do some validation on data entered by users in your application. You need to make sure that a U.S. zip code is entered, which means five digits. You can do it as follows:

```
/**
 * Returns whether or not the passed number is
 * exactly five digits, which the requirements state
 * the customer number is.
 * Matches: 01234 and 85558<br>
 * Does not match: three or 123ert or 999999 or 876
 * @param numToCheck the number you want to check.
 * @return true if the passed number is 5 digits,
 * false otherwise
 */
private boolean validateCustomerNumber(String
        numToCheck){
  return Pattern.matches("^\\d{5}$", numToCheck);
}
```

Pattern and Matcher

As mentioned, there are only two classes in the java.util.regex package.

A Pattern object is a compiled instance of a String representing a regex. The Pattern object can be used to create a Matcher object to match character sequences.

A Matcher object interprets patterns to match them with character sequences. You can create a Matcher directly or through invoking the Pattern class's `matcher()` method. There are three ways to match a pattern using Matcher.

1. The `matches()` method tries to match the complete input sequence against the pattern.
2. The `lookingAt()` methods tries to match the input sequence against the pattern, starting at the beginning.
3. The `find()` method scans the input sequence looking for the next subsequence that matches the pattern.

Regular expressions are commonly used to search through vast reams of digital text. They can also be used to check if a string passed in from a user matches a certain pattern. A good example of this is when you ask users to choose a password, and require that it be six characters and contain at least one number. User validation is another good example—we use it commonly to make sure that an e-mail address entered by a user is well formed.

However, there is another common use for regular expressions, and that is replacing text that matches a certain pattern.

Let's look at a complete example now that demonstrates use of both features: matching strings (and returning true if they match and false if they don't) and replacing character sequences found with different character sequences of our choice.

RegexDemo.java

```
package net.javagarage.demo.regex;

import java.util.regex.*;

/**

 * Demonstrates how to use the Java regular expressions

 * classes. You should be able to use

 * this code right out of the box, as it also serves

 * as a useful library that checks all of these things:

 * <ul>

 * <li>URL

 * <li>email address

 * <li>phone number

 * <li>dates

 * <li>numbers

 * <li>letters

 * <li>tags, such as <body>

 * <li>general characters, such as new lines,

 * whitespace, etc.

 * </ul>

 * <p>

 * All of these regular expressions must be escaped

 * because we are using them in Java. That means that

 * a regex that looks like this: <br>

 * <code>\s+</code>

 * <br>

 * we must write like this:<br>

 * <code>\\s+</code>

 * <br>

 * ...with all of those \ characters escaped.

 * <p>

 * By default, pattern matching is greedy, which

 * means that the matcher returns the

 * longest match possible.

 * <p>

 * These regexes should be just fine for general

 * business app use, but don't put this code in your

 * nuclear reactor or space shuttle (duh).

 *

 * @author eben hewitt
 * @see java.util.regex.Pattern,
java.util.regex.Matcher
 **/
public class RegexDemo {

//Constants for regular expressions for those
actions:

//Constant matches any number 0 through 9.
public static final String NUMERIC_EXP = "[0-9]";

//Constant matches any letter, regardless of case.
public static final String ALL_LETTERS = "[a-zA-Z]";

//POSIX for all letters and numbers. Works only for
        US-ASCII.
public static final String ALPHANUMERIC = "\\p{Alnum}";

//Useful for stripping tags out of HTML
public static final String TAG_EXP = "(<[^>]+>)";

//matches an email address
public static final String EMAIL_ADDRESS_EXP =
"(\\w[-._\\w]*\\w@\\w[-._\\w]*\\w\\.\\w{2,3})";

//matches a URL
public static final String URL_EXP =
"^http\\://[a-zA-Z0-9\\-\\.]+\\.
         [a-zA-Z]{2,3}(/\\S*)?$";

//matches 12/14/2003 and 1/5/03
public static final String DATE_EXP =
"^((0[1-9])|(1[0-2]))\\/([1-9]|(0[1-9])|
         ([1-2][0-9])|3[0-1])\\/[0-9]{1,4}$";

//phone number with area code
public static final String PHONE_EXP =
        "((\\(\\d{3}\\)
?)|(\\d{3}-))?\\d{3}-\\d{4}";

/**

 * Matches a line termination character sequence.

 * In this case, it is a character

 * or character pair from the set:

 * \n, \r,\r\n, \u0085,

 * \u2028, and \u2029.

 */

public static final String LINE_TERMINATION =

"(?m)$^|[\\r\\n]+\\z";


/**

 * Matches more than one whitespace character

 * in a row. For example: "hi there".

 */

public static final String DUPLICATE_WHITESPACE =

        "\\s+";


/**

 * typical characters

 */

public static final String TAB = "[\t]";

public static final String NEW_LINE = "\n";

public static final String CARRIAGE_RETURN = "\r";

public static final String BACKSLASH = "\\";


/**

 * Tells you if a certain String matches a certain

 * regex pattern. Use this method if you want to

 * define your own regex, and not use one provided

 * via the constants.


 * @param s String used as character sequence

 * @param p String regular expression you want to match

 * @return boolean indicates whether the pattern is

 * found in the String.

 * If the pattern is found in the string,

 * returns true.
 */
public static boolean regexMatcher(String input,
        String p) {
CharSequence inputStr = input;
String patternStr = p;
// Compile regular expression
Pattern pattern = Pattern.compile(patternStr);
// Replace all occurrences of pattern in input
Matcher matcher = pattern.matcher(inputStr);
return matcher.find();
}

/**
 * Finds all instances of <code>pattern</code> in
 * <i>string</i>, and replaces each with the
 * <code>replacer</code>. <br>
 * Examples:<ul>
 * <li>Input:regexReplacer("D99D",
 * RegexHelper.ALL_LETTERS, "x")
 * Result: x99x
 * <li>etc
 *
 * @param input String you want to clean up.
 * @param p String defining a regex pattern that you
 * want to match the passed String against.
 * @param replacer String containing the characters
 * you want in the final product
 * in place of the pattern string characters.
 * @return String Containing the same String but with
 * replaced instances.
 */
public static String regexReplacer(String input,
        String p, String replacer) {
    CharSequence inputStr = input;
    String patternStr = p;
    String replacementStr = replacer;
    // Compile regular expression
    Pattern pattern = Pattern.compile(patternStr);
    // Replace all occurrences of pattern in input
    Matcher matcher = pattern.matcher(inputStr);

    return matcher.replaceAll(replacementStr);
}

//shows the results of using the different
//regexes for matching
private static void demoMatchers(){
    print("MATCHING:");

    // This input is not alphanumeric
    print("Is %*& alphanumeric? " +
      regexMatcher("%*&", RegexDemo.ALPHANUMERIC));

    //good email: returns true
    print("Is VALID EMAIL: " +
    regexMatcher("dude@fake.com",
    RegexDemo.EMAIL_ADDRESS_EXP));

    //bad email: returns false
    print("Is VALID EMAIL: " +
    regexMatcher("not @ good",
    RegexDemo.EMAIL_ADDRESS_EXP));

    //good URL: returns true
    print("Is URL: " +
    regexMatcher("http://www.javagarage.net",
    RegexDemo.URL_EXP));

    //bad URL: returns false
    print("Is URL: " +
    regexMatcher("java.com", RegexDemo.URL_EXP));

    //check date 30/12/2003: false
    //no, it isn't localized. you can't have everything
    print("Is VALID DATE: " +
    regexMatcher("30/12/2003", RegexDemo.DATE_EXP));
    //check date
    print("Is VALID DATE: " +
    regexMatcher("5/12/2003", RegexDemo.DATE_EXP));

    //check a phone number
    print("Is VALID PHONE: " +
    regexMatcher("(212)555-1000",
RegexDemo.PHONE_EXP));
}


//shows replacing characters matched
private static void demoReplacers(){
    print("REPLACING:");
    //remove all HTML tags from this code
    //and replace with nothing
    print("Remove Tags: " +
    regexReplacer("<html><body><p>the
             text</p></body></html>",
    RegexDemo.TAG_EXP, ""));

    //remove extra whitespace
    print("Extra whitespace replace: " +
    regexReplacer("far away",
    RegexDemo.DUPLICATE_WHITESPACE, " "));

    //remove and replace new line characters
    print("Remove New Lines: " +
    regexReplacer("1\n2\n3", RegexDemo.NEW_LINE,
    " NEW LINE WAS HERE "));

    //replace tab characters (\t) with pipes (|)
    print("Tab replace: " +
    regexReplacer("Hello\tSweetheart",
               RegexDemo.TAB, "|"));
}

//just to save typing
private static void print(String s){
    System.out.println(s);
}

//run the show
public static void main(String[] a) {

    demoMatchers();
    demoReplacers();
}
}
```

The output of executing RegexDemo.java is

```
MATCHING:
Is %*& alphanumeric? false
Is VALID EMAIL: true
Is VALID EMAIL: false
Is URL: true
Is URL: false
Is VALID DATE: false
Is VALID DATE: false
Is VALID PHONE: true
REPLACING:
Remove Tags: the text
Extra whitespace replace: far away
Remove New Lines: 1 NEW LINE WAS HERE 2 NEW LINE WAS
HERE 3
Tab replace: Hello|Sweetheart
```

As with most of the code in this here *Garage* book, the idea is that the comments are inline, and I want to encourage you to read the code. So I put all the stuff I want to say about RegexDemo.java in the class itself. You can use this code in your environment. It will come in handy maybe.

You can also match and/or replace strings in files (obviously, I think), not just in strings you pass into the static methods of the preceding class. Want to try it? Okay!

Let's say that you've got a file at E:\\old.data, and you want to find all occurrences of the phrase "wood" and replace it with the character sequence "banana". You can do it with the file ReplacePatternFile.

The following is the content of old.data:

```
How much wood could a woodchuck chuck

if a woodchuck could chuck wood.
```

ReplacePatternInFile.java

```
package net.javagarage.demo.regex;


import java.io.*;

import java.util.regex.Matcher;

import java.util.regex.Pattern;


import javax.swing.JOptionPane;


/**

 * <p>

 * Demos how to open an arbitrary text file and

 * replace all occurrences of the given regex pattern

 * with some other character sequence. Neat!

 * <p>

 * And Dude said let John Lee Hooker accompany the

 * writing of this class. And thus it was done.

 * And Dude saw that it was good.

 *

 * @author eben hewitt

 */

public class ReplacePatternInFile {

    BufferedWriter writer;

    BufferedReader reader;


public static void main(String[] args) {

    ReplacePatternInFile work = new ReplacePatternInFile();
    work.replaceFile("E:\\old.data", "E:\\new.data");
    System.out.println("Replaced matches");
}

//this method reads in the file specified by the
first
//param, and writes it out to the file name specified
//by the second param
private void replaceFile(String fileIn,
                String fileOut) {
    File inFile = new File(fileIn);
    File outFile = new File(fileOut);

    try {
    //get an inputstream to read in the desired file
    FileInputStream inStream = new
FileInputStream(inFile);

    //get an output stream so we can write the new file
    FileOutputStream outStream = new
FileOutputStream(outFile);
    //the buffered reader performs efficient reading-in
    //of text from a source containing characters
    reader = new BufferedReader(
    new InputStreamReader(inStream));

    //this will write the new data to our second file
    writer = new BufferedWriter(
    new OutputStreamWriter(outStream));

    //this is the string we want to match for replacing
    Pattern p = Pattern.compile("wood");
    Matcher m = p.matcher("");

    String s = null;
    String result;

    //loop over each line of the original file,
    //and each time we meet an occurrence of p
    //the readLine method
    while ( (s = reader.readLine()) != null ) {
    //reset with a new input sequence
    m.reset(s);

    //replace the matches with this string
    result = m.replaceAll("banana");
    //write the data to the file
    writer.write(result);
    //put a new line character
    writer.newLine();
    }

    //clean up after yourself
    reader.close();
    writer.close();

    } catch (IOException ioe){
          JOptionPane.showMessageDialog(null,
ioe.getMessage());
          System.exit(1);
    }
    }
}
```

After executing the code, we have a new file called new.data. The contents of that file should look like this:

```
How much banana could a bananachuck chuck
if a bananachuck could chuck banana.
```

Well, that's it for this topic of regular expressions. That's enough about it for all I care. I just hope that it's all you care for too. At least for right now. Now I feel uncomfortable. Why since I was in sixth grade I don't know what to do with my hands. Maybe I should take up smoking. Hm. It is possible, after all.

And on that note, let's split.

Chapter 25. CREATING GUIS WITH SWING

DO OR DIE:

- Shake it.
- Please, don't break it.

Let GUI = Graphical User Interface;

Let Swing = JavaProgrammingLanguage.getProgrammingElement(GUI);

Swing.listImportantElements() {

//returns: TopLevelContainer (Frame)

//Intermediate Container (ContentPanel)

//LayoutManager(s)

//Atomic Elements (Components)

Frame.getDefinition() Main window that holds every other component of a Swing application. The frame, which is created as an object of type JFrame, contains controls for iconifying (minimizing) and maximizing the frame, and a little X for closing it.

Frame.getNote() Note that the JFrame is not the only kind of top-level container for Swing apps. JDialog is too, and so is JApplet (which is discussed in its own topic on Applets).

FRIDGE

The back of this book, Chapter 35, features a "Toolkit," which is a load of applications that are pretty good quality that feature popular kinds of functionality, such as displaying scrollable areas, transforming XML and displaying it in a user interface, and connecting to a socket at the click of a menu button. Making GUIs with Java using AWT and Swing is a gigantic, really gigantic topic. The books that cover Swing alone are around 1,400 pages long. This topic tries to show you around the landscape a little, and then points you to the other places in the book where you can see this stuff implemented live. There is simply not room to go into Swing in depth here. But as we look at the landscape, you can get a feel for how things work, and it should be enough to get you started.

Anatomy of a Swing App: Stuff You Typically Need to Do in Swing

Starting the App: InvokeLater()

First, the main method calls a static method from the javax.swing.SwingUtilities class called invokeLater(). This method is suited for starting the application, as its purpose is to update the GUI asynchronously on the event dispatching thread.

You can call this method from any thread if you want the event dispatching thread to run some code. If used in this way, it would be invoked like so:

```
Runnable updateGUI = new Runnable() {

   public void run() {

      new MyGUI();

   }

};

SwingUtilities.invokeLater(updateGUI);
```

Here, we just create a new object of a type called MyGUI, which probably extends JFrame or holds a JFrame member variable, sets up the properties of the frame such as how big it should be and what kinds of components should be in it, and then shows itself.

The invokeLater method causes the Runnable to have its run method called in the dispatch thread of the AWT Event Queue. It won't happen until after all pending events are processed. That thread, the AWT Event Queue, stores events in a queue, and executes them asynchronously and sequentially (in the order stored).

Creating the Frame

Now let's look at how to make a main application window for storing buttons and text fields and other user controls. You do this using a JFrame, which you can give a title, as in the following example:

```
JFrame myFrame = new JFrame("My Application");

//now call methods on myFrame to add controls, set

//the layout, make it visible, and so forth
```

Note that if you call the static method JFrame.setDefaultLookAnd FeelDecorated(true), this must be invoked before you call the JFrame's constructor, or it will have no effect. After you have a frame, there are a few maintenance things you need to set up. First, call frame.setDefaultClose Operation(JFrame.EXIT_ON_CLOSE) to ensure that when you click the X to close the window it will actually close.

Next, you need to call the pack() method, which fits all of the components into the available space according to their preferred size.

Finally, you call setVisible(true) in order to display the frame and its components.

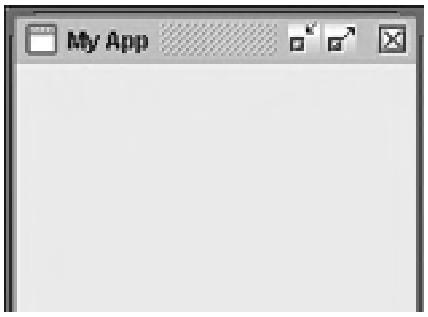
Frame.runExample():

This code demonstrates creating, sizing, and showing an empty frame, similar to the Windows Form you get when you start Visual Studio.

FrameDemo.java

```
package net.javagarage.demo.ui;

/**<p>

 * Very simple frame that does nothing.

 * The purpose is to give you something to

 * start with each time you make a new Swing app.

 * </p>

 * @author Eben Hewitt

 **/

import java.awt.*;

import javax.swing.*;


public class FrameDemo {


      JFrame frame;

    //simply creates the frame, adds an empty

    //label to it so that it has some size,

    //and shows the frame

    private static void launch() {


    /* Change the way the frame looks and feels.

     * decorate it with this command, instead of

     * having the system do it. you must call this

     * method BEFORE creating the JFrame.

     */

    JFrame.setDefaultLookAndFeelDecorated(true);


   //create window, and put a title

   //in top-left corner

   frame = new JFrame("My App");
   //make the app stop when window is closed

frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

   JLabel emptyLabel = new JLabel("");
   emptyLabel.setPreferredSize(new Dimension(200, 200));
   frame.getContentPane().add(emptyLabel,
                              BorderLayout.CENTER);

   //cause the window to be sized in such a way that
   //it snugly fits the preferred size and layouts of
   //the components in it. this is a method of
   //the java.awt.Window class
   frame.pack();
   //show it
   frame.setVisible(true);
  }

public static void main(String...args) {

SwingUtilities.invokeLater(new Runnable() {
   public void run() {
       launch();
   }
   });
}
}
//end FrameDemo.java
```

Figure 25-1. The basic frame will shut down the JVM when closed, and can be resized.



ContentPanel.getDefinition() The frame contains a content panel. The main content pane, which you get a reference to via frame.getRootContent Pane() is the workspace onto which you place buttons, text fields, and other elements with which the user directly interacts.

ContentPanel.getNote() The content panel is also called a pane, like a window pane. In the main, a pane can be plain. It is simply a plane.

There are different kinds of panes, however, including the JScrollPane, which automatically adds scrollbars, and the JTabbedPane. Both of these we make good use of in examples here.

Creating Menus

It is easy in Swing to create menus. First, you create a menu bar to hold each menu. A menu is the little area that drops down out of the menu bar.

```
//create the menu bar
JMenuBar menuBar = new JMenuBar();

// Create a menu
JMenu menu = new JMenu("Tools");
menuBar.add(menu);

// Create a menu item
JMenuItem hacksaw = new JMenuItem("Hacksaw");
hacksaw.addActionListener(someActionListener);
menu.add(hacksaw);

// Install the menu bar in the frame
frame.setJMenuBar(menuBar);
```

The main thing to note here is that clicking menu items creates an action event that you need to handle with an ActionListener implementation. There is a good example of using menus in the Garage Text Editor Toolkit example.

A Note About Mixing Swing and AWT Components

Swing components are lightweight. AWT components, now out of fashion, are heavyweight. That means that it is a bad idea to mix the older AWT components with Swing components. The reason is that you will get unpredictable behavior. More specifically, you will find that the AWT components will always force themselves in front of the sweet little lightweight Swing components. You could have serious trouble displaying a Canvas, for example, in tandem with a JPanel. Just something to beware of.

The designers of the Swing libraries learned a lot from working with AWT, and you should be able to find the components you need in the newer libraries without having to mix.

Disclaimer: The most popular book on Swing is 2 volumes and nearly 3,000 pages. We don't have room for much more. We are really touching the surface here, but this is enough to get started. As this book shows, you can make a number of interactive and GUI programs without going too far into layout managers, and after you have handled one action, you've handled them all.

Adding User Controls

Components.listImportantElements() {

//returns:

- JButton A button that shows text and can do something when you click it.
- JLabel An element used generally for holding textual messages.
- JTextField Like an HTML <input>, this component allows the user to enter text.
- JEditorPane This lets the user enter text or display text-based files.
- JScrollPane A container that scrolls. Use it by creating content in a different panel and then adding that panel to the scroll pane.
- JTabbedPane A container that holds tabbed panes.

/*

Let's talk about a few different, important Swing components in turn. They're just objects, so after you are familiar with their general behavior, you can create them, call setX() methods on them to specify their particulars, add an event handler if necessary, and add them to the content pane.

*/

}

User Controls.stepInto()

The components we discuss in this section are commonly used controls that users need in applications. This is by no means an exhaustive look at the controls Swing makes available. It's just what you need to get started. Anyway, after you are used to working with a few of the controls, it is an easy step to read the API for what other kinds of components you get for free.

JLabel

This is an area. Just a plain area, corresponding to a label in Visual Basic. You can put text on it, which is its usual purpose. Or an image, which is probably a good idea to put there. Labels are just regular folk. Create them like this:

```
JLabel label = new JLabel("You must conform.");
```

The text passed to the constructor will show up on the label, and will be left-justified and centered vertically.

To create a label containing an image and text, you can follow this example:

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Container;
import java.awt.Dimension;
import java.awt.Font;

import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

public class LabelTest {
    public static void main(String[] args) {
        new LabelTest();
    }

    public LabelTest(){
        JFrame frame = new JFrame("Label Test");
```

```
     Container mainPanel = frame.getContentPane();

     mainPanel.add(getTitlePanel("My Label test
          text.", "/images/some.gif"));

     frame.pack();
     frame.setVisible(true);
  }

  private JPanel getTitlePanel(String title,
          String imagePath){
     JPanel titlePanel = new JPanel(new
BorderLayout());

        //use the Dimension class to make a 2-D
        //size of width, height
     titlePanel.setPreferredSize(
          new Dimension(300, 100));

        //make it white instead of default gray
     titlePanel.setBackground(Color.WHITE);

        //the image
     JLabel lblImage = new JLabel();
        //get an image from the path specified
     ImageIcon ico = new ImageIcon(imagePath);
        //add the image to the label
     lblImage.setIcon(ico);

        //the text
     JLabel lblTitle = new JLabel();
        //set the font for the text
     lblTitle.setFont(new Font("SansSerif",
          Font.BOLD, 18));
        //add the text to the label
     lblTitle.setText(title);

     //add the image to the panel
     titlePanel.add(lblImage, BorderLayout.WEST);
```

```
        //add the text to the panel

        titlePanel.add(lblTitle, BorderLayout.CENTER);


        return titlePanel;

    }

}
```

We will cover layouts, which are used above, later in this chapter. Otherwise, the code should be self explanatory. If the image is not found, then it simply won't appear.

JTextField

This control corresponds to an input control in HTML. Users enter text here, or you can cause it to have a default value when it is displayed by calling its setText(String s) method. Then, you can call the getText() method to retrieve its String value.

JButton

The JButton represents a clickable button that can have a text label, an image label, or both. The constructors follow:

- JButton() Creates a button with no set text or icon.
- JButton(String text) Creates a button with text label
- JButton(Icon icon) Creates a button with an icon label.
- JButton(String text, Icon icon) Creates a button with both text and icon.
- JButton(Action a) Creates a button where properties are taken from the Action supplied.

Check out the following ButtonDemo class to see a JButton with a text label that has several properties set, and which performs an action when you click it.

ButtonDemo.java

```
package net.javagarage.demo.ui;

/**<p>

 * Demonstrates how to use a text field, a button,

 * and a label. When you enter text and click the

 * button, the label's text changes to your value.

 *

 * This example uses the default layout, which

 * is FlowLayout. We don't have to do anything

 * to make that work, and components will just
 * stack up after each other.

 * @author Eben Hewitt

 **/

import javax.swing.AbstractButton;

import javax.swing.JButton;

import javax.swing.JLabel;

import javax.swing.JPanel;

import javax.swing.JFrame;

import javax.swing.JTextField;


import java.awt.Color;


import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import java.awt.event.KeyEvent;


public class ButtonDemo extends JPanel

implements ActionListener {


protected JButton button;

protected JTextField txtField;

protected JLabel label;


public ButtonDemo() {

//creates the field to enter text

txtField = new JTextField();

//make the background yellow

txtField.setBackground(Color.YELLOW);

//will disappear if we don't do this

txtField.setColumns(25);

//creates the button object

button = new JButton("Change text");


//places text in the button
```

```
button.setVerticalTextPosition(AbstractButton.CENTER);

//this is left for locales that read left to right
button.setHorizontalTextPosition(AbstractButton.LEADING);

//change some frivolous things just to show
button.setBackground(new Color(80,80,80));
button.setForeground(Color.WHITE);
/*
 * it is sometimes desirable to allow the user to
 * type with the keyboard to fire an event, as
 * opposed to only allowing clicking. to do this, set
 * the mnemonic using the setMnemonic method.
 *
 * The arg to this method is a char, indicating the
 * character that will fire the event, often in
 * combination with the ALT key.
 *
 * it should be one of the characters in the label,
 * in which case it will appear underlined.
 *
 * See the KeyEvent API for pre-defined keys.
 * IE, we could use VirtualKey C,
 * which means ALT + C character.
 */

button.setMnemonic(KeyEvent.VK_C);
//inherited from AbstractButton
//this is what the event listener will pick up
button.setActionCommand("changeText");

//this is what makes the button do something
//when you click it. without registering
//and action listener, nothing will happen.
button.addActionListener(this);

//create the label whose text we'll change
label = new JLabel();
```

```
label.setText("This is the default text");



//Add the components to the container in order.
//since we're using FlowLayout, we don't need
//to do anything else
add(txtField);
add(button);
add(label);
}

//this is what happens when the button is
//clicked (or when ALT C is keyed)
public void actionPerformed(ActionEvent e) {
//we set the button's ActionCommand to this
if ("changeText".equals(e.getActionCommand())) {
//make the label value be whatever
//the user typed
label.setText(txtField.getText());
}
}

public static void main(String[] args) {
//the event-dispatching thread will show the app
//This method of running Swing apps is recommended
//by Sun
javax.swing.SwingUtilities.invokeLater(new Runnable() {
public void run() {
launch();
}
});
}

//sets up the look and feel, creates the frame,
//lines up the objects, and shows them.
```

```
private static void launch() {
JFrame.setDefaultLookAndFeelDecorated(true);

//Create and set up the window.
JFrame frame = new JFrame("Button Demo");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

//Create and set up the content pane.
ButtonDemo newContentPane = new ButtonDemo();
newContentPane.setOpaque(true);
        //content panes must be opaque
frame.setContentPane(newContentPane);

//Display the window.
frame.pack();
frame.setVisible(true);
}
}
//end ButtonDemo
```

Spacing and Aligning Components

Spacers

There is a class that inherits from JComponent called Box. This is a lightweight class that uses BoxLayout as its Layout Manager. You can use this class to get at the convenience methods it makes available. These methods allow you to add invisible components to the layout that help you fit, position, and size the regions of your layout. Specifically, they help you do that by creating certain relationships between your visible control components.

We won't go into them in much depth. Suffice to say, here they are, here is what they're for, and go try them out for yourself to get the feel for them. This is the point in Swing programming that it becomes a little like learning to drive: eventually, you just need to feel when to shift gears and how quickly to let up on the clutch.

Glue

Glue represents space of malleable size in a layout. Adding glue between two components tells the runtime to take excess space in the layout and smoosh it between the components.

```
container.add(buttonOne);

container.add(Box.createHorizontalGlue());

container.add(buttonTwo);
```

You can also createVerticalGlue();.

Area

A rigid area is an invisible region that is always of the specified size. You use it to create a certain amount of space between two components that you don't want to change, or perhaps to push text up toward the top of your window by placing a rigid area below it.

An example follows:

```
static Component Box.createRigidArea(Dimension d)
```

Dimension takes two ints specifying the height and width.

```
container.add(buttonOne);

container.add(Box.createRigidArea(new

Dimension(10,0)));

container.add(buttonTwo);
```

Strut

A strut is an invisible component of fixed length. It is used to force space between two components. The strut has no width at all, except there is excess space surrounding it, in which case it will fill up the space in the manner of any component. There are two kinds of struts: vertical and horizontal.

Create a strut using these methods:

```
static Component createVerticalStrut(int height)

static Component createHorizontalStrut(int width)
```

Here is an example of how to use a strut to put space between two buttons:

```
container.add(buttonOne);

container.add(Box.createVeticalStrut(15));

container.add(buttonTwo);
```

You'll notice in this case, the strut acts like the rigid area. However, it is recommended that you use rigid areas instead of struts, because struts can have unlimited length, which could cause undesirable results in nested components.

JEditorPane

We create a JEditorPane in the blogger application in the Toolkit section. I refer you there to see in detail how that is used. But there is one question that is often asked when figuring out how to use JEditorPanes. And that's how to handle hyperlink clicks.

You do need to do something in fact. If you use a JEditorPane to display a simple HTML page, which is one of its main purposes, nothing will happen when you click on the links. Unless you create an object of a class that implements the HyperlinkListener interface, as shown in the following example:

```
try {
    String url = "http://www.javagarage.net";
    JEditorPane editorPane = new JEditorPane(url);
        //don't let user try to edit content!
    editorPane.setEditable(false);
    editorPane.addHyperlinkListener(new LinkListener());
} catch (IOException e) {
      //handle...
}

class LinkListener implements HyperlinkListener {
    public void hyperlinkUpdate(HyperlinkEvent event) {
      if (event.getEventType() ==
              HyperlinkEvent.EventType.ACTIVATED)
{
         JEditorPane pane =
               (JEditorPane)event.getSource();
         try {
            // Show the new page in the editor pane.
            pane.setPage(event.getURL());
         } catch (IOException e) {
            //handle..
         }
      }
    }
}
```

JTabbedPane

This component provides a way to access an arbitrary set of panels. It capitalizes on CardLayout shown earlier, and offers small tabs atop your JPanels to offer the user an easy way to switch around. This is used frequently in office productivity applications to let the user specify preferences and so forth. It is handy and surprisingly easy to use.

TabbedPaneDemo.java

```
package net.javagarage.demo.ui;
/**<p>
 * Shows a JTabbedPane, which allows you to
 * choose between pages with a tab, like
 * paper manila folders.
 *
 * </p>
 * @author Eben Hewitt
 **/

import javax.swing.JTabbedPane;
import javax.swing.ImageIcon;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JFrame;
import javax.swing.JComponent;
import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.GridLayout;
import javax.swing.SwingUtilities;
import java.net.MalformedURLException;
import java.net.URL;

public class TabbedPaneDemo extends JPanel {
static URL imageURL;

public TabbedPaneDemo() {
//remember that the call to super()
```

```
//MUST be first call in constructor

super(new GridLayout(1, 1));

//create the tabbed pane

JTabbedPane tabbedPane = new JTabbedPane();


/*remember that the present working directory

will be above your classes directory.

so if this class is in

C:\\projects\garage\net\javagarage\etc...

Put the image you want in C:\\projects\garage.

*/


String imagePath = System.getProperty("user.dir") +

"\\kitty.gif";

//get the image that will prefix each tab

ImageIcon icon = new ImageIcon(imagePath);


//create a new panel using our custom

//utility class

JComponent noodlePanel = new TabPanel("prissy

one").getPanel();

//add a tab to the panel

tabbedPane.addTab("Noodlehead", icon, noodlePanel);


JComponent doodlePanel = new

TabPanel("princess").getPanel();

tabbedPane.addTab("Doodlehead", icon, doodlePanel);


JComponent mrPanel = new TabPanel("the boy

kitty").getPanel();

tabbedPane.addTab("Mr. Apache Tomcat", icon, mrPanel);

//add the tabbed pane

add(tabbedPane);


//scrolling tabs makes small navigation arrows for

//use when all tabs do not fit on the screen
```

```
tabbedPane.setTabLayoutPolicy(JTabbedPane.SCROLL_TAB_

LAYOUT);

}



//Makes the icon of the image file we provide, to

//include on each tab. If no file is specified or

//found, null is returned, and no icon is included.

protected static ImageIcon createIcon(String path) {

try {

imageURL = new URL(path);

return new ImageIcon(imageURL);

} catch (MalformedURLException me){

System.err.println("This is not an acceptable URL: " +

path + " \n" + me.getMessage());

return null;

}

}

/**

 * Put the application on the event-handling thread

 * and start it up.

 */

private static void launch() {

//must be called first, and static-ly

JFrame.setDefaultLookAndFeelDecorated(true);

//Create and set up the window.

JFrame frame = new JFrame("Tabbed Pane App");

frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

//create a new instance of this class and

//make the contents visible

JComponent newContentPane = new TabbedPaneDemo();

newContentPane.setOpaque(true);

frame.getContentPane().add(newContentPane,

BorderLayout.CENTER);

//show it

frame.pack();
frame.setVisible(true);

}


public static void main(String[] args) {

//run the app as recommended

SwingUtilities.invokeLater(new Runnable() {

public void run() {

launch();

}

});

}

}


/**

 * Class to encapsulate the business of creating panels

 * that hold the content of the application (in this

 * case just a line of text). The tabs tack on to the

 * top of these panels.

 */

class TabPanel {

JPanel panel;

JLabel filler;

public TabPanel(String text){

panel = new JPanel(false);

filler = new JLabel(text);

filler.setHorizontalAlignment(JLabel.CENTER);

panel.setPreferredSize(new Dimension(250,100));

panel.add(filler);

}


public JComponent getPanel() {

return panel;

}

}


//end TabbedPaneDemo
```

} //end components.stepInto()

LayoutManagers Overview

LayoutManager.getDefinition(): //returns: Layout managers provide different metaphors for organizing the atomic components within a content pane. Layout Managers include FlowLayout, BorderLayout, BoxLayout, CardLayout, GridLayout, and GridBagLayout. You can create your own layout managers, or use a null layout manager, which requires you to position each component using absolute pixel values. FlowLayout is used as the default layout manager if you do not specify a particular one you'd like to use.

LayoutManager.stepInto() { We will step into layout managers for a moment here so that we can forget about them for the rest of the topic. They are the underlying arranger of the space, so it is appropriate to start with them. Just don't get too bogged down by them. I'll try to keep it peppy.

If you have used HTML to make a Web page, using layout managers is a bit like using tables and CSS to position different elements on your page. Layout managers dictate (usually) the size and position of components within the UI.

You set the layout of a panel using the setLayout() method, as in the following:

```
//get the main content pane of your frame:

Container pane = frame.getContentPane();


//make this pane to use border layout

pane.setLayout(new BorderLayout());


//put a component in a region

pane.add(new JtextField(10), BorderLayout.EAST);
```

Let's look at each layout manager. To do so, we will create a totally generic frame using the BasicFrame class so we don't have to retype a bunch of code for each example.

BasicFrame.java

```
package net.javagarage.demo.swing.layouts;


import javax.swing.JFrame;

/** <p>

Just for use in demo-ing different Layout

managers. Implements the singleton pattern.
```

```
**/
public class BasicFrame extends JFrame {

private static BasicFrame frame;

private BasicFrame() {
super("My Frame");

}

public static BasicFrame getInstance() {
if (frame == null)
frame = new BasicFrame();
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
return frame;

}
}
```

Switch (LAYOUT_MANAGERS) {

Now we look at different ways to organize controls on your panels.

Case: FLOW LAYOUT :

This is the simplest to use, and is used in many Swing examples in this book. In part, that's because you don't have to do much to make it go, and it is used by default by Jpanels. It arranges components in a flow—that is, by adding them in a certain direction. The effect is similar to typing into a word processor: as you add new words, they just get tacked onto the end. FlowLayout has three constructors.

1. FlowLayout()

   Nuff said.

   There are two component orientations possible for a Flow Layout: left to right and right to left. They are

   ```
   ComponentOrientation.LEFT_TO_RIGHT

   ComponentOrientation.RIGHT_TO_LEFT
   ```

   You specify which you want to use like this:

   ```
   contentPane.setComponentOrientation(

   ComponentOrientation.RIGHT_TO_LEFT);
   ```

2. FlowLayout(int alignment)

   Three alignments can specify the orientation in the FlowLayout constructor. They are

   ```
   FlowLayout.LEADING (components should be left-aligned)

   FlowLayout.CENTER

   FlowLayout.TRAILING (components should be right-aligned)
   ```

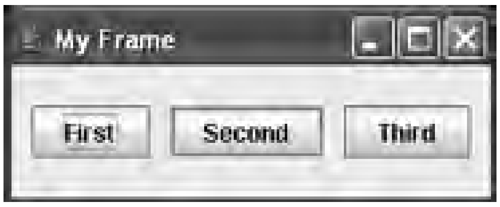
3. public FlowLayout(int alignment, int horizontalGap, int verticalGap)

   The horizontal and vertical gap parameters allow you to specify the number of pixels to put between each component. The default gap is 5.

FlowLayoutExample.java

```
package net.javagarage.demo.swing.layouts;

import java.awt.Container;
import java.awt.FlowLayout;
import javax.swing.JButton;

/**<p>
 * Shows how to use a FlowLayout
 * with an alignment and horizontal and
 * vertical gaps
 * between each component.
 * </p>
 * @author Eben Hewitt
 **/
public class FlowLayoutExample {
public static void main(String[] args) {
BasicFrame frame = BasicFrame.getInstance();
Container pane = frame.getContentPane();

int hGap = 10;
int vGap = 20;
pane.setLayout(new FlowLayout(FlowLayout.RIGHT,
        hGap, vGap));

pane.add(new JButton("First"));
pane.add(new JButton("Second"));
pane.add(new JButton("Third"));

frame.pack();
frame.setVisible(true);
}
}
```

Figure 25.2. A flow layout with the mouse hovering over the second button.



Case: BORDER LAYOUT :

Border layout resizes and arranges the components within it to fit in five different regions: PAGE_START, LINE_START, CENTER, LINE_END, and PAGE_END. If you don't specify a constant indicating the region onto which you want to place a component, it will be added to the center.

The placement of the regions looks like this:

PAGE_START

LINE_START CENTER LINE_END

PAGE_END

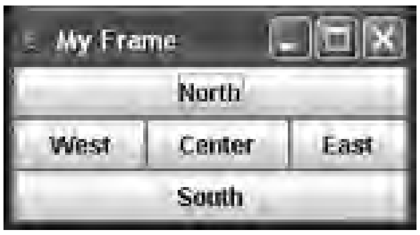
Components in a Border layout are layed out according to their preferred size. That means that filling each region with a JButton will allow each component to fill up the space, but adding JLabels to each region will not, as the label's preferred size is just large enough to hold its data.

BorderLayoutExample.java

```
package net.javagarage.demo.swing.layouts;


import java.awt.BorderLayout;

import java.awt.Container;

import javax.swing.JButton;

import javax.swing.JLabel;


/**

 * Demonstrate using border layout.

 * @author Eben Hewitt

 * @see BasicFrame

 **/

public class BorderLayoutExample {
```

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

BasicFrame frame = BasicFrame.getInstance();

Container pane = frame.getContentPane();

pane.setLayout(new BorderLayout());
pane.add(new JButton("Center"), BorderLayout.CENTER);
pane.add(new JButton("North"), BorderLayout.NORTH);
pane.add(new JButton("South"), BorderLayout.SOUTH);
pane.add(new JButton("East"), BorderLayout.EAST);
pane.add(new JButton("West"), BorderLayout.WEST);

frame.pack();
frame.setVisible(true);
}
}//end BorderLayoutExample
```

Figure 25.3. The border layout.



Case: BOX LAYOUT :

This layout manager arranges components from one of two directions: horizontally or vertically. The components in each region will not wrap, so if you resize the window, the components will stay arranged as you specified.

The BoxLayout class features only one constructor.

```
BoxLayout(Container cont, int axis);
```

You create a box layout specifying the orientation (axis) you want to use. It dictates along what kind of line your components will be layed out. You have four choices.

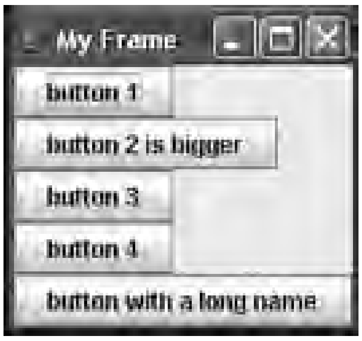
1. X_AXIS: Add components horizontally, from left to right.
2. Y_AXIS: Add components vertically, from top to bottom.
3. LINE_AXIS: Components should be layed out according to the value of the container's ComponentOrientation property. That is, if the container's ComponentOrientation is horizontal, components are layed out horizontally.
4. PAGE_AXIS: Components should be layed out according to the value of the container's ComponentOrientation property. If the container's orientation is horizontal, components are layed out vertically; otherwise, they are layed out horizontally, and components are layed out left to right. For vertical orientations, components are always layed out from top to bottom.

The first argument to the constructor specifies the container that you want to stick the layout to.

BoxLayoutExample.java

```
package net.javagarage.demo.swing.layouts;


import java.awt.Container;

import javax.swing.BoxLayout;

import javax.swing.JButton;


//demos a simple box layout

public class BoxLayoutExample {


public static void main(String[] args) {

doSimple();

}


static void doSimple(){

BasicFrame frame = BasicFrame.getInstance();

Container pane = frame.getContentPane();


//when you add components, they will be

//added vertically, from top to bottom

pane.setLayout(new BoxLayout(pane,

BoxLayout.Y_AXIS));


pane.add(new JButton("button 1"));

pane.add(new JButton("button 2 is bigger"));
pane.add(new JButton("button 3"));

pane.add(new JButton("button 4"));

pane.add(new JButton("button with a long name"));


frame.pack();

frame.setVisible(true);

}

} //end of simple box layout
```

Figure 25.4. A simple box layout.



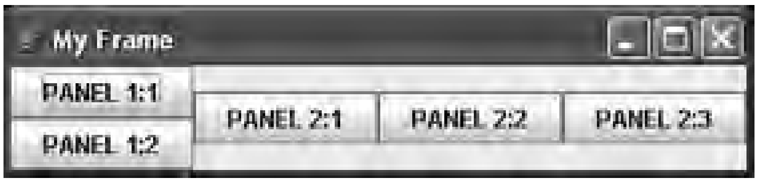
Note that you can nest box layouts to achieve more sophisticated effects, and exert greater control over how your components are arranged. What follows is an example of how to do just that sort of thing.

NestedBoxLayout.java

```
package net.javagarage.demo.swing.layouts;


import java.awt.Container;

import javax.swing.BoxLayout;

import javax.swing.JButton;

import javax.swing.JPanel;


// demos nested box layouts

public class NestedBoxLayouts {
```

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

BasicFrame frame = BasicFrame.getInstance();

Container pane = frame.getContentPane();


//X == Horizontal

pane.setLayout(new BoxLayout(pane,

BoxLayout.X_AXIS));


//we're going to add two panels to the

//content pane

JPanel p1 = new JPanel();

JPanel p2 = new JPanel();


//so when we add the panels, they will go side

//by side (horizontally)

pane.add(p1);

pane.add(p2);


//now make each panel have its OWN box

//layout panel one (the left panel)

//will arrange its components vertically (Y)

p1.setLayout(new BoxLayout(p1, BoxLayout.Y_AXIS));


//panel 2, the right panel, will arrange

//horizontally, from left to right

p2.setLayout(new BoxLayout(p2, BoxLayout.X_AXIS));

//add two buttons to the left panel

//notate buttons like this:

//Panel Num : Button Num

p1.add(new JButton("PANEL 1:1"));

p1.add(new JButton("PANEL 1:2"));


//add three buttons to the right panel

p2.add(new JButton("PANEL 2:1"));

p2.add(new JButton("PANEL 2:2"));

p2.add(new JButton("PANEL 2:3"));


frame.pack();
frame.setVisible(true);

}

} //end NestedBoxLayout
```

Figure 25.5. One box layout nested within another allows more sophistication.



Note that we have nested layouts of the same kind here to demonstrate how to do it so you get the control you need. But you don't have to nest layouts of the same type. You can nest a grid layout inside a box layout inside a flow layout if you want. It is common to nest layouts to get the functionality, flexibility, and control required.

Case: CARD LAYOUT:

A card layout treats each component in the container as a card on a stack. Just as with a deck of cards, only one card is visible at a time. As with much in Swing programming, the order of invocation matters; the first panel added to the layout is what is displayed when the frame is first shown.

The CardLayout class offers a set of methods to flip through the cards sequentially, or to show a particular card to show a panel based on its String name. This is useful in working with weezards (step-by-step panels the user must go through). Note that Swing also gives you for free the tabbed pane, which is covered next with an illustrative example in case you need that kind of specific functionality. In fact, using a tabbed pane is easier than using card layout, because a JTabbedPane implements its own layout.

Following are the steps to make this layout go:

```
//create a panel to hold all the cards

JPanel cards;

final static String PANEL_ONE = "My first panel";

final static String PANEL_TWO = "My second panel";


//...

//now make each card, which is just a JPanel

JPanel card1 = new JPanel();

JPanel card2 = new JPanel();


//...


//initialize the panel that holds the cards
```

```
cards = new JPanel(new CardLayout());


//they get String names here

cards.add(card1, PANEL_ONE);

cards.add(card2, PANEL_TWO);


//Now to show a card, you get the layout from the

cards JPanel

CardLayout layout = (CardLayout)(cards.getLayout());


//call the show() method and pass it the String name

layout.show(cards, PANEL_TWO);
```

That's all you need to do. The following methods of the CardLayout class let you choose panes:

```
void first(Container)

void next(Container)

void previous(Container)

void last(Container)

void show(Container, String)
```

Again, unless you're doing a wizard, it's generally easier to use a JTabbedPane to get the same effect.

Case: GRID LAYOUT:

Grid layout presents components within a grid. Each cell in a grid is the same size. You place one component in each cell. Components within a grid cell take up all of the available space in the cell.

There are three constructors available for creating a grid layout.

- public GridLayout() Creates a grid with a single row, and one column for each component.
- public GridLayout(int rows, int cols) Creates a grid with the specified number of rows and columns. Sometimes.
- public GridLayout(int rows, int cols, int hGap, int vGap) Creates a grid with the specified number of rows and columns (sometimes), and adds space between the components.

Note that you can also set the number of rows and columns using the methods setRows() and setColumns(). However, the number of columns you specify is ignored completely if the value you specify is non-zero. Whether you use setColumns() or pass them into the constructor. In this case, the number of columns is determined by the number of components you have over the number of rows. Huh?

If I create a grid layout and create six components with five rows, I have more components than rows. So the layout will spill over, creating two columns. It then will distribute the components as evenly as possible over each column—ignoring my request to set a certain number of columns. I wind up with two columns of five rows each, with components in each of the first three rows of both columns, and empty cells at locations (4,0), (4,1), (5,0), (5,1).

These methods will throw an IllegalArgumentException if both the rows and columns are set to 0.

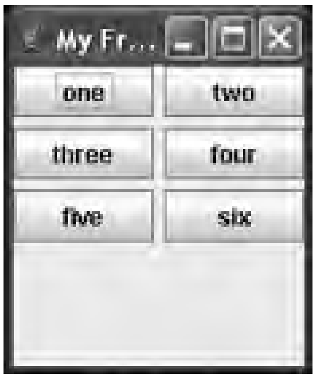
Here is an example.

GridLayoutExample.java

```
package net.javagarage.demo.swing.layouts;


import java.awt.Container;

import java.awt.GridLayout;


import javax.swing.JButton;


//demo how

public class GridLayoutExample {

static int rows, cols, hGap, vGap;


public static void main(String[] args) {

BasicFrame frame = BasicFrame.getInstance();

Container pane = frame.getContentPane();

//let's have 5s—all around!

rows = cols = hGap = vGap = 5;


//create layout: notice how my pleas

//for certain number of columns is viciously

//ignored, laughed at even.

pane.setLayout(new GridLayout(rows, cols, hGap, vGap));


//add components

pane.add(new JButton(" one "));

pane.add(new JButton(" two "));

pane.add(new JButton(" three "));

pane.add(new JButton(" four "));

pane.add(new JButton(" five "));

pane.add(new JButton(" six "));
//we have space left over for two more

//in each column because we have 10 cells:

//5 rows * 2 columns


frame.pack();

frame.setVisible(true);

}

}

//end of Grid Layout example
```

Notice that if you resize the window, the layout does not shift. You still have the number of columns the layout manager created.

Figure 25.6. The grid layout.



Case: GRID BAG LAYOUT:

FRIDGE

You have to be careful in keeping track of your grid. If you add components into the same cell, they can overwrite each other in the display, causing one to be in front of the other. This is not only undesirable in itself, but can cause strange behavior. For example, if you resize the window, you might find the higher z-indexed component flipping in front of the other.

GridBagLayout (a.k.a. GBL) can be very tricky to work with. It is far more complex to create, manage, keep track of, and modify than other layout types. Things sort of come down to two choices in Swing apps: nest a number of other layout types within each other as necessary, or use GridBagLayout.

Now that we're scared, all a GBL does is allow the placement of components in a grid of rows and columns (so far so good, it's just like Grid Layout, we can do this, okay, what's next?...), *allowing components to span multiple rows and/or columns* (nnnnnoooooooooo!!!!!!!).

Not all columns have to have the same width, and not all rows have to have the same height.

This is actually a lot like nested HTML tables, isn't it? It is enough like them that we'll be able to manage it. If you can write an HTML page that nests tables within tables to control your layout at a granular level, you are a lot of the way home with GBL. The problem turns out to be not so much the GBL itself, but the fact that the layout gets the size of the cells in the grid from the preferred size of the components within it. For that reason, you will likely have to do some experimenting as you work with GBL, because this changes across components. For example, a button will fill up all of the available space, whereas a label will only take what it needs bare minimum.

Constraints

The GBL specifies the size and position of each component using constraints. Constraints are objects of type GridBagConstraint. You don't need a new GridBagConstraint for each component; you can reuse a constraint if it's convenient.

When you create a constraint, you can set the following fields: gridx and gridy.

Specify the row and column at the upper left of the component, with coordinates (0,0) representing the upper-left column. X represents the horizontal plane and Y the vertical plane. You can also use the default value GridBagConstraints.RELATIVE to indicate that the component should be placed immediately to the right of or immediately below the component that was added to the container just before this component was added.

- Gridwidth Specify the number of columns in the display. Default is 1.
- Gridheight Specify the number of rows in the display. Default is 1.
- Fill Used to indicate the manner in which to resize the component when the component's display area is larger than the component's requested size.

  Possible values:

  FRIDGE

  GridBagLayout allows a component to span multiple rows only when the component is in the column farthest to the left, or the component has positive gridx and gridy values.

  NONE Default.

  HORIZONTAL The component will entirely fill its display area horizontally. The height will not change.

  VERTICAL The component will entirely fill its display area vertically. The width will not change.

  BOTH The component will fill its entire display area.
- ipadx, ipady Specifies how much padding to add to the minimum size of the component. Default is 0.
- Insets Specifies the minimum amount of space between the component and the edges of its display area (internal padding). The value is specified as an object of type Insets. By default, components have no external padding.
- Anchor Use to determine where to place your component when the component is smaller than its display area. Valid values are CENTER (default), PAGE_START, PAGE_END, LINE_START, LINE_END, FIRST_LINE_START, FIRST_LINE_END, LAST_LINE_END, and LAST_LINE_START.
- weightx, weighty Use the values to specify how to distribute space among columns (weightx) and among rows (weighty) during resizing.

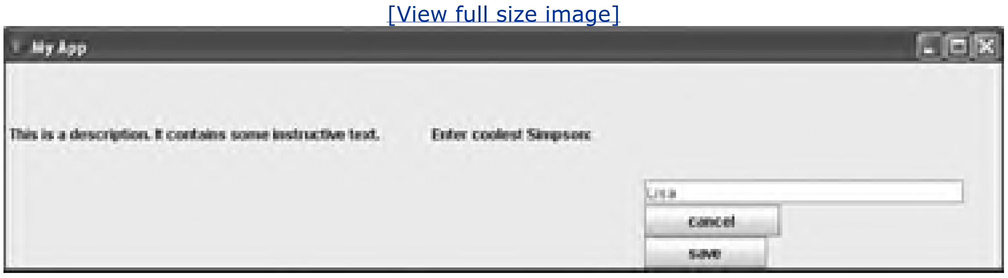
GridBagLayoutExample.java

```
package net.javagarage.demo.swing.layouts;

import java.awt.Container;
import java.awt.Dimension;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;

import javax.swing.Box;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;

//you could do this to save typing:
//import static javax.swing.GridBagConstraints.*;
//i don't here in order to be explicit

//demos a grid bag layout
public class GridBagLayoutExample {

public static void main(String[] args) {
doSimple();
}

static void doSimple(){
//BasicFrame frame = BasicFrame.getInstance();
JFrame frame = new JFrame("My App");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
Container pane = frame.getContentPane();

//create the layout and constraints
GridBagLayout layout = new GridBagLayout();
GridBagConstraints constraints = new GridBagConstraints();
frame.setLayout(layout);

//create all of the controls we will add
JLabel lblDesc = new JLabel(" This is a description. " +
"It contains some instructive text. ");

JLabel lblEnter = new JLabel("Enter coolest Simpson: ");

JTextField txtField1 = new JTextField(20);

JButton btnSave = new JButton("save");

JButton btnCancel = new JButton("cancel");


//now create new constraints for each component

//before adding it to the pane


//constrain Description Label

//gives height to the whole area

constraints.ipady = 50;

//puts space around the label

constraints.ipadx = 20;


layout.setConstraints(lblDesc, constraints);

pane.add(lblDesc);


//constrain Enter label

constraints.ipady = GridBagConstraints.NONE;

layout.setConstraints(lblEnter, constraints);

pane.add(lblEnter);


//constrain Text Field

constraints.gridwidth = GridBagConstraints.CENTER;

constraints.anchor =

GridBagConstraints.LAST_LINE_START;

layout.setConstraints(txtField1, constraints);


pane.add(txtField1);


//constrain Cancel Button

constraints.gridx = 2;

//add constraints to Cancel button

layout.setConstraints(btnCancel, constraints);

//add Cancel button to the pane
```

```
pane.add(btnCancel);


pane.add(Box.createRigidArea(new Dimension(30,0)));


//constrain Save button
constraints.gridx = 3;
constraints.gridy = 3;


layout.setConstraints(btnSave, constraints);
pane.add(btnSave);


frame.setSize(frame.getPreferredSize());


frame.pack();
frame.setVisible(true);
}
}
```

Figure 25.7. The grid bag layout in action.

[View full size image]



} //end LayoutManagers.stepInto();

Handling Action Events

Examples in the Toolkit, such as the GaragePad doodler, show how to capture mouse motion events. There are also examples in the doodler application and the text editor that show how to handle action events from menu items when they're clicked.

In Swing, you create user controls for the user to interact with. When the user does something to a control—for example, clicks a button—that action fires an event. You create listeners that do something when that event is fired, and then attach them to the user controls. Actions can optionally hold information about tool tip text or icons, though this is rare.

Specifically, to handle an event you override the `actionPerformed()` method.

```
public void actionPerformed(ActionEvent evt) {

        // do something

}
```

There are a couple of different ways to do this. Please see the blog on Inner Classes earlier in the book for a discussion and demonstration of implementing action listeners. Please see the Toolkit in the back of this book for numerous, complete, working examples of handling action events.

Chapter 26. BLOG ENTRY: SOFTWARE DEVELOPMENT BLACK MARKET

DO OR DIE:

- Star Date 8675309

There is not one thing called a software development department. Or the "Programming Division of IS." Or whatever it is called at your company. Each department is very different whether it employs 3 programmers, or 30, or 3,000. It's different yet again whether you help sell books for an e-commerce site, make garage doors or thermostats work, help create a program that gets burned onto CDs and shrink-wrapped and sold in Wal-Marts throughout the world, or make, automate, and integrate business applications for a government entity. But in each of these very different places seethes a very real and very scary possibility—the possibility that your team is slowly bled for resources (people, money, tools, access, and so on). We have all felt it. That's bad for us, and we have all talked it to death because it's a drag. What we don't talk about as much is what happens to the organization when it starves us like that. What happens should be no surprise. The same thing happens in the world when people are starved of resources, money, tools, and access to things they need. Think about books that have been banned. Or the prohibition on alcohol in the 1920s. Or drugs of all kinds. These things don't cease to circulate because they've been outlawed. They still thrive. But they thrive in a black market. Which is a very dangerous place. Maybe you've never experienced this developer's black market. Maybe the reason that you've stayed at your same company for 10 years (ha) is that they lavish you with time, money, resources, tools, people, and access. But I rather doubt it.

A black market is where people illicitly acquire and distribute goods and services.

Do you work in a developer's black market? Ask yourself the following. Have you ever run down the hall to the offices of your friends down in networking and asked for write permissions on a directory somewhere on the network? You just need this quick fix, just this once, then your app will work and you can all go to lunch, you tell her. You explain that there is not a security policy in place but this would do the trick and your manager is on a conference call and you can't move forward until that directory is available for your app to write to. This workplace has a black market.

If you call someone in your organization that works in a related department and get him to do a favor for you to help make your application work, you have a black market. Your organization's software is dependent on the personal good will that its workers generate toward others, or, if you are not an optimist, how easy it is for you to manipulate or coerce your coworkers.

You shouldn't have to cut deals to get a project done. If you have to cut deals to get your project done, it is because your organization is asking for something that they can't afford or don't want to pay for.

In my view, you're going to either pay now, or pay later. And when you pay later, the cost is always, without fail, far, far more. This is true for testing and commenting and debugging too.

If a company lets its departments run on the black market, it is going to pay later.

If your department's product depends on a black market, you must help stop it immediately. Determine what you need to get your job done. Not what they want to hear. What you really need. The real hours. The real cost of 14 copies of that IDE. The real number of programmers. These things we are used to asking for. We do this a lot. It's easy to put them into MS Project and everyone can print it out and think they have a plan. Typically, this is the culprit right here.

Just like no one in their right minds would think that a significant set of ideas can be satisfactorily represented in a whiz-bang PowerPoint presentation, don't think for a second that the fundamental coherence of a plan can be represented in MS Project.

I am not bashing Microsoft here. Those applications are great for what they do. I am saying that they don't do what we hope they would: they don't give us a plan anymore than Word writes the great American novel simply because it has a (flawed) grammar checker and lets you type.

Do not mindlessly assent to working on incoherent projects. You will become miserable. Ultimately you will quit and try to sell your family on how exciting Fresno really can be since they have a new programmer position waiting for you there.

Impress upon your managers the importance, the necessity, of a long-term, coherent plan.

Do not make one-off apps in the dark. Consider long-term architecture. Make modular applications that allow you to maximize the re-use of code. This kind of thing takes planning, yes. But it also takes knowledge of what you're going to do next, so that you can see what you might want to re-use.

Ensure that changes or architecture you want to introduce will have a visible, positive impact on customers. If it doesn't help your customers, you have a very hard case to make.

Yes, business changes. Yes, the business managers cannot predict where the market will go. They don't have a crystal ball. No one is asking them to predict the future. Many times, what project you do next depends on whether you get a certain grant, or whether a certain bid is successful. Obviously.

If you are working on business applications or creating a product that might participate in a suite of products, there are things you need to know. You need support from people with power regarding your plans. The people in power don't know this. Let them know it.

It is impossible to build a service-oriented architecture in an organization without a long-term plan and the knowledge, input, and support of the business people who make decisions about your overall long-term goals. It is likely that they do not know or care what a service oriented architecture is. Show them the benefit, or the necessity, and tell them you need some information. Because you can't string apps together with tape.

You need to consider a framework, and cross-cutting features such as authentication, authorization, customization, interoperability, platform support and reliance, and so on. And if your job is to write code, you almost certainly don't have the answers to these questions on your own.

We in IT departments are in a tough spot. It is polite for us to suggest that we only exist to support the larger goals of our government, or our basketball shoe-making company, or even our software development company. This is obviously, trivially true. It is not the whole story. Business people do not think of us as a real department, such as Finance or Human Resources. They think of us as a meta-department. An expense. As if Human Resources or Finance or the Police Department is not an expense.

If you work on business apps or infrastructure, you are not simply in product development. You're serving two masters, each of which only knows the other by its sordid reputation. And that politesse can actually be a disservice to the business. Demand to participate in the goals. Because IT is different: It is a separate entity and yet a meta-entity that touches everyone every day.

Demand that the stake holders in an application are present and communicative. Software projects fail when organizations think that it is only the job of the software developers to make software. Stakeholders must participate. Management must provide resources and get behind projects. Customers must use the software.

The IT department must understand the long-term goals of an operation, and make a long-term IT plan to support those goals. Often we go this far. But then we start working on our own and a black market springs up. Go a step further. Go back to the business people and get their buy and support of your plan. Route out the black market.

Chapter 27. DATES AND TIMES

TO DO LIST:

- Dry cleaning
- Pick up the girl from soccer
- Put Java Dates and Times to good use

Here we go, into the land of dates and times. Dates are not the sorts of things I think are fun. I find dates cruel, punishing, and difficult to wield in different languages. It's surprising. I would think that dates would be fairly straightforward—after all, we've had dates for a long time. Our various calendars are more or less elegant, but their implementation and use in programming languages often seems to fall short of the mark. They are complicated, messy things.

Everybody knows what dates are and I wager that few readers are interested in knowing how and why they work the way they do in Java. I wager that you want a few utility methods that will just help you solve your current problem, no questions asked. If I'm wrong, the Java API makes terrific, suspense-filled reading.

Dates

In this topic, I will just present a few classes that do a lot of different things with dates. You can see what kinds of classes, methods, and fields are available to you, and hopefully you can just copy and paste methods here with little modification into your own projects.

Dates.java

```
package net.dates;

import java.util.*;
import java.text.*;

/**
This class contains a number of useful methods
that make it convenient to do common operations
involving dates. It uses java.util.Calendar and
java.util.GregorianCalendar to do so.
<p>
Hopefully you can paste this
directly into your own utilities library.
<p>
Just modify the classes to accept calendar objects
and return the int values. I didn't do this here
so that it would make the code easier to read.
<p>
See the Java API documentation for Calendar and
GregorianCalendar for many examples of usage.
*/

public class Dates {

Calendar calendar = Calendar.getInstance();


public static void main(String...ar) {
    numberOfDaysInMonth();

    dayOfWeek();

    getDateDifference();

}




//find out the number of days in a month of a given year

public static void numberOfDaysInMonth() {


//make calendar of the month

//you could pass this in as a parameter

    Calendar c = new GregorianCalendar(2004,

                Calendar.JULY, 1);


    //get number of days in this month

    int days = c.getActualMaximum(Calendar.DAY_OF_MONTH);


log(days); //31


    //month in a leap year

    c = new GregorianCalendar(2004,

                Calendar.FEBRUARY, 1);

    days = c.getActualMaximum(Calendar.DAY_OF_MONTH);


log(days);// 29


}


//gets the number of years difference from

//some past date and now

public static void getDateDifference() {


    Calendar now = Calendar.getInstance();


//the date we want to compare to now: Shakespeare is born

    Calendar compareDate =
new GregorianCalendar(1564, Calendar.APRIL, 23);


  //get difference based on year

  int yearDifference = now.get(Calendar.YEAR) -

compareDate.get(Calendar.YEAR);


log(yearDifference);//440 years ago


}




//The day-of-week is an integer:

//1 (Sunday), through 7 (Saturday)

public static void dayOfWeek() {


  Calendar anv = new GregorianCalendar(2004,

          Calendar.MARCH, 27);

  int dayOfWeek = anv.get(Calendar.DAY_OF_WEEK); // 7


log (dayOfWeek);


}


public String getDate() {

    return getMonthInt() + "/" + getDayOfMonth() +

        "/" + getYear();

  }

/**

   * Useful for returning dates in a style

   * presentable to end users.

   * Returns date in this format:

   * Thursday, October 10, 2002

   * </p>

   */

  public String getLongStyleDate() {

    String myDate =

        getDay() + ", " + getMonth() + " "

        + getDayOfMonth() + ", " + getYear();

    return myDate;
    }

    public String getSlashDate() {
        String myDate = getMonthInt() + "/"
                + getDayOfMonth() + "/" + getYear();
        return myDate;
    }

        ///returns the year
    public int getYear() {
        return calendar.get(Calendar.YEAR);
    }
        ///gets the month as a String
public String getMonth() {
        int mo = getMonthInt();
        String[] months = { "January",
                "February", "March",
                "April", "May", "June",
                "July", "August", "September",
                "October", "November", "December" };
        if (mo > 12)
            return "?";

        return months[mo - 1];
    }
        //gets month as int on scale of 1 - 12
        //it is 0 - 11 by default
public int getMonthInt() {
    return calendar.get(Calendar.MONTH) + 1;
    }

public int getDayOfMonth() {
        return calendar.get(Calendar.DAY_OF_MONTH);
    }
        //gets the day of week as a string
public String getDay() {
          int d = getDayOfWeek();
          String[] days = {"Sunday", "Monday",
                    "Tuesday", "Wednesday",
               "Thursday", "Friday", "Saturday"};

          if (d > 7)
              return "error";

      return days[d - 1];
}

public int getDayOfWeek() {
          return calendar.get(Calendar.DAY_OF_WEEK);
}

public int getEra() {
          return calendar.get(Calendar.ERA);
}

public String getUSTimeZone() {
          String[] zones = {"Hawaii", "Alaskan", "Pacific",
                         "Arizona", "Mountain", "Central",
                         "Eastern"};
          return zones[10 + getZoneOffset()];
      }
public int getZoneOffset() {
       return calendar.get(Calendar.ZONE_OFFSET)/
                           (60*60*1000);
}

public int getDSTOffset() {
          return calendar.get
                    (Calendar.DST_OFFSET)/(60*60*1000);
}

public int getAMPM() {
          return calendar.get(Calendar.AM_PM);
}

public String getTime() {
          return getHour() + ":" + getMinute() + ":" +
                 getSecond();
```

```
}

public int getHour() {
     return calendar.get(Calendar.HOUR_OF_DAY);
}

public int getMinute() {
     return calendar.get(Calendar.MINUTE);
}

public int getSecond() {
     return calendar.get(Calendar.SECOND);
}

public static String getDateAsString(Date dt) {
     return getDateAsString(dt, DateFormat.SHORT);
}

public static String getDateAsString(Date dt, int
                      format) {
     if( dt == null )
         return "";

     return DateFormat.getDateInstance(format)
                   .format(dt);
}

public String getDateTime() {
     return getDate() + " " + getTime();
}

public static java.util.Date getDate(String str)
                              throws ParseException {
     return
     DateFormat.getDateInstance(DateFormat.SHORT).
               parse(str);
}

    /**
```

```
     * <p>Useful for getting the date in a format
     * useable by JDBC.
     * Returns current date in this format:
     * 2004-03-21 for March 21st, 2004
     * </p>
     */
public static java.sql.Date getCurrentJDBCDate(){
     java.sql.Date date = new java.sql.Date(
                (new java.util.Date()).getTime());
     return date;
  }

public static java.sql.Date
getJDBCDate(java.util.Date dt) {
     if( dt == null ) {
         return null;
     }
     return new java.sql.Date(dt.getTime());
}

     //will print out anything we pass it
     //just a convenience to save typing
private static <T> void log(T msg) {
     System.out.println(msg);
}

}
```

Notice that the preceding class has a number of methods that make it convenient to use dates in various situations: to print to the screen in a user-friendly way, to pass to an SQL database (most of these methods store dates in a manner different than Java does), and for representing dates in different formats appropriate for different occasions.

Time

The following class contains methods for formatting times in ways you are likely to want to do.

Times.java

```
package net.dates;

import java.util.*;

import java.text.*;


public class Times {


 Calendar calendar = Calendar.getInstance();


    public static void main(String...a) {

    log(get24HourTime() );

    log(getAMPMTime() );

    log( getSimpleTime() );

  }


/**

    * Returns time in this format:

    * 14.36.33

    */

   public static String get24HourTime() {

   Date d = new Date();

   Format formatter = new SimpleDateFormat("HH.mm.ss");

   return formatter.format(d);

   }

   /**

    * Returns time in this format:

    * 01:12:53 AM

    */

   public static String getAMPMTime() {
```

```
  Format formatter = new SimpleDateFormat("hh:mm:ss a");
  return formatter.format(new Date());
  }

  public static String getSimpleTime() {
    Format formatter = new SimpleDateFormat("h:mm a");
      return formatter.format(new Date());
  }

public int getHour() {
    return calendar.get(Calendar.HOUR_OF_DAY);
  }

  public int getMinute() {
    return calendar.get(Calendar.MINUTE);
  }

  public int getSecond() {
    return calendar.get(Calendar.SECOND);
  }

  public String getTime() {
    return getHour() + ":" + getMinute() + ":" +
         getSecond();
  }

    //will print out anything we pass it
    //just a convenience
  private static <T> void log(T msg) {
    System.out.println(msg);
  }
}
```

Running this class outputs something like this, depending on what time it is:

```
18.25.04

06:25:04 PM

6:25 PM
```

This class not only offers some utility methods that you can put to use in your own applications, but it makes use of the Format class and its subclasses in the java.text package (such as SimpleDateFormat). I recommend checking out the API documentation for these classes to find out about more usage options.

Chapter 28. USING TIMER TASKS

DO OR DIE:

- Start a task to run repeatedly at a certain interval
- Start a task after a specified delay
- Start a task to run at a specific time

This topic elucidates the use of tasks, made available to us via the java.util.TimerTask API. This is a wonderfully practical and easy-to-use API.

Using TimerTask to Do Stuff at Intervals

If you've ever used a UNIX cron job, you're used to needing to do stuff at timed intervals. Say you want to gzip the log files and move them in an archive every night at 3 a.m. Or maybe you need to start a process that will go out and check a server for some kind of update, and you want to check it every hour.

I once wrote an application that received user signups, and wrote them to a file. There were a lot of signups every day, but there wasn't a need to act upon them immediately. So the client asked that every hour we would do some junk to the data and then transfer it over to their AS/400 via FTP. Then they would do some junk to the data over there, and my app would check every couple of hours to see if they had written out the new file, and if they had, fetch the whole thing. So, anyway, if you need to do something repeatedly, or start doing it after a delay, java.util.TimerTask is the ticket.

We'll look at both ways in the upcoming pages.

Scheduling a Task to Start After a Delay

There are a few different ways to use the TimerTask to schedule and repeat jobs. You can specify to delay the start of a task using a millisecond delay, or by specifying a date.

DelayTask.java

```
package net.javagarage.tasks;


import java.util.Timer;

import java.util.TimerTask;


/**

 * Uses java.util.Timer schedules the task to start

 * running three seconds after it's instantiated.

 * @author eben hewitt

 */

public class DelayTask {

//hold it in a member so everybody can get to it

protected Timer timer;


//public constructor accepts the

//number of milliseconds to delay before running

//the task

public DelayTask(int msDelay){

timer = new Timer();

//pass the schedule method the task that you want
//to start, and the delay

timer.schedule(new PrintOutTask(), msDelay);

}


//using inner class to get to the timer member

//make it extend TimerTask

class PrintOutTask extends TimerTask{

//we must implement the run() method

public void run(){

//do the work here...

System.out.println("Task is doing some work...");

//terminates and discards this timer

timer.cancel();

System.out.println("All done.");

}

}


public static void main(String[] args) {

System.out.println("Scheduling task...");

new DelayTask(3000);

System.out.println("The task has been scheduled.");

}

}
```

The output initially is

```
Scheduling task...

The task has been scheduled.
```

And then after a three second delay, the following is printed:

```
Task is doing some work...

All done.
```

So the main thing that you need to do to schedule a task is as follows:

- Create a class that extends TimerTask.
- In that class, implement the inherited run() method, and do your work in there.
- Create an instance of the Timer class. This starts a new thread.
- Create an instance of the class that extends TimerTask (here, called PrintOutTask, because his job is to print out statements).
- Schedule the task to start.
- When you need to stop the task, call the timer's cancel() method.

You can schedule a task to begin after a delay, but you can also schedule it to start at a certain moment in time. Let's try that now.

Scheduling a Task to Start at a Certain Moment in Time

In the following code, we do essentially the same thing that we did earlier, but this time we specify exactly when we want it to run. You do this by using a different constructor of the TimerTask class.

StartAtTimeTask.java

```
package net.javagarage.tasks;


import java.util.Calendar;

import java.util.Date;

import java.util.Timer;

import java.util.TimerTask;


/**

 * <p>

 * Starts a task at a particular date/time.

 * <p>

 * Note that if the scheduled execution time is

 * in the past, the task will start immediately.
 * @author eben hewitt
 */
public class StartAtTimeTask {

public static void main(String[] args) {
int delay = 2000; //2 seconds
int period = 5000; //5 seconds

//Get the Date corresponding to 3:58:00 pm today.
  Calendar calendar = Calendar.getInstance();
  calendar.set(Calendar.HOUR_OF_DAY, 15);
  calendar.set(Calendar.MINUTE, 58);
  calendar.set(Calendar.SECOND, 0);

Date scheduledDate = calendar.getTime();

System.out.println("Task scheduled for " +
           scheduledDate);

Timer timer = new Timer();
timer.schedule(new TimerTask() {
public void run(){
System.out.println("The task ran at " + new Date());

}
}, scheduledDate);

}
}
```

The output shows that the task ran at a different time than the task was scheduled for, because that time was in the past. Otherwise, it would have printed 15:58:00 as expected.

```
Task scheduled for Fri Jan 02 15:58:00 GMT-07:00 2004

The task ran at Fri Jan 02 15:59:08 GMT-07:00 2004
```

Note that the program will continue running after the job is scheduled, and will keep running. We don't call the cancel() method. That's because normally the reason you schedule a task is so that the scheduling thread can be alive to know when it is supposed to do its work.

Scheduling a Task that Runs Repeatedly at a Given Interval

As with my FTP application mentioned previously, you often need to check for a status or get an updated file or post some new data at a given interval, say every hour. The following code shows how to do this.

RepeatingTask.java

```
package net.javagarage.tasks;

import java.util.Timer;

import java.util.TimerTask;


/**

 * <p>

 * Demonstrate how to use a Timer that

 * causes some code to execute repeatedly

 * at a given interval.

 * @author eben hewitt

 */

public class RepeatingTask {


public static void main(String[] args) {


int delay = 2000; //delay for 2 seconds

int interval = 60000; //repeat every minute


Timer timer = new Timer();


timer.scheduleAtFixedRate(new TimerTask() {

public void run() {

// This is where you do the work

System.out.println("Current date and time is: " + new
java.util.Date());

} //end run method of our anonymous TimerTask

}, delay, interval); //end call to scheduler


}

}
```

When I run it, it outputs the following, and would keep outputting forever until the VM is shut down or something more colossal happens like the world blows up, which has the effect of shutting down the VM.

```
Current date and time is: Fri Jan 02 14:43:09 GMT-07:00 2004

Current date and time is: Fri Jan 02 14:44:09 GMT-07:00 2004

Current date and time is: Fri Jan 02 14:45:09 GMT-07:00 2004
```

Here are all of the overloaded Timer methods that are available:

```
schedule(TimerTask task, long delay, long period)

schedule(TimerTask task, Date firstTime, long delay)

schedule(TimerTask task, Date time)

schedule(TimerTask task, long delay, long period)

scheduleAtFixedRate(TimerTask task, long delay,

    long period)

scheduleAtFixedRate(TimerTask task, Date firstTime,

    long period)

cancel()

purge()
```

Stopping a Timer Thread

As we have seen, timer tasks will run forever if you don't shut down the thread. So there are a few different ways we can stop a timer task that's running.

1. Unplug the machine.
2. Call System.exit(0), which shuts down all of the threads in an app.
3. Create the timer as a daemon thread. Doing so will automatically stop the thread once there are only daemon threads left executing, and not user threads. See the "Creating a TimerTask as a Daemon" section later.
4. After all of the tasks have finished executing, remove all of the references to the Timer object. This is not a great way to do this if you need specific control over the cessation of this task, because when it is garbage collected, the thread terminates. But you aren't guaranteed of knowing when exactly that will be.
5. Call the Timer object's cancel() method. That method terminates the timer and throws away any tasks that are currently scheduled. After you cancel a timer in this manner, no more tasks may ever be scheduled on it.

Now that we see the different ways to do it, let's look at an example that offers us the most control, which is a good thing: calling the timer.cancel() method.

StoppableTask.java

```
package net.javagarage.tasks;


import java.util.Timer;

import java.util.TimerTask;

/**

 * <p>

 * Shows that we can extend TimerTask and just

 * override the run method there to do the work.

 * The benefit of this over doing it the other

 * way is that anonymous inner classes don't have

 * access to the non-final variables of the enclosing

 * class, so we can't call timer.cancel and such.

 * Here we can call timer.cancel(), which is what

 * we're trying to do.

 *

 * @author eben hewitt

 */

public class StoppableTask {

int delay = 0; //no delay

int interval = 2000; //run every two seconds
```

```
Timer timer;

//constructor
public StoppableTask() {

//make thread a daemon thread
timer = new Timer();

System.out.println("Starting task");

timer.schedule(new SuperTask(), delay, interval);
}

public static void main(String[] args) {

new StoppableTask();

}

// this inner class guy does whatever the
//task's work is
class SuperTask extends TimerTask {

private int counter = 1;

public void run() {

if (counter <= 3 ){
//This is where you do the work, just like Thread
System.out.println("This is me, the task, doing my
    work " + " time number " + counter + "!");
counter++;
} else {
System.out.println("All done.");
//put this thread to rest
timer.cancel();
}

}
} //end SuperTask class

} //end StoppableTask class
```

Here is the output if everything falls into place:

```
Starting task
This is me, the task, doing my work time number 1!
This is me, the task, doing my work time number 2!
This is me, the task, doing my work time number 3!
All done.
```

I suggest you buy as many blues albums as you can. Start with Robert Johnson, Skip Jones, and Muddy Waters, and work your way through Blind Willie McTell, JB Lenoir, Koko Taylor, Willie Dixon, and Keb Mo'.

Creating a TimerTask as a Daemon

This is easy and fun and it has a purpose. My favorite kind of thing. If you need to run a task, and then stop it after it has run, you need to stop it manually as we have seen. Except in one case: If you create the timer as a daemon, your task's thread will stop automagically when it realizes that there are only daemon threads left (and no user-invoked threads).

Here's how we do it:

```
new Timer(true);
```

I told you it was easy.

Why would you want to schedule a thread as a daemon thread? Well, think of a background process. If you are going to need this task to schedule repeated maintenance of the app, that you need to do throughout the application's life, this is a good choice. Think "garbage collector."

Creating a Timer for a Swing GUI Application

These timers are terrific, but they are meant for executing the kinds of tasks we mentioned—performing maintenance or doing some kind of transfer or other programmatic function. There is a different, related timer class for use in Swing applications. Don't start to panic because now there's even more about this when you didn't really care that much to begin with you just wanted a little timer and now you have to read all this junk but don't worry it will be okay. The functionality that is provided by the Swing timer and the java.util.Timer class are pretty much the same. There are a couple of differences.

- The Swing timer is useful if you are building a Swing app and don't need a bunch of timers. The util Timer class is scalable up to thousands of tasks, whereas you don't want to use the Swing timer for more than oh 25 or so.
- The main difference in terms of implementation is that Swing timers can schedule tasks to run on your GUI's event thread, because they all share that thread. Although you can subclass java.util.TimerTask and call the EventQueue.invokeLater() method to replicate this functionality, it is built in with the Swing timer.

Here's how you do it:

1. Create an object of type javax.swing.Timer.
2. Register an action listener on it.
3. Start the timer using the start() method.

The following code creates a timer and starts it up. After the elapsed delay, an action event is fired every delay (here, every two seconds). The second argument to the Timer constructor represents the guy who will do the work—an ActionListener.

```
int delay = 2000; // two seconds

 ActionListener worker = new ActionListener() {

       public void actionPerformed(ActionEvent evt) {

//do some task

       }

   };

 new Timer(delay, worker).start();
```

This event firing business will continue until someone puts it out of its misery by calling stop(). On the other hand, if you want it to execute only once, you can call the timer's setReapeats(false); method.

Chapter 29. APPLETS

DO OR DIE:

- Check out how to make applets
- Check out how to view applets
- Check out some gossip about applets

Back in 1995, everyone loved applets. Now, everyone hates them. Some computer science courses still start their Java courses with applets. Although that made sense 10 years ago, it doesn't make sense now, and, alas, the applet is relegated to little more than a glorified footnote here.

In this topic, we will look at how to write and use Java applets. Applets are small applications that run on the client. The early success of Java was due in large part to a successful demonstration of an applet running on a client machine via an applet container in a Web page.

There is some debate at this time regarding the usefulness of applets. They can be cumbersome to download, and the browser wars have made it difficult to deploy mission-critical applets and be sure that your users are seamlessly able to use them.

There are many very attractive alternatives to applets. Although these alternatives don't necessarily provide all of the functionality or all of the security that applets do, they often prove good enough for developers—and with many of these competing technologies offering easier development and deployment, applets have had a hard row to hoe of late.

Differences Between Extending JApplet and Applet

You can use either one. Because JApplet actually extends Applet, much of the functionality is identical. There are a few differences, however.

With JApplet, you have access to the content pane, which can be called using getContentPane(). If you have a content pane of your own (such as a panel) that you want to replace it with, you can call setContentPane(). When you add components to your applet, you add them to the content pane, not the frame.

FRIDGE

Dude. Make sure you write an HTML page that can display your applet. That is the purpose of applets—to be called from within HTML. Just because it is called appletviewer doesn't mean you pass it the name of the applet class. Pass it the name of the HTML file that references the applet class. I know I just said it, but boy does that happen a lot.

Also, you get the Metal look and feel by default. If you are on Windows, for example, and want to make your applet look like Windows, you need to manually switch to Native look and feel.

BorderLayout, not FlowLayout, is the default layout.

Lastly, you use the paintComponent() method (not paint()) to render your text and graphics.

Viewing Applets

There are two ways to view an applet you write: via Web page and with the appletviewer utility that ships with the JDK. Let's write a quick applet, view it in with the appletviewer tool, and then view it in a browser. Note that both ways of viewing an applet require an HTML page. The only way you can get out of writing the HTML page for testing purposes is with an IDE. For example, if you are using Eclipse (free from www.eclipse.org) to write your Java code. Write your applet class, and then click Run... > Java Applet. Works like a champ. Of course, you need to write the HTML once you go to deploy….

Writing an Applet

Let's just write an applet and try to view it and see what comes out of the woodwork, shall we?

SimpleApplet.java

```
package net.javagarage.applets;

import java.awt.Color;
import java.awt.Container;
import java.awt.FlowLayout;

import javax.swing.JApplet;
import javax.swing.JButton;
import javax.swing.JLabel;

/**
 * <p>
 * Example of very simple JApplet.
 * @author eben hewitt
 */
public class SimpleApplet extends JApplet {

public void init() {
Container content = getContentPane();
content.setBackground(Color.PINK);
content.setLayout(new FlowLayout());
content.add(new JLabel("Hello, world"));
content.add(new JButton("Poke me"));
   }

}
```

This applet doesn't do much at all. You see a label, you see a button, you press the button, nothin' happens. Put whatever code you want inside there. My purpose is to show you that you need to extend JApplet or Applet, and then write some stuff in the init() method body.

Now that the writing is out of the way, let's view that sucker. If you're using Eclipse or another IDE, you know what to do—cheat. If you want to rack up a little extra karma, you can write the HTML file now and view it with the appletviewer tool that ships with the JDK. Here's a file we can use.

ShowSimpleApplet.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>
<head>
<title>Applet Test</title>
</head>

<body>

<applet align="center"
    code="net.javagarage.applets.SimpleApplet.class"
    width="200" height="100">
</applet>


</body>
</html>
```

Notice a couple of things here. First, we use the .class file extension. Second, we use the fully qualified class name, which means including package names.

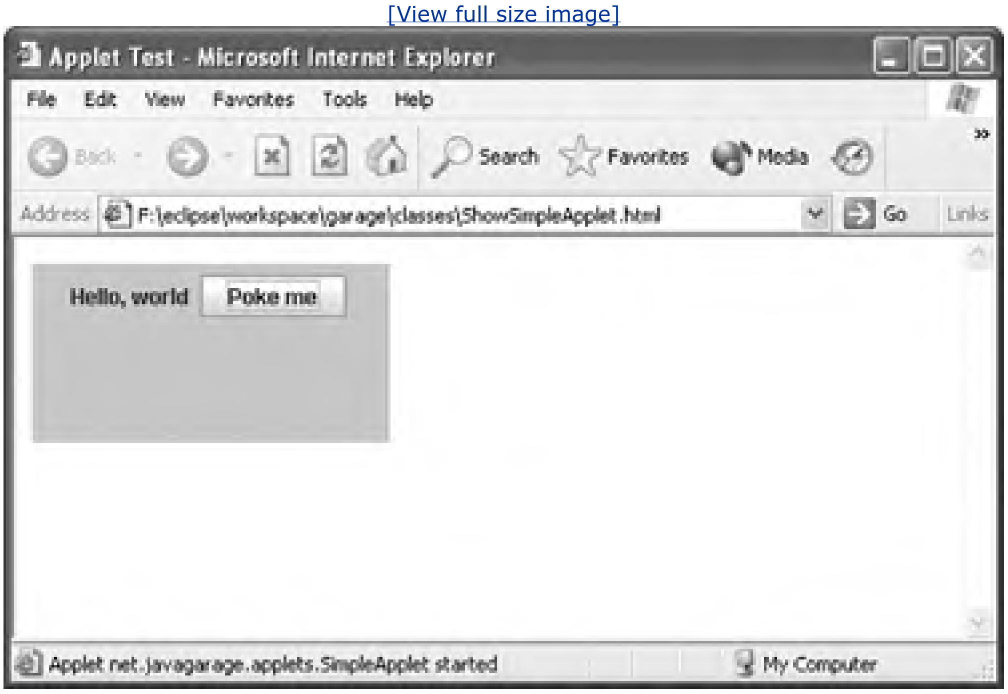
Now, you can just open your browser and point it to that HTML page to view the applet. Or, you can be fancy and use the JDK appletviewer tool at a terminal window, like this:

```
>appletviewer
F:\\eclipse\\workspace\\garage\\net\\javagarage\\applets\\Show
SimpleApplet.html
```

This is the full path to the file on my Windows XP machine. Note that you need not include the drive. Adjust the command for the location of your HTML file. This should launch the viewer program and display the applet, as shown in Figure 29.2.

Figure 29.2. Our little applet is all grown up and out in the world.

[View full size image]


Applet Test - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address F:\eclipse\workspace\garage\classes\ShowSimpleApplet.html
Go Links
Hello, world
Poke me
Applet net.javagarage.applets.SimpleApplet started
My Computer

If something went wrong and your HTML page just shows a gray box with a little red X in the upper-left corner, either there is a fatal error of some kind or (most likely) your applet cannot be found. You can check the path to the file and try again (make sure you included the fully qualified name of the applet class, including packages)—OR, you can right-click on the applet and select Open Java Console. This will show you the stack trace leading up to the problem. It is very likely that one of the following occurred:

1. You have specified the path to the applet incorrectly.
2. You have left off the .class extension from the name of the applet class in the HTML.
3. You have placed the HTML file in a place where it can't find the applet class: remember that the HTML should be at the root of your classes directory (the one that has "net" in it)—that's where java executes from.

Notice what I had to do here. If you look in the address bar, you'll see that the HTML page is referenced from within the classes directory. That's because that's where the class is. If you make a new file in your editor, it is likely to be sitting in the source directory—not the classes directory. Don't tell me that you have your source code in the same directory as your classes—that would be punishable by something pretty bad. A whole week without playing Road Blaster or Gauntlet, for example.

Applet Life Cycle Methods

There are several methods that allow you a little control over what happens during the life of an applet. Here they are in brief.

```
public void init() {

 // Called once by the browser when it starts the applet
}
public void start() {
  //called when the page that contains
     //the applet is made visible
}
public void stop() {
  //called when the page that contains the applet
     //is made invisible
}
public void destroy() {
 //called when the browser destroys the applet
}
public void paint(Graphics g) {
  //called when the applet is repainted
}
```

A Few Applet Tricks

Here are a few neat things that you can do within your applet, including some that might be downright necessary.

Showing a Message in the Browser's Status Bar

The status bar of the browser typically will display messages in the bottom-left corner. This feature is so retro mid-90s, it is impossible to resist.

```
applet.showStatus("All your base are belong to us. Ha ha ha");
```

Nuff said.

Making the Browser Visit a URL

You can redirect the browser using the showDocument() method, like this:

```
try {
      URL url = new URL(getDocumentBase(),
      "http://www.example.com/some.jsp");
      applet.getAppletContext().showDocument(url);
    } catch (MalformedURLException e) {
//deal
    }
```

Passing Parameters into Applets

You can pass a parameter into an applet from the HTML page that displays the applet. You do so using the applet.getParameter(String s) method. Then, you pass the parameter in via a special HTML tag called <param>.

Here is the code to get the parameter:

```
public void init() {
    String parameterName = "stockPrice";
    String s = applet.getParameter(parameterName);
}
```

This code will return null in the event that no parameter named "stockPrice" can be found.

Here is the HTML required to pass it in:

```
<applet code=MyApplet width=200 height=200>
   <param name=stockPrice value="$212.95">
</applet>
```

Note that if you are using a servlet container or some other app server like ColdFusion or ASP.NET or PHP, you can set the value of the parameter dynamically—that is, at runtime.

Loading Image Files in Applets

You can use image files inside your applets.

```
Image image;
public void init() {
   //get the image
   image = getImage(getDocumentBase(),
             "http://www.example.com/some.gif");
}
public void paint(Graphics g) {
   //draw image in upper-left corner
   g.drawImage(image, 0, 0, this);
}
```

Playing Audio Files in Applets

You can cause your applet to play audio files that are available via URL as well.

```
public void init() {

        // Load audio clip

        AudioClip ac = getAudioClip(getDocumentBase(),

"http://www.example.com/some.au");

//play the audio

        ac.play();


        //stop audio

        ac.stop();


        //play audio continuously

        ac.loop();

    }
```

With these easy methods, you can supply buttons to your applet that respond just like stereo controls.

We have admittedly skimmed over applets here (sort of). That's because they simply do not hold the attractiveness they once did. Server-side apps are more powerful in many ways. There is Java WebStart (which allows you to run Web-distributed programs locally with automatic updates), and applets have proven slow and difficult to manage. But they have a certain important place in Javaland, so they should be discussed. With what we've covered here, you certainly have enough to get started. The real appeal of applets of course is in the applications that you write and distribute as applets. Much of what we cover throughout the rest of this book could be created as applets knowing what we have gone over.

If you are eager to see some pretty sophisticated applets, try visiting www.java.com or http://java.sun.com/products/plugin/1.4.2/demos/plugin/applets.html. Sun has some very cool examples here. They probably will put a 5.0 page up, but it isn't there as of this writing, so check back for it.

PREV < Day Day Up > NEXT

Not-Applet Technologies

Some of these not-applet technologies include Flash rich user interfaces, Java WebStart, and of course Active X.

Rich user interfaces and client applications can be built now using Flash. The most recent version of Flash brings Action Scripting to the forefront, and features improved database access and XML processing over previous versions. Although Flash requires a browser plug-in too, and is arguably not easier to write than an applet is, Flash applications on the client are becoming more popular and useful, especially when integrated with the server-side Flash Gateway for Macromedia JRun or CFMX.

You can also execute ActiveX controls if you are in a Microsoft environment. These are commonly used to support such business as the display of Crystal Reports that are dynamically generated and printed to the browser.

A very interesting technology for executing full-blown programs on the client is called Java WebStart. It comes with Java 1.5 and is similar to applet technology, except that it doesn't require a browser to run in. You can write complete, full-blown Swing applications that can gain access to the local file system and do more than what is allowed in applets, and the apps will execute as if they are local. This technology bridges both worlds: the ease of maintenance and distribution that the Web affords, and the sophistication that desktop apps afford.

Applets did start the excitement surrounding Java, then moved out of the limelight as Java moved to the server. Although servlets had been available for a couple of years, they are tedious to write, and are poor choices for pages that need to write out to the browser for reasons we will see later. JavaServer Pages were introduced in 1999 and quickly garnered a lot of the interest previously reserved for applets. In an interesting twist, there is a significant trend by now for moving complex application functionality back onto the client. Java WebStart is one answer to that cry for client-side power. We'll have to wait for the *More Java Garage* book to get more of the low down on WebStart though.

Applet Basics

Applets are regular Java classes that extend Applet. Because of this, you can do just about anything that Java allows you to do from within an applet that runs in its own space on the client. There are, however, key restrictions. For example, you are not allowed to open a socket connection from inside your applet, except one back to the same server from which the applet came. Additionally, you cannot access the local file system. Beyond constraints such as these, applets are very powerful.

There are a few steps to getting an applet running on a client.

- Write a Java class that extends the java.applet.Applet class or javax.swing.JApplet class and imports the packages you need.
- Write an HTML page that calls the applet using the HTML <object> tag or <applet> tag.

Using the HTML <object> tag, you specify the size and location of the applet on the page, as well as the class file to be used. The browser then loads the plug-in, which contains its own Java Virtual Machine.

In the beginning, you could only view an applet by viewing the Web page in Sun's HotJava browser. This browser is currently in version 3.0 and is still available for free download. However, it is not a full-featured browser, it is slow, and it has little support for the kinds of developments in the language (such as CSS) that users have come to expect.

FRIDGE

You should already have the plug-in installed. It comes with the JRE. If for some reason you don't, you need to download it from www.java.com to execute the examples in this section. If you do have it installed, you can open the plug-in outside a browser and review the options it makes available to you. On Windows machines, you access it by double-clicking the icon in the Control Panel. You can also try opening Internet Explorer and opening the Tools menu. You should see "Sun Java Console" there.

Applets moved into the big time when they were adopted by Internet Explorer and Netscape browsers. Using a virtual machine embedded in the browser itself, applets could be embedded in a Web page using the HTML <applet> tag. As of HTML 4.0, the W3C has deprecated this tag in favor of <object>, which is more general. Note that older browsers may not recognize the <object> tag. Browser manufacturers have not updated their Java Virtual Machines for a version later than Java 1.1. This poses obvious difficulties to developers interested in deploying applets with functionality defined in later versions of Java.

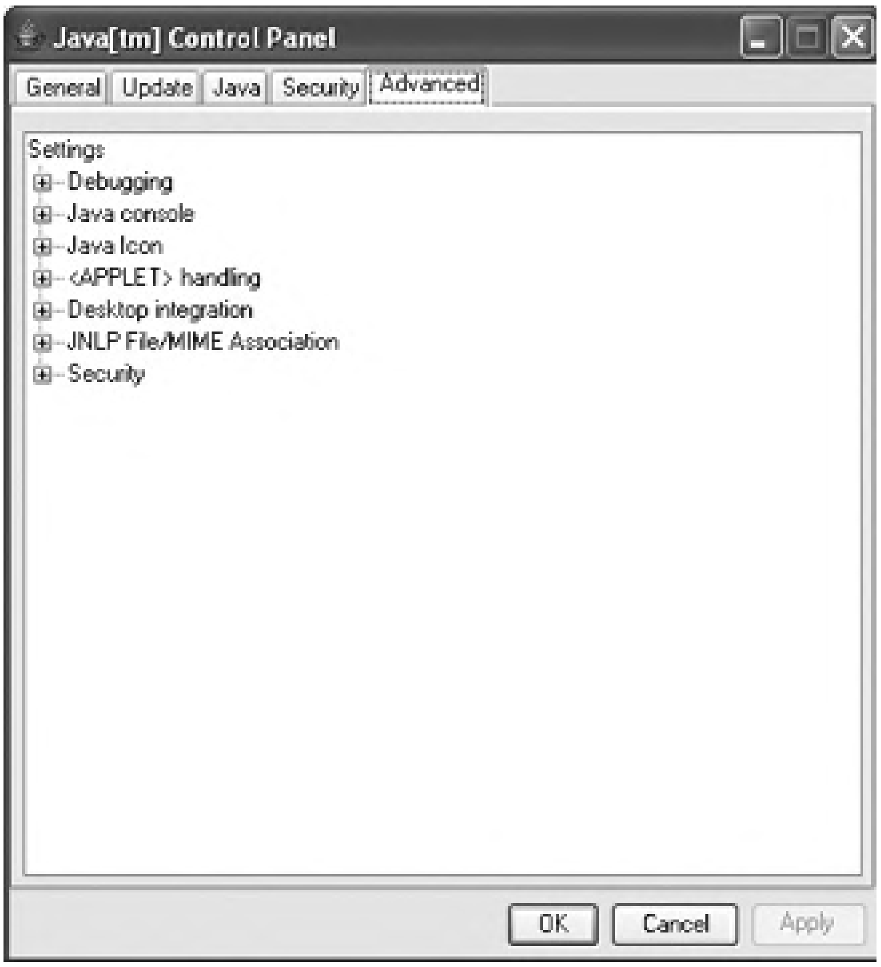
So, Sun devised a plan to propagate use of applets without being at the mercy of Netscape and Microsoft to keep up-to-date. This plan became the Java plug-in. You are likely familiar with this plug-in if you have used the <cfform> controls <cftree>, <cfgrid>, and so forth in ColdFusion 5 or later. This 5MB download allows both Internet Explorer and Netscape to execute applets using this runtime, which should always be up-to-date. Another benefit to the plug-in is added user control over the execution environment. The plug-in allows users to switch between different versions of Java Virtual Machines, for instance.

Eventually, Sun determined that limiting the plug-in to working with the newer <object> tag was dissuading developers from using applets. As of now, the fate of applets is admittedly up in the air.

If you run an applet, you should see the small Java coffee cup logo in your system tray. This indicates that an instance of the Java applet plug-in is running. You can right-click on this icon and choose Open Console to view information regarding your running applet. If the applet has errors, they will show up here, which is good to know for debugging.

The other option of interest in the Java plug-in is the Control Panel, as shown in Figure 29.1. Click Open Control Panel to bring it up.

Figure 29.1. The Java applet control panel offers a way to tinker with network settings and automatic updates.



Settings
- Debugging
- Java console
- Java Icon
- <APPLET> handling
- Desktop integration
- JNLP File/MIME Association
- Security

Here you can view your security settings and fine tune how applets are run in your browser.

There are basically two options for writing applets. You can extend java.applet.Applet. You can also choose to write applets using Swing components, which are sophisticated GUI elements located in the javax.swing package. To write applets using Swing components, you instead extend javax.swing.JApplet.

Chapter 30. FRIDGE: BIG DADDY FLAPJACKS

In an alarming change of plans, I'd like to have breakfast now. I'm not sure what kinds of questions inspire you. Maybe questions about the size of the universe, or how life is started, or the salary you'll get writing Java applications. Probably my favorite question is, "What's for breakfast?"

Let's just take a little break before going back to that hard application writing work. I'd like to have breakfast. If you would care to join me, we're having pancakes. And why not? I think everybody loves pancakes. Unless you are a vegan; then maybe you wouldn't like pancakes at all, and you will have to drink Diet Mountain Dew instead for your breakfast. Though I could be wrong about that.

Here is the recipe, which I really think you'll like. I'm telling you: try these. They're easy to make, and they really taste delicious. If you don't like to cook, maybe you can just chat with your sweetheart while you mix them up. That's what I do.

Ingredients:

1 egg

1 cup buttermilk

2 tablespoons vegetable oil

1 cup flour

1 tablespoon sugar

1 teaspoon baking powder

1/2 teaspoon baking soda

2 good pinches of salt

Grease your griddle. I just spray Pam on it for one second. Then start your griddle over a hot flame so it's ready when your batter is ready. In a mixing bowl, beat the egg. Then add remaining ingredients, stirring completely after adding each new ingredient.

Here's the secret to this recipe: let the batter sit for five minutes before you pour it. Then, when you're ready, pour batter onto grill using maybe a 1/4 cup measuring cup. Flip the flapjacks when they start to bubble on top.

This recipe makes eight good size Big Daddy Flapjacks. Enjoy immediately with real maple syrup and a hot cup of java. Now you'll be well nourished and ready for another exciting day of programming.

Chapter 31. USING SYSTEM AND RUNTIME

DO OR DIE:

- Great lamb chop ideas—just in time for the holidays!
- Take advantage of insider tips and tricks for making millions playing the market
- Investigate the Runtime class
- Get the static methods of the System class to do some work for a change
- See what the Java Runtime class can do

There is a class called System. It represents the underlying host for your application. You are doubtless familiar with it from your endless calls to System.out.println(). Well, friend, it does more. A lot more. And the Runtime class represents the Java Virtual Machine to your application. These two classes can be used in different ways to improve usability, fine tune your apps, improve performance, and make you the life of every party. Let's have a little look-see.

Using the System

The System class cannot be instantiated. But you can use its fields to get a hold of the standard out, standard in, and standard error IO streams. These streams represent how to print to the console, how to receive the input the user enters at the console, and the venue for printing error data.

The System class features a number of utility methods that you can use to perform miscellaneous tasks. Let's look at the methods, in order of sexiness.

Printing Data to the User

We have seen this a number of times already. I include it here for pure pedantry (and because there is actually some stuff that we might find useful). To print to the standard output stream, which is the console the user can view when executing a Java application, write thusly:

```
System.out.print("Some string");
```

The sister of the print method, println(), prints the text and as an added bonus appends a system-specific newline character. Now, you do not have to pass a String into the print() or println() method. These methods are both overloaded to accept a String, each of the primitive types, a char array, and an Object. This is where it gets interesting.

When an Object is passed to the method, its toString() method is automatically invoked. That means we can do this:

```
Object obj = new Object();

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

Here is the result:

```
java.lang.Object@119c082
```

The implementation of toString() in Object prints a String equal to this call:

```
getClass().getName() + '@' +

Integer.toHexString(hashCode())
```

It is recommended, however, that you always override toString() in your objects. Remember that every Object is ultimately a descendant of java.lang.Object, but they are the immediate descendants of all other superclasses. So if you don't override it, you'll get either the same useless information as shown in the preceding when someone—or someone you love—calls your Object's toString() method, explicitly or implicitly.

Determining Runtime Speed

The first method we'll look at is currentTimeMillis().

```
long System.currentTimeMillis()
```

This returns the current time in milliseconds, as follows:

```
Time: 1068434679594
```

This number, a primitive long, is the total number of milliseconds between midnight, January 1, 1970, and the current time. Although it may seem humorous in a department meeting to tell your boss that you'd be happy to meet him in 894,957 milliseconds to discuss your promotion, you might be the only one laughing. There is a very common use for such information: You can use it to figure out how long it is taking your code to run.

Say you've got a complex operation and you're concerned that it's taking too long. You can figure that out with some fancy tools that cost thousands of dollars, or you can trust old reliable: currentSystemMillis().

To help us flush out the bottlenecks in our application, we can get the time before and after chunks of suspect code, and subtract to determine total execution time. The code in SystemDemos.java shows this.

SystemDemos.java

```
public class SystemDemos {

public void testCurrentTimeMillis(){

    //some poorly written code that takes a while

    System.out.println("Bad Execution time: " + doBadCode());
    System.out.println("Improve Execution time: " +
         doImproveCode());
}

private long doBadCode(){
    long before = System.currentTimeMillis();
    for (int i = 0; i < 100000; i++){
        String s = new String("Object " + i);
    }
    long after = System.currentTimeMillis();
    return after - before;
}

private long doImproveCode(){
    long before = System.currentTimeMillis();
    String s = "";
    for (int i = 0; i < 50000; i++){
        s = "Object " + i;
    }
    long after = System.currentTimeMillis();
    return after - before;
}

public static void main(String[] args) {
    SystemDemos sys = new SystemDemos();
    sys.testCurrentTimeMillis();
}
}
```

Here is our result:

```
Bad Execution time: 200
Improve Execution time: 60
```

Note that this way of measuring execution time is not precise enough for applications that require serious performance. There is time taken to perform the method call that is not accounted for, and threading issues that are not accounted for, as well as random system events, such as spikes occasioned by other apps running. This method is good enough to get a rough outline. For more serious stuff, you can get a profiler to precisely measure performance.

In SystemDemos.java, we inspect two different methods: one that constructs strings in the naughty way (String s = new String("x"));, and another that constructs them the happy way. In each method, we create 100,000 different String objects and check how long it takes. There are only two differences between the methods. As you can see, redeclaring the String object in each of 100,000 iterations of the loop is an expensive proposition. It takes more than three times as long to achieve the exact same result. Those milliseconds could be precious in a complex application, and this is a simple, quick way to get a rough idea of whether a bit of code is going to end up a problem child.

Exiting an Application

You can announce your departure from an application using the exit(int status) method. This will shut down the currently running Java Virtual Machine by calling the exit method in the Runtime class, making the following two calls equivalent:

```
System.exit(0);

//or...

Runtime.getRuntime().exit(0);
```

By convention, an exit value of 0 indicates normal system exit, whereas any other number (commonly -1) indicates a deleterious state.

Forcing the Garbage Collector to Run

You can't.

Suggesting Kindly That the Garbage Collector Run

This you *can* do. The garbage collector runs in a separate thread from the thread of execution, reclaiming space here and there when it finds objects that can no longer be referenced, or other unreachable code.

The garbage collector has been around the block. She's seen objects come and go, every day throughout the years. And she's not about to take any lip from some upstart crow telling her how to manage her memory.

To suggest that the garbage collector run, which does not guarantee that it runs, try tempting her with this:

```
System.gc();
```

Note that, like other methods in the System class, this is the effective equivalent of calling

```
Runtime.getRuntime().gc();
```

This means that the Java Virtual Machine will have made its "best effort to reclaim space" (API documentation). But if the VM is busy, its best effort may not leave anyone thinking, "Whoa, that was really a good effort." In 99 cases out of 100, however, the call to gc() can be helpful, assuming that you need the memory back immediately, because the garbage collector will pick up method local variables very soon after the method returns.

Here is an example that demonstrates.

private void testGC(){

```
System.out.println("Free mem kb before bad code: " +

Runtime.getRuntime().freeMemory() / 1024);

for (int i = 0; i < 100000; i++){

String s = new String("Object " + i);

}

System.out.println("Free mem kb AFTER bad code: " +
Runtime.getRuntime().freeMemory() / 1024);

System.gc();

System.out.println("Free mem kb after GC call: " +

Runtime.getRuntime().freeMemory() / 1024);

}
```

And the result:

```
Free mem kb before bad code: 1864

Free mem kb AFTER bad code: 1493

Free mem kb after GC call: 1898
```

Notice that the garbage collector sees that all of these String objects are, as my friend John from Texas used to say, "a big show about nothing," and knows it can remove them. The call to the garbage collector freed up half a megabyte of memory. Running this program with the call to System.gc() commented out yields this result:

```
Free mem kb before bad code: 1864

Free mem kb AFTER bad code: 1493

Free mem kb after GC call: 1493
```

Executing an External Application in Java

There are many times that you might like to launch an application that is running on the user's machine. For example, an external Web browser could be used to display help pages. There is a way to do this that is as easy as running it yourself from the command line. All you have to know is its name on the system. To make your Java application open Windows Explorer, write

```
try {

Runtime.getRuntime().exec("explorer");

} catch(IOException ioe){

System.out.println(ioe.getMessage());

}
```

You have to catch or rethrow the IOException that this call can occasion. Note that we just pass in the system name, not including the file extension.

Passing Arguments to an External Application

Not only can we start an external application, we can pass arguments to it. Instead of allowing Windows Explorer to choose where to open, let's make it bend to our will by passing in the name of the directory in which we would like it to open: the Eclipse IDE workspace directory. We need to make two changes.

```
try {

String[] arg = {"F:\\eclipse\\workspace"};

Runtime.getRuntime().exec("explorer", arg);

} ...//catch statements
```

The first thing we've done is we've created an array of Strings with one value, which represents the arguments to our external app. Second, we called the overloaded exec() method that accepts a String array of arguments (sound like any method you're familiar with?), and passed that sucker in.

Java will run the application if it can. Here, we start Windows Explorer, which will open in the Eclipse workspace directory.

This functionality is sometimes employed to start a browser and display a Web page in it. Note that you don't need to know the path to the program you want to launch. If you can type it in a terminal and make it run, it will work here. This functionality is very similar to doing this in Windows: Click Start > Run. Type cmd to get a prompt. Enter explorer F:\eclipse\workspace (or whatever path you want to open). You're good to go.

Interacting with the User

There are a few different ways to interact with the user from within an application. For desktop software, you use classes in the Swing packages javax.swing. There are servlets and JavaServer Pages that allow you to interact from a Web server. However, there are many occasions on which you just want something simpler. It can be difficult to set up Swing applications with sophisticated layouts, events, and handlers. There are many times too when a console application is not only fine, but is the best choice for interacting with the user. In these cases, you turn to the System class.

Reading User Input from the Console with System.in

When you want to print something out to the console, you use the print() or println() methods of the System.out object. To read user input from the console, get the static System.in object, which is of type InputStream. As with the OutputStream provided by the out object, this stream opens automatically when your program starts. So all you have to do is read from the stream. Sounds straightforward enough, right?

Well, not really.

We know that dealing with IO is often a little tricky, just because there are so many ways to do it. What type of field is InputStream? It is abstract, representing the superclass of all classes that accept data as byte arrays. That means that we need to wrap it in a class that defines a method for accepting bytes of input as characters. However, we're talking about a fairly costly operation: using a byte array reader to read data, then converting it to characters with an InputStream reader, returning the converted text constantly. What we need is a way to do all of that, but make a little temporary storage area, a buffer, for the character data. The most efficient way to do that is to wrap our InputStreamReader with a BufferedReader. So the call that gets data from System.in looks like this:

```
BufferedReader in

  = new BufferedReader(new InputStreamReader(System.in))
```

Now we have a reference that we can use to read each line of user input. To do that, we use a loop that checks for the end of the input, like this:

```
while((line = in.readLine()) != null) {

//strip spaces out just to show an operation happening

line = line.replaceAll(" ", "");

}
```

If you are going to do this sort of thing (use runtime.exec()),you really should read the next section after the Calculator code, called "Thread Issues with Runtime.exec()."

Note that if you want to compile the preceding code in a class, you will need to catch the exceptions thrown by the reader classes.

Toolkit: A Simple Calculator

This program demonstrates how you might write a simple calculator in Java. It incorporates a number of the programming constructs we've covered so far in this book, and it could be usefully incorporated into other applications. The main thing it does that is of interest here is that it uses the standard input and shows how to read user-entered data in a complete, working app.

It demonstrates using File I/O (BufferedReader and InputStreamReader), using regular expressions, and exception handling.

```
package net.javagarage.apps.calculator;


import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.util.regex.Matcher;

import java.util.regex.Pattern;


/**<p>

 * Demos a working calculator, usable from the

 * console. The advantage is that it shows, in

 * a small program, how to fit different pieces

 * together.

 * <p>

 * More specifically, it shows how to read in

 * data the user types on the console, and how

 * to parse strings in a real-world way.

 * <p>

 * Limitations include the fact that a legal

 * expression is constituted by two operands

 * and one operator; i.e., we can't do this:

 * 4 * (5 + 6)

 * </p>

 * @author eben hewitt

 * @see BufferedReader, Float, String, Pattern

 **/

public class Calculator {
```

```
private String readInput() {
    String line = "";

    try {
    //get the input
    BufferedReader in = new BufferedReader(
            new InputStreamReader(System.in));

    while((line = in.readLine()) != null) {
        //strip spaces out and do calculation
        line = "result: " +
                calculate(line.replaceAll(" ", ""));
    }
    } catch (IOException e) {
        System.err.println(e.getMessage());
    } catch (NumberFormatException nfe){
        System.out.println("Error->" +
                nfe.getMessage());
        System.out.println("Enter an expression: ");
        //recursive call. Otherwise the program stops.
        readInput();
    }
    return line;
}

private float calculate(String line){
    //these will hold the two operands
    float first, second;
    String op = "";
    int opAt = 0;
    //build regex to parse the input
    //because * is an operator in regex, we
    //need to escape it.

    Pattern p = Pattern.compile("[+/-[*]]");
    Matcher m = p.matcher(line);
    //when you find an arithmetic operator in the
    //line, get its position
    if (m.find()){
        opAt = m.start();
    } else {
        //user didn't enter a valid expression
        System.out.println("usage-> Enter a valid
                   expression");
        return Float.NaN;
    }
    //get the part of the string before the operator
    //that is our first operand
    first = Float.parseFloat(line.substring(0, opAt));
    //find the occurrence of one of the operator
    //symbols
    op = line.substring(opAt, opAt+1);
    //get the number between the operator and the end
    //of the string; that's our other operand
    second = Float.parseFloat(line.substring(opAt+1,
    line.length()));
    if (op.equals("*")){
        System.out.println(first * second);
    }
    if (op.equals("+")){
        System.out.println(first + second);
    }
    if (op.equals("-")){
        System.out.println(first - second);
    }
    if (op.equals("/")){
        System.out.println(first / second);
    }

    return Float.NaN;
}

/**
 * You can hit CTRL + C to cancel the program
 * and start over.
 * @param args
 */
public static void main(String[] args) {
    System.out.println("Enter an expression: ");
    new Calculator().readInput();
}
}
```

Result

Here is what a sample execution of the program looks like:

```
Enter an expression:
655+99807
100462.0
65 * 876
56940.0
510 / 70
7.285714
34 - 9876
-9842.0
fake input
usage-> Enter a valid expression
```

The limitations of this program are that it doesn't handle nested expressions and it doesn't have a control for the user to stop the program; it will just run until you manually shut it down.

Thread Issues with Runtime.exec()

Stuff doesn't always come out as we plan. Sometimes things go wrong in our code. But sometimes, we just haven't done as much as we could have to plan for contingencies.

The previous examples of using the System.in stream work fine. If you need to access System.err during execution of your program, though, you have two streams coming in, and only one place to read them. You have tried to be a good programmer, understanding that sometimes bad things happen to good programs, and handle them. But this presents a special kind of difficulty. Here is the common problem.

Your program tries to read from the standard input, and then tries to read from the standard error stream. This doesn't seem like what you want, but how else are you going to do it, as your `while` loop processes? You can't read from both standard in and standard error at the same time, right? (You can, and should). Because your process could instantly try to write to standard error (if something goes wrong in your program before anything is written to standard in). If that happens, your program could block the on `in.readLine()`, waiting for the process to write its data. However, back at the ranch, the program is trying to write to standard error. It might be expecting you to read standard err and free up the buffer. But you don't do this because you are waiting for the process to write to standard error. You can see where this is headed. Deadlock.

This is when threads are really called for. Look at your program and honestly answer this question: "How many things am I trying to do here?" If the answer is more than one, ask yourself this: "Do any of these tasks require waiting for event?" If the answer is yes, that guy is a good candidate for a thread. Doing things in one thread can be more complex obviously, but often prevents your application from crashing after many pending and unresolved issues are piled up higher and higher without chance at resolution. The best-case scenario here is usually very poor application performance.

To achieve this smoother functionality, we need to do a little work of our own. Specifically, we need two classes: one to represent threads for standard input and standard error, and another to be the app that places the call to runtime.exec(). First, the mama app.

MultithreadedSystem

```
package net.javagarage.sys;


/**

 * <p>

 * Demonstrates use of threads to handle standard

 * error stream interruptions of input in order

 * to avoid deadlock.

 *

 * @see Thread, ThreadedStreamReader

 * @author eben hewitt

 */

public class MultithreadedSystem {


public static void main(String[] args) {


MultithreadedSystem m = new MultithreadedSystem();


try {

//m.doWork(new String[]{"explorer", "dude"});


//try using a program that uses the stdin, like this,

//remember that each command must be separate

m.doWork(new String[] {"cmd", "/c", "start", "java",

"net.javagarage.apps.calculator.Calculator"});

} catch (Exception e){
```

```
e.printStackTrace();

}


System.out.println("All done");

}


/**

 * This is where you would do whatever work with the

 * standard input that you want.

 * @param String[] command The name of the program

 * that you want to execute, including arguments to

 * the program.

 * @throws Exception

 */

public void doWork(String[] command) throws Exception {


//this will hold the number returned by the spawned app

int status;


//use buffers to stand in for error and output streams

StringBuffer err = new StringBuffer();

StringBuffer out = new StringBuffer();


//start the external program

Process process = Runtime.getRuntime().exec(command);


//create thread that reads the input stream for this process

ThreadedStreamReader stdOutThread =

new ThreadedStreamReader(process.getInputStream(), out);


//create thread that reads this process'

//standard error stream

ThreadedStreamReader stdErrThread =

new ThreadedStreamReader(process.getErrorStream(), err);


//start the threads

stdOutThread.start();

stdErrThread.start();
```

```
//this method causes the current thread to wait until

//the process object (the running external app)

//is terminated.

status = process.waitFor();


//read anything still in buffers.

//join() waits for the threads to die.

stdOutThread.join();

stdErrThread.join();


//everything is okay

if (status == 0) {

System.out.println(command[0] + " ran without errors.");


//you can print the values of the out

//and err here if you want

//if the result is not 0, the external app

//threw an error

//note that this is by convention only, though most

//programs will follow it

} else {

System.out.println(command[0] + " returned an error status: "

+ status);

//you can print the values of the out and

//err here if you want

}

}

}
```

The second class will extend thread, and we'll direct the output using it.

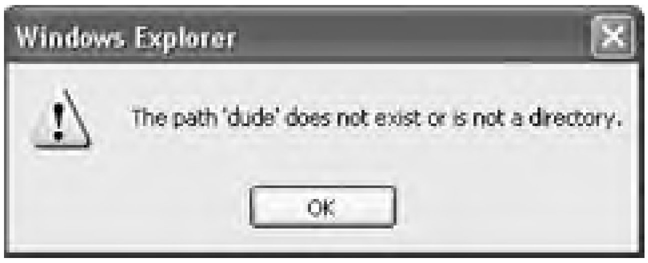
ThreadedStreamReader

```
package net.javagarage.sys;


import java.io.InputStreamReader;

import java.io.InputStream;


/**

 * <p>

 * File: ThreadedStreamReader

 * Purpose: To ensure access to standard error

 * System.err when reading a buffered stream with

 * System.in.

 * <p>

 * Note that typically in your programs you want to

 * implement the Runnable interface to do

 * multithreading, but here we are only ever going to

 * use this class for this thread-specific purpose,

 * so it is okay. It is preferable to implement

 * Runnable in general, because then you are free to

 * extend a different class (and Java does not allow

 * multiple inheritance).

 *

 * @author eben hewitt

 */

public class ThreadedStreamReader extends Thread {


private StringBuffer outBuffer;

private InputStreamReader inputStream;


//constructor accepts an input *(

//for instance, and

/**

 * @param InputStream either standard err or standard in

 * @param StringBuffer

 */

public ThreadedStreamReader(InputStream in, StringBuffer out){
```

```
outBuffer = out;

inputStream = new InputStreamReader(in);

}


//override run() to do the thread's work

public void run() {

int data;

try {

//hold character data in the buffer until we get to the end

while((data = inputStream.read()) != -1)

outBuffer.append((char)data);


} catch (Exception e) {

//tack the error message onto the end of the data stream

outBuffer.append("\nError reading data:" + e.getMessage());

}

}

}
```

Executing this application with arguments of explorer and dude gets us the error shown in Figure 31.1.

Figure 31.1. Windows Explorer starts up but cannot find this directory and returns an error status, which we capture.



The path 'dude' does not exist or is not a directory.

Our application then prints the following and exits:

```
Process explorer returned an error status: 1

All done
```

Of course, executing that program doesn't give us access to these streams, so it isn't entirely useful, but it proves in the simplest possible manner that our program does work. Which is not a bad practice.

Now, if you have the Calculator program compiled, you should be able to call it just as I do here. If it won't run, perhaps because you get an IOException: Cannot create process, error=2 or something like that, make sure that your path is set up correctly. You can check if your system can execute the Java command by opening a command prompt and typing java. If usage information prints out, you're in business. So go ahead and run it.

If everything goes as planned, our call to the Java calculator program should look like this after we're finished using it.

```
cmd ran without errors.

All done
```

I love it when a plan comes together.

Now let's look at another useful example of getting the runtime to execute something.

Toolkit: Getting the MAC Address from Your NIC

To show that there are useful things that are made possible with this business,, and to do something that's possibly useful, let's write a program that works on both Linux and Windows and calls programs that ship with each of those operating systems. Let's get the MAC address off of our network cards.

MACAddress.java

```
package net.javagarage.misc;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;

/**<p>
 * Uses system tools in windows or linux to
 * get the user's Media Access Control addresses
 * and print them out.
 * <p>
 * Demonstrates practical use of runtime.exec().
 * On Linux, you must be root to run it.
 * <p>
 * One shortcoming of this program is that if you
 * encounter any errors in the process, they aren't

 * registered.

 **/

public class MACAddress {

//if true, the messages passed into the log() method

//will print; if false, they are ignored

static boolean debug = true;


/**

 * Determine what OS we are on, and call a different

 * program to do our work depending on the result.

 * @return

 */

public static ArrayList getMACAddresses(){

//what regex matches the String representation

//of the address returned?

String matcher = "";


//what program is used on this OS to get the MAC address?

String program = "";


//how many places does this address String require

//us to move back?

int shiftPos = 14;


//different operating systems have different programs

//that get info about the network hardware

String OS = System.getProperty("os.name");


if (OS.startsWith("Windows")){

/* ipconfig returns something like this:

 * 00-08-74-F5-20-CC

 */

matcher = ".*-..-..-..-..-.*";

program = "ipconfig.exe /all";
```

```
}
if (OS.startsWith("Linux")){
/*ifconfig returns something like this:
"HWaddr 00:50:FC:8F:36:1A"
the lastIndexOf method will set the substring
to start 14 spaces back from the last :, which is
the beginning of the address
*/
shiftPos = -39;
matcher = ".*:..:..:..:..:.*";
program = "/sbin/ifconfig";
}
//sorry no macintosh
return getMACAddresses(matcher, program, shiftPos);
}

/**
 * Overloaded method uses the program that
 * ships with the current OS and parses the string
 * that it returns.
 * @return String The address of each
 * device found on the system.
 */
private static ArrayList getMACAddresses(String
matcher,
String program, int shiftPos){
String line = "";
Process proc;
ArrayList macAddresses = new ArrayList(2);
try {
//run the windows program that prints the MAC address
proc = Runtime.getRuntime().exec(program);
BufferedReader in = new BufferedReader(
new InputStreamReader(proc.getInputStream()));
while((line = in.readLine()) != null) {
//the matches method determines if a given
//String matches the passed regular expression
if (line.matches(matcher)) {
int pos = line.lastIndexOf("-")-shiftPos;
//add the address to the list
macAddresses.add(line.substring(pos,
line.length()));
}
}
} catch (IOException e) {
log("Error using ipconfig: " + e.getMessage());
proc = null;
System.exit(-1);
}
return macAddresses;
}

/**
 * Print out the info we parsed. These are separated
 * methods because you might want to do something
 * other than just print it out.
 * @param devices
 */
public static void printAddresses(ArrayList devices){
int numberDevices = devices.size();
System.out.println("Devices found: " + numberDevices);
for(int i=0; i < numberDevices; i++) {
System.out.println("Device " + i + ": " +
devices.get(i));
}
}

//convenience method for printing messages to stdout
private static void log(String msg){
if (debug){
System.out.println("—>" + msg);
}
}
//start program
public static void main(String[] args) {
printAddresses(getMACAddresses());
```

```
System.exit(0);

}


} //eof
```

The result is shown here:

```
Devices found: 2

Device 0: 00-20-A6-4C-93-40

Device 1: 00-04-76-4D-D6-B5
```

My laptop has one standard Ethernet adapter (device 1) and one wireless 802.11 card (device 0). Note that you can toggle the debug boolean to print out information regarding the current state of the program.

Determine Number of Processors on the Current Machine

This ability could come in handy if, say, you had some software product you were licensing per processor. You could use this method to help ensure user compliance:

```
Runtime.getRuntime().availableProcessors();

//returns an int.

//In my case, 1.
```

Read *1984* again.

The war is not meant to be won...

Determine When Your Application Is About to Exit

What do you do when you want to shut down your Windows box? You click on Start. Likewise, when your application is shutting down, it doesn't stop at all. It rather starts something new. The application starts any registered shutdown threads. Only when they complete their work will the application exit. This is the case for normal termination—for instance, when you call this code:

```
System.exit(0);
```

Passing the 0 int to the exit method means, by convention, that everything is hunky dory in your application, and that you just want to end it. Typically, a non-zero number (such as -1), as you might imagine, indicates a non-normal termination.

Abnormal termination is typically unexpected, but there are times when the chances are greater that something is going to run amok in your app. Like when you pass control over to something else, such as a native library or a network resource. The user can shut down the JVM abruptly by typing CTRL + C.

If there is some work that you would like to do but the application is about to shut down, you can register a shutdown thread, known as a hook, to do your dirty work. Maybe this dirty work is like, "write a message out to a log, log the user out, make sure he isn't working with any shared document keys, or send an e-mail." Some would argue that these things are perhaps too much work to do in a shutdown hook.

For example, we could write an entry to a log file when a user starts an application, and then call the shutdown hook to write the time when he shuts it down. This could give us possibly useful metrics on the order of, "How long did Sally do any work before goofing off on the Internet for 4.5 hours?" Here's how you do it.

ShutdownHook.java

```
package net.javagarage.sys;

/**

 * <p>

 * Demos the usefulness of the

 * Runtime.getRuntime().addShutdownHook()

 * method.

 * @author eben hewitt

 */

public class ShutdownHook {


private void init(){

//register a new shutdown thread

   Runtime.getRuntime().addShutdownHook(new Thread() {
```

```
  //
  public void run() {
  //do work here
System.out.println("In shutdown hook");
  }
  });
}

private void doSomething(){
System.out.println("App started...");
}

public static void main(String[] args) {
ShutdownHook hook = new ShutdownHook();
hook.init();
hook.doSomething();
try {
System.out.println("Sleeping...");
Thread.sleep(5000);
} catch (InterruptedException ie){
System.out.println(ie.getMessage());
}
}
}
```

Here are a few final considerations regarding shutdown hooks:

- It is not guaranteed that the application will run your shutdown hooks. If the user unplugs the machine, for instance, they won't run. Also, if the process is terminated externally—via a kill process command, for instance—your hooks are not guaranteed to run.
- Do little work in these hooks. The user will expect that your app will shut down cleanly and quickly when he requests it; it is good not to test his patience.
- Code carefully in this method. This is obviously not a time to play fast and loose with things. The app is shutting down, and the JVM is shutting down; make sure your code here is thread safe.

That's enough about that. I'm going to watch some of the women's college volleyball championships. Oregon State is just demolishing Cal.

System Properties

There are certain things that the JVM always knows about, that you might like to know about, too. These things consist of information about the user's working environment. That's why they are called system properties.

Commonly Used System Properties

Below are listed some of the more commonly referenced system properties. For example, your application might want to perform some specific behavior if the underlying platform is Windows, or you might want to open a JFileChooser dialog that allows the user to save a file, and have it present the user's home directory as its starting location.

user.dir	User's current working directory
user.home	User's home directory
user.name	Account name for currently logged in user
file.separator	File separator (for example, "/")
line.separator	Character representing a line separator on this system
path.separator	Path separator (for example, ":" on Linux, ";" on Windows)
os.arch	Operating system architecture
os.name	Operating system name
os.version	Operating system version
java.class.path	Java classpath
java.class.version	Java class version number
java.home	Directory where Java is installed
java.vendor	String indicating the vendor of this JVM
java.vendor.url	Java vendor URL
java.version	Java version number

Java provides an easy and convenient way to access these properties and to set them, which we'll see how to do now.

PropertiesDemo.java

```
package net.javagarage.demo.properties;


import java.util.Enumeration;

import java.util.Properties;


/** <p>

 * Shows how to get all of the system properties.

 * Note that your methods should really do one thing,
 * not multiple things. Here we violate that for demo
 * purposes; that is, we both get the property AND
 * print it in the same method. Typically, you don't
 * do this.
 *
 * @author eben hewitt
 **/
 public class PropertiesDemo {

 public static void printSystemProperties(){
 //get the system properties
 Properties props = System.getProperties();

 //returns an enumeration
 Enumeration propNames = props.propertyNames();
 while(propNames.hasMoreElements()) {
 //get current property
 //cast to String type, since return type is Object
 String propName = (String)propNames.nextElement();

 //get the value of current property
 String propValue = (String)props.get(propName);
 System.out.println(propName + " : " + propValue);
 }
}

public static void main(String[] args) {

//print only the value of the name specified
System.out.println("Operating System: " +
System.getProperty("os.name"));

//probably not a smart idea (changing system
//properties with brute force), but...
System.setProperty("os.name", "Solaris 8");
System.out.println(System.getProperty("os.name"));

//you can make a new property too, which is cool
System.setProperty("javagarage.coolnesslevel", "supercool");

System.out.println(

System.getProperty("javagarage.coolnesslevel"));


//print all of them

printSystemProperties();

}

}
```

There are a few things going on here. The first is that we can reference system properties by name and get a String value back. The second is that we cannot only get the values of pre-defined properties, but we can create arbitrary new ones, and set them to values of our choosing. We can also overwrite the values of system properties, such as we do with the os.name property. (Not brilliant programming.) Last, we see that we can get an enumeration of all of the properties, and loop over it to print out each value.

There are a couple of things to keep in mind about these properties. The first is access. A desktop application has full access to all of these properties. An applet, which is typically deployed via a browser, does not. There is a subset of the properties that are available, but for security reasons, many are omitted. When your applet tries to reference a property it is not allowed to get at, an exception is thrown indicating the attempted security breach.

The second note is more about good programming practice. You can store values in properties that your application uses, such as preferences for fonts or a welcome message, or other user setting preferences. And although you will see code floating around that uses properties in this way, it is no longer a good idea. A few years ago, this was not only an acceptable use of properties, but one of their intended uses. Which is why you can write to them (duh). However, in recent versions of the Java language, more functional APIs have emerged that address this need in more specific and robust ways. These include the Preferences API, which is used to good effect in the Toolkit Garage Pad example, and the XML APIs, which allow you to store data in a hierarchical format. See, the System Properties stuff is all flat, all on the same plane. It can't be organized. Although the Preferences API requires a little more seductive talk to get it going, it allows you to do stuff like attach properties to individual classes, which is well worth the little bit of extra programming investment.

Also, if you do choose to put a value into the Properties Hashtable, you should be aware that it does not check if the value you add is actually a String, even though that is all they are supposed to store. So use them judiciously.

FRIDGE

Note that you can also use these guys for debugging. In particular, notice the java.class.path and java.library.path properties—you might find that your application requires some functionality provided by a JAR file that is not actually on your classpath, or that it is the incorrect version, or like that. Also, classes in the java.io package will interpret any relative path names as starting with the value of System.getProperty("user.dir"); so it might help you find that file that doesn't seem to be where you thought it would be.

Chapter 32. USING THE JAVA DEVELOPMENT TOOLS

The Java SDK contains a number of tools in addition to the commonly used java interpreter and javac compiler. This topic gives you an overview of all of the tools in the SDK, and details the most commonly used of those.

Using Common SDK Tools

appletviewer

The Java Applet Viewer command allows you to execute applets outside of the context of a Web browser. To view an applet, pass the URL of the HTML document with the embedded applet to the appletviewer program as an argument. If the document does not contain any applets, the program does nothing. The program will run each applet it finds in a separate window.

appletviewer recognizes applets embedded using the `<object>`, `<embed>`, and `<applet>` tags.

The following options are available:

- `-debug` Starts the appletviewer using the Java debug program jdb, allowing you to debug applets.
- `-encoding encodingName` Specifies the input HTML file encoding that should be used when reading the URL.
- `-Jjavaoption` Passes the specified `javaoption` as an argument to the Java interpreter.

Usage: `appletviewer [options] url file`

jar

The JAR tool combines multiple files into a single compressed file called a Java ARchive. It uses the same compression algorithm as Zip files, which means that you can open JAR files using any standard Zip utility. Its primary purpose is to facilitate ease of distribution when managing complete applications. For example, you can bundle your entire application into a single JAR for distribution, and others can use it by simply dropping it into their classpath. Alternatively, a JAR file can be set to be self-executing, so that the freestanding application it contains can be run by double-clicking. Applets have made good use of JAR files, because they facilitate all of the classes in an application being downloaded at one time, which saves considerable HTTP traffic, and speeds execution. In addition, JAR files can be signed so that users can determine the application's author.

If you use UNIX, you will find it easy to begin working with the `jar` command, as its syntax is nearly identical to TAR.

Options

The following options are available for the `jar` command:

- `c` Creates a new archive in a file named `f`, if `f` is also specified.
- `u` Updates an existing JAR file when `f` is specified.

  For example,

  jar uf myApp.jar myClass.class updates the JAR file named myApp.jar by adding to it the class named myClass.class.
- `x` Extracts the files and directories from JAR file `f` if `f` is specified, or the standard input if `f` is omitted. If input files are specified, only those files will be extracted; otherwise, all of the files in the directory will be extracted.
- `t` Lists the table of contents in JAR file `f` if `f` is specified, or the standard input if `f` is omitted. If input files are specified, only those files and directories will be listed; otherwise, all of the files and directories will be listed.
- `i` Generates index information for the specified JAR file and its dependents.

  For example,

```
jar i myApp.jar
```

generates an index called INDEX.LIST in the specified JAR file.

- f Specifies the name of the JAR file on which you want to perform an operation, such as create or update. The option f and the name of the JAR file must both be present if either one of them is present. Omitting f and JAR file accepts a JAR file from standard input (when you specify the x and t commands) or sends the JAR file to standard output (when using the c and u commands).
- v Generates verbose output. That is, it tells you a lot about what it is doing as it does it.
- 0 That's a zero, not the capital letter o. It tells the JAR application not to use compression, but to simply store the files.
- M If you specify the c or u commands, M prevents the creation of a Manifest file.
- m Includes name : value pairs that are in the Manifest file located in META-INF/MANIFEST. MF. A name: value pair is added unless one already exists with the same name, in which case its value is updated. The m and f options must appear in the same order that Manifest and JAR file appear when typing the command.
- -C dir Changes to the directory specified in dir temporarily during execution of the jar command while processing the following filesToAdd argument.

  ```
  jar uf myApp.jar -C classes . -C bin myClass.class
  ```

  The preceding command updates myApp.jar by adding to it all of the files of any type in the classes directory and does not create a directory named classes in the jar. It then temporarily changes to the original directory (.) before changing to the bin directory to add the single class myClass.class to the JAR file.

- -Joption Passes option to the underlying Java runtime, so possible values for option here are the same as the options available to be passed to the java launcher itself.

Typical Examples

The following is how you frequently will use the jar tool:

```
% jar cf myApp.jar *.class
```

This example takes all of the files in the current directory with a .class extension and bundles them into a resulting JAR file called myApp.jar. A Manifest entry is automatically created. A Manifest file stores meta-information regarding an application in the form of name : value pairs.

For executing a JAR file, see the next section.

java

The java tool launches Java applications. It starts a Java Runtime Environment, loads a specified class, and invokes that class' main method. You should use the class' fully qualified name. You may pass arguments to the application; any arguments that are not options passed.

Typical Examples

The following is how you frequently will use the java tool.

```
% java -cp Main.class
```

The java tool will look for the startup class and other necessary classes in three locations: the bootstrap classpath, the installed extensions, and the user classpath.

- -client Selects the Java HotSpot Client Virtual Machine.
- -server Selects the Java HotSpot Server Virtual Machine.
- -classpath The user classpath.
- -cp. Alternative method of specifying the -classpath option. Specifies a list of directories, JAR archives, and Zip archives to search for class files. Class path entries are separated by colons (:) on UNIX, Solaris and Linux, and a semi-colon(;) in Windows. Specifying -classpath or -cp overrides any setting of the CLASSPATH environment variable that might already exist. If neither -classpath nor -cp is used and CLASSPATH is not set, the user class path consists of the current directory (.).
- -Dproperty=value Sets a system property value.
- -d32 -d64 Specifies whether the program is to be run in a 32-bit or 64-bit environment. If neither -d32 nor -d64 is specified, the default is to run in a 32-bit environment, though this will likely change once 64-bit processors become more prevalent.
- -enableassertions[:<package name>"..." | :<class name>] -ea[:<package name>"..." | :<class name>] Enables assertions.
- -disableassertions[:<package name>"..." | :<class name>] -da[:<package name>"..." | :<class name>] Disables assertions (clever…).
- -enablesystemassertions –esa Enables assertions in all system classes.
- -disablesystemassertions –dsa Disables assertions in all system classes.
- -jar Executes an application encapsulated in a JAR file. The first argument to the call is the name of the JAR file containing the application. The manifest of the JAR file must contain a line like this: Main-Class: classnameContainingMainMethod. Using this option, the JAR file becomes the source of all user classes, and the runtime ignores other user class path settings.
- -verbose -verbose:class Displays verbose information regarding each loaded class.
- -verbose:gc Displays information regarding garbage collection events.
- -verbose:jni Displays information about Java Native Interface method execution.
- -version Displays information regarding version information. After the version info is displayed, the program exits.
- -showversion Displays information regarding version information. After the version info is displayed, the program

continues.

- -? –help Displays the usage information in this section and exits.
- -X Displays information regarding non-standard options and then exits. Non-standard options are not presented here.

Here is how to execute a program called "runme.jar" that has been bundled as a JAR:

```
java –jar runme.jar SomeArgument
```

It is assumed that you have created a manifest file for this JAR indicating which class in the application contains the main method.

Discovering Other SDK Tools

There are a number of other tools in the SDK that we won't cover in detail here, because they are often used only in advanced enterprise applications, or else are intended for dealing with fairly specific issues that we don't address here. It is a good idea at least to know about what tools are readily available for your use should your application call for them.

General Tools

These tools are not readily classifiable under one umbrella label, but they come with the kit and you might find them useful.

- javah Generates C programming language headers and stubs. You use it to write native methods.
- extcheck Detects JAR conflicts.
- jdb The Java debugger. Typically, you will want to use an interactive debugger available in an IDE, which is easier.
- javap Java class file disassembler.

Remote Method Invocation Tools

The RMI tools assist in creating applications that allow for interaction of components over the network.

- rmic Generates stubs and skeletons for remote objects. As of Java 5.0, RMI stubs are automatically generated for you.
- rmiregistry The remote object registry service.
- rmid RMI activation service daemon.
- serialver Returns class serialVerUID.

Java IDL and RMI-IIOP Tools

These tools assist in creation of apps that need to interact with CORBA/IIOP and OMG-standard IDL.

- tnameserv Provides access to the naming service.
- idlj Generates Java source code files that map an IDL interface and enable a Java application to use CORBA functionality.
- orbd Provides a way for clients to find and invoke persistent objects on a server in the CORBA environment.
- servertool Provides a convenient interface for registering, unregistering, starting, and shutting down a server.

Internationalization Tools

These help you create applications that are available to different locales. A locale customizes how data is presented and formatted for specific places in the world.

- native2 ascii Converts text to Unicode Latin-1 character set.

Security Tools

The tools in this set assist in setting security policies for your applications.

- keytool Manages keystores and certificates.
- jarsigner Generates and verifies JAR signatures.
- policytool A GUI tool for managing policy files.

Kerberos Tools

The following tools are specific to Kerberos, and if you're using Solaris 8, you have equivalent functionality provided by that OE.

- kinit Tool for obtaining Kerberos v5 tickets.
- klist Command-line tool to list entries in credential cache and key tab.
- ktab Tool to help user manage entries in the key table.

Java Plug-In Tool

This utility is used with the Java plug-in.

- unregbean Unregisters a package JavaBeans component over Active X.

Chapter 33. FAQ

THE LOWDOWN:

- Fast answers to common questions

This section contains a number of Frequently Asked Questions. It is meant to serve as a quick reference guide for when you're working and you need to find out how to do a variety of common tasks quickly.

Setting the CLASSPATH

Sometimes, your programs will reference external libraries that are custom libraries of your own, or a third-party library, or a package in the Java 2 Enterprise Edition. Use the set command to set the classpath environment variable. The classpath tells the Java Virtual Machine where to find necessary class libraries. These libraries can be directories, .jar files (most commonly), .class files, or even .zip files. The path to a .jar or .zip file ends with the file name; the path to a .class file ends with the directory name.

In Windows

First, open a command prompt. Then type

```
set CLASSPATH=.;C:\java\MyClasses;C:\java\My.jar
```

Note that setting the classpath in this manner will only sustain the variable for the current command prompt session. Closing the prompt and opening a new one resets the variable.

In Linux/UNIX

In the bash shell you can type the following to set the classpath:

```
export CLASSPATH=/home/someapp/MyClasses.jar:

$CLASSPATH
```

Import Declaration

The import command allows you to reference a class in the imported package only by its class name instead of having to type the fully qualified name every time you reference it in your class. For instance, java.lang.String is the fully qualified name for the String class. The String class is contained in the java.lang package. Because java.lang is always imported automatically, we are free to refer to a String without prefixing it with its package name.

To import all of the classes in a package, use a wildcard.

```
import java.sql.*;

// now you can refer to any class in the

// java.sql package by class name only in your code:


class myClass {

try {

    //look, ma! no package!

    Statement stmt = con.createStatement();

    //....

}...
```

Note that the import statement should be the second noncomment statement in your class file, following a package declaration.

You can import the static methods of a class using the new static import feature of Java 5.0:

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

Inheritance

The following example shows how to create a subclass using the extends keyword. In this example, Vehicle is the superclass, and Car is the subclass that inherits all of Vehicle's behavior. Car then will define behavior specific only to Cars.

```
class Vehicle {}

class Car extends Vehicle {}
```

Defining and Implementing an Interface

An interface is a set of requirements to which classes must conform if they want to use the service provided by the interface. To implement an interface, first declare that your class will implement the given interface; second, define the methods in the interface.

java.util.Collection has an interface that defines, among other methods, an add() method. This ensures that any implementation of java.util.Collection will support its methods.

An interface is defined as follows:

```
public interface Collection {

public abstract boolean add(A o)

...

}
```

An interface is implemented as follows:

```
public class myClass implements Collection {

//code here. can write, for instance, add() since

//add() is a method of java.util.Collection, and

//this class implements that interface. Eventually

//all methods in Collection must be implemented...

...

}
```

Exceptions

Throwing an exception means that you don't want to deal with it: You will let the caller handle it (or pass the buck too). Declare that your method throws an exception using the throws keyword, like this:

```
public void someMethod() throws Exception {
    if(somethingBad) {
    throw new Exception("Something bad happened: ");
    }
}
```

If you are using library code that throws an exception and you don't want to deal with it in a try/catch block, you can just say that your method throws this exception, and not think about it.

Alternatively, you can catch an exception to handle the problem inside the method, as shown here:

```
    public void someMethod() {
        // code for an operation that could cause an
        // exception...
        try {
        // ...some code
    } catch(Exception e) {
        // exception handling code here
    }
    finally {
        // this code executes whether an exception
        // was generated or not. The finally clause
        //is optional
    }
}
```

Working with JAR Files

A JAR file (Java ARchive) uses the same algorithm as a Zip utility does to compress numerous documents into one archive document. The benefit of a .jar file is that the developer can combine HTML, applets, class files, images, sounds, and everything else that makes up an application, and put them all into a JAR to facilitate easy distribution of the application. The jar tool is a Java application tool, regularly available with the SDK.

Note that you can add v to the command-line call options for verbose output, to indicate what is happening while your JAR is being created.

1. Creating a New JARNavigate to the directory in which you want to create the JAR.
2. Type the following at the command line:

```
jar cf myJar *.class
```

Adding All of the Files in a Directory to the JAR

1. Navigate to the directory in which you have the JAR.
2. Type the following at the command line:

```
jar cvf myApp.jar
```

Extracting the Contents of a JAR

FRIDGE

If you type the jar command and, instead of working, Windows tells you something like, "'jar' is not recognized as an internal or external command, operable program, or batch file," it means that the system cannot find the jar program to run it. So we have to put the jar program on the path (the list of places that Windows goes to find things to execute). We do this by setting the PATH environment variable to include the location of your bin directory (<JAVA_HOME>\bin), because that's where the jar program lives.

1. Navigate to the directory in which you have the JAR.
2. Type the following at the command line:

```
jar -xvf myApp.jar
```

You should see the program extracting the files into the pwd (Present Working Directory).

What Is the Java Virtual Machine?

The JVM interprets and executes bytecodes, which are the executable representation of a Java program. The JVM emulates a hardware platform (hence, the name Virtual Machine), including registers, program counter, and so forth.

What's In the SDK?

The SDK consists of the following items:

- A JVM, or Java Virtual Machine
- A Java compiler, which takes a plain text source code file and turns it into executable bytecodes
- appletviewer, a program for viewing applets
- A debugger
- Example java programs
- A tool for creating and managing security keys
- A remote method stub and skeleton generator
- A registry server for handling remote method invocations (method calls coming from objects on a different JVM)
- C and C++ headers for extending the JVM with interfaces to native code
- Native libraries for embedding a JVM into other native applications
- Native libraries for the portions of the JVM that are platform dependent, such as those that help generate graphical user interface components.

Setting JAVA_HOME in Windows

The environment variable JAVA_HOME sets the location of your Java SDK so it can be accessed from various locations.

On Windows XP

1. Navigate to Start > Control Panel > System.
2. Choose the Advanced tab.
3. Click Environment Variables.
4. In the System Variables pane, click New.
5. For Variable Name, type JAVA_HOME.
6. For Variable Value, type C:\j2sdk1.5.0 for a default installation of the SDK. You may need to adjust your value based on your installation.

On Linux/UNIX

1. Type the following code at the shell (you may need to change the location of the Java home directory for your system):

```
$ export JAVA_HOME=/home/usr/jdk1.5
```

2. To set the JAVA_HOME into the PATH, type the following:

```
$ export PATH=$JAVA_HOME/bin:$PATH
```

Setting the PATH in Windows

The PATH environment variable tells the system where to find the java, javac, jar, and other executables. This enables you to compile Java classes, run your programs, make .jar files, and so forth, from any directory on your computer.

1. Navigate to Start > Control Panel > System.
2. Choose the Advanced tab.
3. Click Environment Variables.
4. In the System Variables pane, double-click the PATH variable. Scroll to the end of the list of variables, and add the following text (for default installation of SDK 1.5.0):

```
C:\j2sdk1.5.0\bin
```

Checking Current Java Version

It is not uncommon to have many different JREs running on the same machine at the same time. To check the current default version:

1. Open a command prompt.
2. Type `java –version`.

You will see output similar to this:

```
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0-b91)

Java HotSpot(TM) Client VM (build 1.5.0-b91, mixed mode)
```

Compiling and Running Programs

1. Open command prompt or shell
2. Change directories to directory where the .java file you want to compile is stored by typing `cd`.
3. To compile: `javac myClass.java`
4. To run: `java myClass`

Primitive Data Types

byte	- 8 bits
short	- 16 bits
int	- 32 bits
long	- 64 bits
float	- 32 bits
double	- 64 bits
boolean	- True or false
char	- 16 bits, Unicode

Declaring and Initializing Variables

Primitives

```
boolean hasRoadRage = true;

char size = 'm';

int age = 11;

long userID = 42L;
```

Arrays

Arrays can be created and populated as shown in the following. Make an array that has two cells for holding int values. The two cells are then populated with the values 11 and 12, respectively.

```
int[] numbers = new int[2];

numbers[0] = 11;

numbers[1] = 12;
```

For the preceding array, calling array.length returns 2.

In the next example, an int array is created with three cells and populated at the same time.

```
int[] numbers = {1,2,3};
```

Arrays can also be populated in loops. Here, we create a new object to be stored in each cell of an array.

```
Employee[] employees = new Employee[5];

for (int x = 0; x < employees.length; x++) {

employees[x] = new Employee();

}...
```

Objects

Create a new object of type Button like this (assuming you have imported the javax.swing.Button class):

```
//reference type of Button, variable name is

//saveButton, and we use the constructor that accepts

//a String label. here we set the label's value to

//'save' right away.


Button saveButton = new Button("save");
```

Class Definition

You can have multiple class files in the same source (.java) file, but only one of them may be declared public.

```
public class MyClass {

    // class definition here

}


class AnotherClass {}
```

Class Modifiers

You can add class modifiers to the declaration that changes the definition of the class.

public

This class modifier indicates that you want it to be available to classes other than just those in the package in which it is declared. That is, you'll let anybody use it. If your class does not include the public modifier, it can only be used by other classes within the same package.

final

To make your class so it cannot ever be subclassed, you include the final modifier in the class declaration. This example is from the standard API.

```
public final class String {...}
```

abstract

Declare an abstract class to indicate that its implementation is not complete (that you didn't write code to do the work in all of the methods). Only abstract classes may have abstract methods. For example, the following is *illegal*:

```
public class MyClass {

public abstract void someMethod();

}
```

In the preceding example, you must either implement the method or declare the class abstract and implement the method in a subclass.

strictfp

This modifier indicates that your class will use the strict IEEE 754 standard for floating point (fp) variables, whether or not the methods or variables themselves are declared strictfp.

Declaring Methods

This example declares a method called setQuantity() that accepts a variable called qty of type int. Note that this is only something you can do in an abstract class or interface—declare it without implementing it.

```
public void setQuantity(int qty);
```

This example declares a method called getQuantity() that returns a variable of type int.

```
public int getQuantity();
```

Methods declared without an access modifier will have default access.

```
void setQuantity(int qty);
```

Calling Methods

Methods are called using dot notation against the object that defines the method. Your code has to account for the return type that the method declares. For example, if getQty returns an int, you call the method to set a variable of type int to the return value. Say that the getQty method returns an int with a value of 5:

```
int theQty = inventory.getQty();

//theQty now has a value of 5
```

You can also call a method directly to get the value without having to set its value into a variable. This is useful in constructors in particular: Say that we have an employee class that defines a constructor that accepts an int value for an ID and a String for a name, and a Manager class with a similar constructor:

```
Employee employee = new Employee(999, "Kevin Bacon");

Manager manager = new Manager(employee.getID(),

employee.getName());
```

You may also have a class that defines static methods—that is, methods that do not require an instance of the class to operate. An example from the API follows:

```
Collections.sort(myArrayList);
```

Notice that we call the sort method directly on the Collections class. The Collections class happens to consist exclusively of static methods that work with collections data.

Overloading Methods

Overloading a method means defining multiple methods in the same class, each with the same name, each of which accepts a different argument list.

```
public void calculateSquareRoot(int i);

public void calculateSquareRoot(double d);
```

The preceding calculateSquareRoot() method is said to be overloaded.

Overriding Methods

Overriding a method means defining a method in a subclass that has the same signature as the method of the same signature in the superclass. For example, the String class overrides the equals() method defined in its superclass Object. The following class definitions show this in action:

```
public class Test {

public void printQuestions(){

System.out.println("Multiple Choice...");

}


public static void main(String[] args) {

Test test = new EssayTest();

test.printQuestions();

}

}


class EssayTest extends Test {

public void printQuestions(){

System.out.println("Essay...");

}

}
```

Executing this file prints "Essay" to the standard out.

Reading User Input from the Standard In

Sometimes, you want to accept user data at runtime from the console. The following code example demonstrates how to read user input from the standard in (System.in).

```
//do imports and such

public static void main(String[] args) {

    System.out.println("Please say something ");

    try {

        BufferedReader is = new BufferedReader(

                new InputStreamReader(System.in));
        String input;

        while((input = is.readLine())!= null){
            //consider doing something even more
            //exciting here with the input
            System.out.println("You wrote: " + input);
          is.close();
          System.exit(0);
        }
      } catch(IOException ioe){
        System.err.println("Exception: " +
                              ioe.getMessage());
      }
    }
```

Package Declaration

A package is a logical and physical grouping of Java classes and interfaces, whose purpose is to minimize name conflicts. When using the package statement, it must be the first noncomment line in your source file. A package is declared using the package keyword as shown here:

```
package net.javagarage.utils;

// class definition here...
```

Chapter 34. PACKAGING & DEPLOYING JAVA APPLICATIONS

DO OR DIE:

- How to package your application in a compressed JAR file
- How to create an executable JAR
- How to make your application presentable with a shortcut and icon.

It is likely that you will take your Java skills and head for the Web. That is not a bad idea, considering that roughly 80% of all new Java development happens on the Web. Although 68% of statistics are made up on the spot, this one indicates that not much is happening on the desktop for Java. Sun is doing considerable work to change this perception, and indeed there are a number of fantastic things happening on the desktop with Java. To stay on top of cool new developments with Java GUI applications, check out the Swing Connection (and Swing Sightings in particular) online at http://java.sun.com/products/jfc. You also might enjoy project Looking Glass, which brings a real 3D feel to OS windowing.

But programs come in many forms. There are enterprise applications, systems development, desktop applications, embedded systems, and micro applications. And many of these environments either require or encourage your packaging of an application in some form. Packaging is the process of preparing your application for easy deployment or distribution. Deployment is the process of putting your application in a context where it can be run by a user.

Herding Cats

Imagine trying to distribute an application, uncompressed, to end users: "Now downloading file 2 of 768…". This is what my friends from Texas call herding cats. It is obviously easier to deal with one file than many files, so typically packaging involves taking the many files in your application and putting them into one or two manageable files that can then be FTP'd or copied across the network or burned onto a DVD for sale in stores.

Making One File: Packaging into a JAR

If you have used Microsoft's Visual Studio, you know that it is very easy to make a program you write into an executable file. In Java, you don't make .exe files as you do in Visual Studio. Instead, you make a JAR file, which can be executed on a system that has a Java Runtime installed. It can be a bit of a drag, more of a drag in my view than it should be for a language this mature. But there is always a trade-off between convenience and control. In the long run, it is nice to have the control; so, okay, I will stop whining about it and thank Sun for their prescience. Now let's see how to do it using the JAR tool.

Packaging Using the JAR Tool

A JAR file is a file compressed with the same algorithm used to compress Zip files. JAR stands for Java ARchive. You create a JAR file using the jar command. If you are on Windows, go to this directory: <JAVA_HOME>/bin/. This is where the tools in the SDK live, and therefore where you find the jar program.

GOT JAVA EXECUTABLES?

If you don't care about being a Java "purist" and know that all of your users are on a Windows environment, you can compile your program to native code, bundling your application as an .exe file, executable only in a Windows environment. Although the JDK does not ship with any native compilers, some IDEs allow you to do this (JBuilder for example). There are also a few free programs out there that, as you might imagine, work to greater or lesser degrees. However, as JoJo the Internet Guy says, this is not a five-star idea. You obviously lose portability, perhaps even some functionality, and there are alternatives. For example, you could write a batch file or shell script to make things a little less punishing for your users. We'll see how to do this in a moment. But if you must have things your way, here are some starting places:

JBuilder

http://www.xlsoft.com/en/products/jet/

http://www.excelsior-usa.com/jet.html

If you type jar at the console, the program will display usage options. Here, we will create a JAR file in the C:\apps directory called BlogApp.jar. It will contain all of the files with a .class extension in the target directory.

In running the command, the c is for "create," the v for "verbose" (which means, "tell us what you're doing in detail as you do it"), and f means that you will specify the resulting file name. Run the following command to create the JAR:

```
C:\>jar cvf BlogApp.jar blogger

added manifest

adding: blogger/(in = 0) (out= 0)(stored 0%)

adding: blogger/BloggerFrame$1.class(in = 1017) (out=564)

(deflated 44%)

adding: blogger/BloggerFrame$2.class(in = 864) (out=479)

(deflated 44%)

adding: blogger/BloggerFrame$3.class(in = 828) (out=458)

(deflated 44%)

adding: blogger/BloggerFrame$4.class(in = 741) (out=401)

(deflated 45%)

adding: blogger/BloggerFrame.class(in = 3801) (out=1945)

(deflated 48%)

adding: blogger/BloggerKeys.class(in = 670) (out=422)

(deflated 37%)

adding: blogger/blogs/(in = 0) (out= 0)(stored 0%)

adding: blogger/blogs/blog.html(in = 52) (out= 42)

(deflated 19%)

adding: blogger/StartApp.class(in = 884) (out= 520)

(deflated 41%)

C:\>
```

Executing the command, as the output shows, creates a new file in the current directory called blogger.jar. Class files are compressed as added, and a manifest is created. A manifest is a text file that specifies meta data about your application. It must be in a directory called META-INF and the file itself must be called MANIFEST.MF (both the file and the directory are all uppercase). The chief purpose of the manifest file is to point the runtime to the location of the class file containing the main method so it can start the program.

Adding the Manifest

The JAR tool can create a manifest for you if none exists. But then it will not contain some necessary information, such as the class containing the main method. You will need to then extract the JAR, add the information to the manifest, and recompress the JAR. Might as well just create the manifest right off the bat.

Navigate into the JAR and open the META-INF directory and then the MANIFEST.MF file. This is a plain text file in which you can specify things that you want the application to know at runtime. Although the JAR tool created this folder and this file, it did not pick out for us the class with the main method, which it will need to know in order to start the application automatically. So, we will just write our own manifest in a plain ASCII text file. Do this:

1. Make sure that your classes are compiled in the appropriate structure (that is, in their packages).
2. Write a plain text file called manifest.mf. In this file, type the following: Main-Class: package.otherpackage.MainClass. Do not include the .class extension. Your text here will be used as an entry in the real MANIFEST.MF file produced by the JAR tool.
3. Navigate to the working directory one level *above* the first-level package in your app.
4. Run the jar command like so: jar -cvmf MANIFEST.MF MyApp.jardirectory/subdirectory, where subdirectory contains your class files.

This will create a JAR file called MyApp.jar in the current working directory:

```
jar -cmf MANIFEST.MF MyApp.jar

    net/javagarage/test/StartApp
```

Note that there are many different subtle ways to manipulate JAR file creation. Check out the usage information if you need more control.

Creating JAR Files with Eclipse

This process is somewhat simpler using the Eclipse IDE. Any reasonable IDE should give you the ability to create a JAR for your application; we use Eclipse here because it is free and pretty reliable and we've been using it a lot in this book.

1. With an open project, click File > Export....
2. Select the JAR file and click Next.
3. Click on the application in the Select the Resources to Export window, because you may not want to include everything. Choose the classes directory (and the source code if you want to distribute it).
4. In the Select the Export Destination text field, choose where you want your JAR to end up, and give it a name.
5. Click Next, and Next again.
6. Click the Use Existing Manifest from Workspace radio button and browse to the location of your manifest. Click Finish and you're done.

Executing the JAR file

FRIDGE

If you have problems running this command, as many people do, try the following hack, which seems to do the trick sometimes: Put the line with the Main-Class entry immediately following the Manifest-Version: 1.0 line (if you create the complete manifest manually). Also, make sure there is a carriage return after your Main-Class entry, or it won't work properly. Sheesh.

To execute your JAR file, type the following command at the console:

```
java -jar MyApp.jar
```

If you need to pass arguments into your application, you can pass them in just as you normally do with the java command:

```
java –jar MyApp.jar myFirstArg mySecondArg
```

This passes your executable JAR file to the java program for execution.

Alternatively, in Windows, you can double-click the JAR file.

If you get an error message saying, "Could not find the main class," this is one time you shouldn't believe what you read. There obviously is a problem; it just might not be the problem you're told it is. It could mean that there is a dependency that your application has that is unfulfilled. For example, if your application sends an e-mail using the javax.mail package, you need to put the J2EE library in your application folder and add an entry for it in the manifest. Specify that you need the j2ee.jar file on your classpath this way:

```
Class-Path: j2ee.jar
```

Then, make that file accessible to your application.

Note too that JARs can store other JARs. This is commonly done to deploy a complete application. Say you have an executable JAR, and then some other files that you want to distribute with that executable JAR (like a README.txt, a properties file, etc.). That you can do, as long as you are okay with your users un-JARing that distribution JAR in order to get to your executable JAR.

But say you have a JAR file that is a library of classes that do some particular thing, such as make PDF files. You've downloaded this JAR, and your application code relies on it. You need to *un-jar* (extract) that JAR so that the classes in it are all flat with your application classes. Then you can re-JAR the whole thing all together. Otherwise, your executable code won't know how to find the classes it needs.

Making your Java Program Execute Automagically on Startup

So there I was on the corner of Central and Jefferson, just holding up a lamppost, if you got to know, when up comes this little sassafras grousin' for a light and just like that he says a me, "I got a Java program, see. And it needs to start when my OS starts, automatically." Here's how to do it in both Windows and Linux.

In Windows

Write a batch file that executes your program using the java command, and create a task that executes the batch file. Easy!

1. Navigate Windows Explorer to C:\WINDOWS\Tasks. Click Add Scheduled Task.
2. Click Next and browse to the batch file and click on it to select it.
3. Click the button that indicates when you want to run the program, type your password, and click Finish.

   Here is an example so you can get the feel for it.

   First, I create a file called MyStartupMessage.bat. It contains the following command:

   ```
   rem eben was here

   java net.javagarage.packaging.MyStartupMessage
   ```

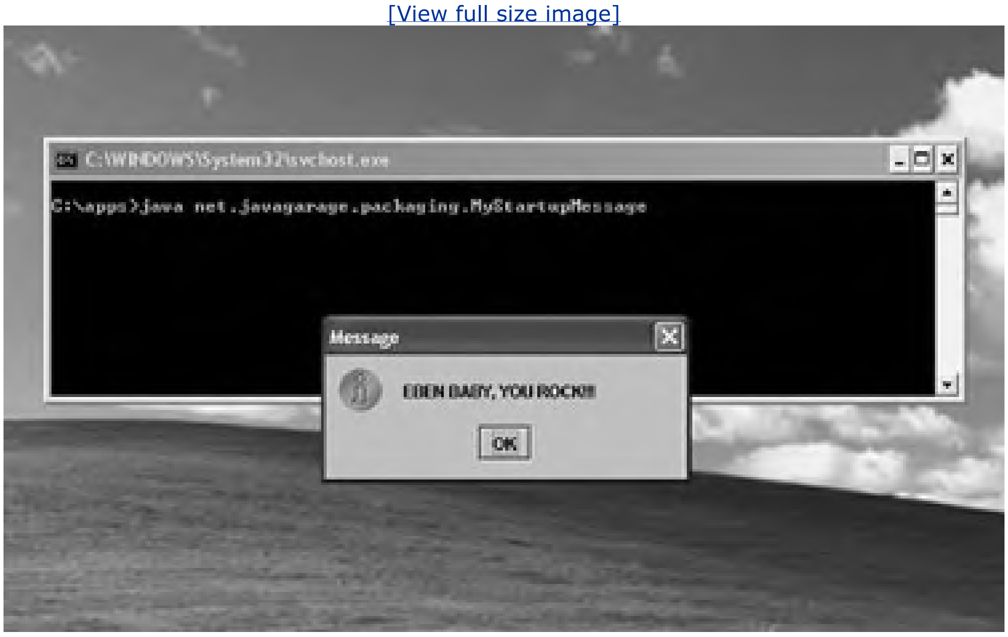
This batch file is in a folder called C:\apps. Also in that folder is the folder net, which contains the folder javagarage, which contains the folder packaging, which contains the Java program MyStartupMessage.class. That class contains the following important method:

```
public void helpSelfEsteem(){
    JOptionPane.showMessageDialog(null,
        System.getProperty("user.name").toUpperCase() +
        " BABY, YOU ROCK!!!");}

      //run the program
public static void main(String[] args) {
      helpSelfEsteem();
}
```

Just create the Windows task as indicated in the preceding, and you're good to go. Figure 34.1 shows the output I get.

Figure 34.1. The application executing automatically when the system starts up.

[View full size image]



```
C:\apps>java net.javagarage.packaging.MyStartupMessage
```

In Linux RedHat 7.3 and Later

The steps are similar if your target OS is Linux. If you've got a Linux box with a graphical user interface like KDE, you can just navigate to System Tools > Task Scheduler and add the task.

But say we have installed Linux as a server and don't have any graphical user interface libraries installed (in which case we can't run the preceding message dialog program). The corresponding function is called cron. Here's how to do it. Let's make a lame-brained Java program called MyStartupMessage that writes out a file telling you what time the server started.

It looks like this:

```
//do whatever work you want to perform after startup
private static void doWork(){
    //on my windows system this is
    //C:\Documents and Settings\eben

    String fileName = System.getProperty("user.home");

    //on windows this is \, on unix it is /
    String slash =
                System.getProperty("file.separator");
    fileName += (slash + "startupMessage.txt");

    File file = new File(fileName);
    try {
        BufferedWriter out = new BufferedWriter(
                              new FileWriter(file));
        out.write("The server restarted at " +
                                    new Date());
        out.close();
    } catch (IOException e) {
    System.err.println(e.getMessage());
}
```

Now check your user home folder; the file is in there with the timestamp.

PREV < Day Day Up > NEXT

Creating an Icon for Your Java Application on Windows

Say that you have created your JAR, and created a batch file that performs certain useful tasks such as setting environment variables, and now you want to make it pretty like a real, professional type application. To do this, create an icon, create a shortcut to the .bat file, and then change the icon used by the shortcut.

1. Right-click on your .bat file or executable JAR file. Select Create Shortcut.
2. Select Properties.
3. Click Change Icon....
4. Choose one of the many icons included with Windows just for a test. Browse to the location of your .ico file and click Apply for the real deal.

Don't be discouraged with all of this deployment business. There are ways of automating all of the necessary file copying, moving, deleting, generation, and even the Java compiling, JAR-ing, and more. The most popular way by far of automating all of this business is Ant. You can read about Ant, and download it for free at http://ant.apache.org. It integrates well with the most popular IDEs, and if you are using Eclipse (http://www.eclipse.org), then you already have Ant.

FRIDGE

What's that? You don't have an .ico file? Such a file is an icon, a special kind of bitmap. You need an editor that will help you create them; you'll have a hard time in Photoshop alone or other typical graphic editors. I like the free Icon Studio from CoffeeCup software (http://www.coffeecup.com/freestuff/). It's full-featured and you can't beat the price. Just download and install it, then create your own icon and choose it for use on your shortcut. Windows will replace it immediately, and you're good to go.

Chapter 35. TOOLKIT

DO OR DIE:

Demonstrate a number of complete, varied, useful, quality apps, including

- Name: A simple data class
- A Credit Card Validator
- Garage Pad: A Simple Text Editor
- An RSS Feed Reader
- Garage Doodler: A Drawing Pad

This topic offers a gallery of straight-shooting Java applications that illustrate how to use the concepts we've covered in complete, working, real-world type applications. After all, that is the purpose of all this—to get us writing complete applications. The idea is that you will really understand how the concepts fit together if you *see* them fitting together. It can be very frustrating to read a book filled with tiny code snippets, none of which compile on their own, so I didn't want to do that.

Here, therefore, is a collection of different small applications that you can study, incorporate into your own projects, or otherwise use freely as you see fit. Although I don't warrant the usability of this code for any particular purpose, it is presented here in order to support the topics of this book, to help you learn to program Java better. Translation: Use it, sell it, do whatever you want with it; I wouldn't launch my space shuttle on it though, and I wouldn't wonder (aloud, to me, in e-mail) why certain "features" are "missing," because its purpose is academic. Though it would generally be fine in production. Some concepts are presented in a simplified, or repeated manner, in order to help the ideas sink in.

Comments interspersed frequently throughout the files provide the discussion. This will hopefully encourage you to use code comments to communicate with the inheritors of your applications, instead of relying on model documents or using case diagrams.

A Name Data Class

Often in books, we see programs that learn us how to do things like calculate and draw and other action-verb-oriented things. In business, however, we often need to also represent things. Object-oriented programming has as one of its tenets that an object contains data and behavior. Sometimes, however, this behavior is either fairly limited, or is implemented fully in increasing usability. Take, for example, a simple thing like a person's name.

Many applications need to represent names, but they often are implemented poorly. On the one hand, a complete name gets represented as a single String. This makes it almost impossible to deal with in searches, comparisons. Other times, names are only slightly more conscientiously dealt with by offering separate Strings for each part of the name. This does make searching and comparison more accurate, but has the deleterious effect of decoupling related data, which is a maintenance nightmare. If your manager tells you that now your name collection form needs to include a middle name, for instance, you must update many different parts of the application. In a Stuts-based Web application, for instance, this would be a real drag.

If, however, you are dealing with a Name *class*, the organization of your data is shielded (encapsulated) behind it, so your maintenance is far easier. But you know that already. What are some other benefits to this approach? Well, for one thing, you gain total control over presentation. You can display this data exactly how you want, and offer ways for clients to include certain parts or omit others (such as the ever-questionable middle name). It also makes sure (well, encourages you, if you are being lazy) that you override equals and hashcode, and think hard about your application design and implementation.

The downside is that a Name is generally given to another entity, such as a Person. So you probably have a Person class, with a Name member, and maybe an Address member (which would be a complex object itself!), and so on, and all of a sudden you have proliferated classes and slowed down your application as it goes to the trouble of creating seven or eight objects when all you wanted was two lousy little Strings. Now, if you aren't going to use any of that additional information in a very sophisticated way, that might be overkill.

Obviously, you need to determine what your application is focused on, how many concurrent users it is likely to have, what the data will be used for, and what kinds of views you will need of the data before you go off the deep end like this. Sometimes a couple of Strings is just what the doctor ordered.

Hey! This class uses generics, in order to give you a taste for it. They are not explained in this book, and neither is the Collections API. Don't worry. The following ArrayList business simply says this: "Make a list (which is like an array that is resizable), and let it accept only objects of type Name." Then we just call the static sort method to sort them according to our implementation of the Comparable interface.

Demonstrates

Overloaded methods, overriding methods, overloaded constructors, using generic types, using Lists, implementing interfaces, sorting Collections, using switch/case constructs, constants, JavaDoc, formatting, and usability considerations.

Name.java

```
import java.io.Serializable;

import java.util.ArrayList;

import java.util.Collections;

import java.util.List;


/** <p>

 * Represents a person's name in numbing detail.

 * Meant for use by another class (i.e, "Person").

 * Implements the Comparable interface so that Name
 * objects can be easily sorted alphabetically.
 * Implements the Serializable interface, which is
 * merely a marker interface, in order to indicate
 * that it is safe to serialize objects of this type.
 * This class will only compile under SDK 1.5.
 * </p>
 * @author eben hewitt
 **/
public class Name implements Serializable, Comparable<Name> {

    private String title=""; // ie, "Miss" or "Dr."
    private String suffix=""; // ie, "junior" or "III"
    private String given=""; //first
    private String middle=""; //multiple middle
    private String family=""; //last

    public static final int FAMILY_GIVEN_MIDDLEINIT = 0;
    public static final int FAMILY_GIVEN = 1;
    public static final int GIVEN_FAMILY = 2;
    public static final int GIVEN_MIDDLE_FAMILY = 3;
    public static final int GIVEN_MIDDLEINIT_FAMILY = 4;
    public static final int FULL_NAME_US = 5;
    public static final int PREFIX_FULL_SUFFIX_US = 6;

    //overloaded constructors
    /**
     * Constructs Name with first and last names.
     */
    public Name(String given, String family) {
        this.given = given;
        this.family = family;
    }
    /**
     * Constructs Name with first, middle,
     * and last names.
     */
    public Name(String given, String middle,
                String family) {
        //re-use the constructor defined for just
```

```
         given + family
    this(given, family);
    this.middle = middle;
}
/**
 * Constructs complete Name object. You can have
 * multiple overloaded constructors. First calls
 * the earlier constructor—that call must be
 * first line of code here.
 */
public Name(String title, String given,
             String middle,
             String family, String suffix) {
    this(given, middle, family);
    this.title = title;
    this.suffix = suffix;
}

// helper methods
/**
 * Gets initial character of first name as an
 * initial (with an added ".").
 * For example "Eben" returns "E.".
 * @return The initial character of the first
 * name.
 */
public String getGivenInitial() {
    return given.charAt(0) + ".";
}
/**
 * Gets initial character of middle name as an
 * initial (with an added ".").
 * For example "Mann" returns "M.".
 * @return The initial character of the middle
 * name.
 */
public String getMiddleInitial() {
    return middle.charAt(0) + ".";
```

```
}
/**
 * Gets initial character of first name as an
 * initial (with an added ".").
 * For example "Hewitt" returns "H.".
 * @return The initial character of the last
 * (family) name.
 */
public String getFamilyInitial() {
    return family.charAt(0) + ".";
}
/**
 * Gets initial character of each of first,
 * middle, and last names
 * (with an added ".").
 * For example "Eben Mann Hewitt"
 * returns "E. M. H.".
 * @return The initial characters of the first,
 * middle, and last names.
 */
public String getInitials() {
    return getGivenInitial() + " " +
              getMiddleInitial() + " " +
              getFamilyInitial();
}

/**
 * Returns the various parts of a name together
 * in common representations.
 * @param repType The manner in which you want
 * the Name represented, useful
 * for different purposes:
 * <ul>
 * <li>(default) FULL_NAME_US: Eben Mann Hewitt
 * <li>FAMILY_GIVEN_MIDDLEINIT: Hewitt, Eben Mann
 * <li>FAMILY_GIVEN: Hewitt, Eben
 * <li>GIVEN_FAMILY: Eben Hewitt
 * <li>GIVEN_MIDDLEINIT_FAMILY: Eben M. Hewitt
 * <li>PREFIX_FULL_SUFFIX_US: Mr. Eben Mann
 * Hewitt, Esq.
 * @return The name, as represented by
 * the parameter.
 */
public String getName(int repType) {
    switch (repType) {
    case FAMILY_GIVEN_MIDDLEINIT:
         return getFamily() + ", " + getGiven()
                     + " " +
                     getMiddleInitial();
    case FAMILY_GIVEN:
          return getFamily() + ", " + getGiven();
    case GIVEN_FAMILY:
          return getGiven() + " " + getFamily();
    case GIVEN_MIDDLEINIT_FAMILY:
          return getGiven() + " " +
                getMiddleInitial() +
                " " + getFamily();
    case PREFIX_FULL_SUFFIX_US:
    //ternary operators used to remove
    //unnecessary whitespace
    //in the event that title or suffix is blank
          return (getTitle() != "" ? getTitle()
              + " " : "") + getGiven() +
              " " + getMiddle() + " " +
              getFamily() +
              (getSuffix() != "" ? (", " +
              getSuffix()) : "");

    default :
    case FULL_NAME_US:
          return getGiven() + " " + getMiddle() +
                 " " + getFamily();
    }
}
/**
 * Override of Object's toString method.
 */
public String toString() {
    return getName(GIVEN_FAMILY);

}


/**

 * Override of Object's equals method.

 * Determines object equality by testing family,

 * middle, and given names as well as class.

 *

 */

public boolean equals(Object other){

    //are they the same object?

    if (this == other)

          return true;

    //are they of same type?

    if (other != null && getClass() ==

                 other.getClass()){

          //since the classes are the same type,

          //downcast and compare attributes

             Name n = (Name)other;

             if (n.getFamily().equals

                   (this.getFamily()) &&

                n.getGiven().equals

                   (this.getGiven()) &&

                n.getMiddle().equals

                   (this.getMiddle()))

          return true;

    }

          return false;

}


//overloaded getName() method:

public String getName(){

    return getName(PREFIX_FULL_SUFFIX_US);

}



/**

 * Required by the Comparable interface. Returns

 * 0 if the objects have the same
 * first-middle-last combination. Its purpose is
 * to help us perform during a call to
 * {@link java.util.Collections#sort}.
 * Alphabetical order is preferable here.
 * @return int indicating if the objects can be
 * considered equivalent. A value of 0 indicates
 * equivalency. A negative value indicates this
 * Name comes earlier; a positive value indicates
 * this Name comes later.
 * @throws ClassCastException
 * @see java.lang.Comparable#compareTo
 * (java.lang.Object)
 */
public int compareTo(Name anotherName) throws
   ClassCastException {
   if (! (anotherName instanceof Name))
          throw new ClassCastException("Object
                      must be instance " +
                      "of Name to compare. ");

   String thisFirstAndLast = this.toString();
   String otherFirstAndLast =
             anotherName.toString();

   return
   thisFirstAndLast.compareTo(otherFirstAndLast);

}

//get and set methods
public void setGiven(String given) {this.given =
         given; }
public String getGiven() { return given; }
public void setMiddle(String middle)
        { this.middle = middle; }
public String getMiddle() { return middle; }
public void setFamily(String family)
        { this.family = family; }
public String getFamily() { return family; }
```

```
    public void setTitle(String title)
        { this.title = title; }
    public String getTitle() { return title; }
    public void setSuffix(String suffix)
        { this.suffix = suffix; }
    public String getSuffix() { return suffix; }


    //included just for testing
    public static void main(String[] ar){
      Name homer = new Name("Homer", "J.",
            "Simpson");
      Name marge = new Name("Marge", "Simpson");
      marge.setMiddle("B.");
      System.out.println("Are they same? " +
            homer.equals(marge));

      System.out.println(marge.getName
  (Name.FAMILY_GIVEN_MIDDLEINIT));

      System.out.println("Implicit toString: " +
          homer);
      System.out.println("Is Homer equal to Homer?
          " +
          homer.equals(new Name("Homer", "J.",
              "Simpson")));

      //illustrate the sorting ability
      List<Name> names = new ArrayList<Name>();
      names.add(marge);
      names.add(homer);
      //sort and print

      Collections.sort(names);

      for (Name aName : names) {
         System.out.println(aName);
        }
    }
}
```

Result

```
Are they same? false

Simpson, Marge B.

Implicit toString: Homer Simpson

Is Homer equal to Homer? true

Homer Simpson

Marge Simpson
```

If we had not sorted the previous list of names, then they would be returned in the order added ("marge" first, and "homer" second).

The previous Name class is useful for modeling data-type classes. Of course, classes in Java are meant to hold both data and the means of performing operations, but some classes offer a service and some classes represent simple nouns. Such as "Name." Cheers.

(source: AP, Rutgers University Study, Douglas L. Kruse)

AMERICAN COMPANIES SAVED MORE THAN $155,000,000 IN 1996 BY ILLEGALLY HIRING UNDERAGE CHILDREN. THIS PRACITCE CONTINUES TODAY.

PREV < Day Day Up > NEXT

Credit Card Validator

This little program validates credit card numbers according to their structure. That is, it does *not* tell you anything regarding the account associated with the credit card number passed to it. All it does is tell you that the argument passed could be a real credit card number. It does this using the Luhn formula. This app is easily incorporated into any e-commerce application you might have in place, and works well as a front-line validation before incurring the overhead required to check the account and process the transaction.

Demonstrates

Converting Strings, ternary operator, validating input, designing the methods of a class, JavaDoc, using the Character class, for loops, arrays.

```
package net.javagarage.misc;


/** <p>
 * Shows how to validate a credit card number.
 * Credit card numbers should check out against
 * the LUHN formula (MOD 10 algorithm).
 * <p>
 * Will check out fine against the numbers commonly
 * used for testing, such as 4111 1111 1111 1111
 * <p>
 * On a credit card, the first number identifies
 * the type of card (MC, Visa, Amex, etc), and the
 * middle digits belong to the bank, and the last
 * digits to the customer.
 * <p>
 * Gives you code that you might one day really need,
 * and demos how to use for loops, Character,
 * separate work, and use the ternary operator.
 *
 * @author eben hewitt
 * @see Character,
 **/
public class CreditCardValidator {


/**
 *Use private constructor to disallow
 *creation of objects. it isn't necessary
 *since we're just doing an operation, not
 *storing data
 */
private CreditCardValidator() { }

/**
 * Determines if the number checks out against the
 * algorithm—obviously it does not perform any
 * operation with respect to the user's account.
 * <p>
 * @param String the number to be validated
 * @return boolean true if the card is valid,
 * false otherwise.
 **/
public static boolean validate(String input) {
    //first off, make sure we're even
    //in the ballpark of a real number
    //(not NULL and between 13 and 16 digits)
    if (input == null ||
        (input.length() < 13) ||
        (input.length() > 16)){
    reject();
}
//remove spaces, dashes, etc
String number = cleanup(input);

//this will be our total
int total = 0;
//multiply all digits but the first one by 2
int position = 1;
//Starting with the second-to-last-digit...
for(int i = number.length() - 1; i >= 0; i—) {
// make each digit in base 10
int digit = Character.digit(number.charAt(i), 10);
//multiply every other digit by 2
//starting with the second one
digit *= (position == 1) ? position++ : position—;
//if the digit is greater than 10,
//try mod 10 + 1

total += (digit >= 10) ? (digit % 10) + 1 : digit;

}


//a valid number MOD 10 will be 0

//the result of this expression is the boolean return value

return (total % 10) == 0;

}


/**

 * Clean the passed string so that it contains

 * only numbers, not spaces or dashes or anything

 * extraneous.

 * @param String the number as entered by user

 * @return String the cleaned-up number

 **/

public static String cleanup(String input) {

char[] result = new char[input.length()];

//index of the char array

int idx = 0;

for (int i = 0; i < input.length(); i++) {

char thisChar = input.charAt(i);

//use static Character wrapper class

//method to see if it's a digit

if (Character.isDigit(thisChar)) {

//it is, so keep it

result[idx++] = thisChar;

}

}

/*

 * String has a constructor that builds a

 * String from a subset of a char[], using

 * the char[], the starting position (here, 0),

 * and the ending position (here, idx).
 */
return new String(result,0,idx);
}

/**
 * Used when the user passes bad information to
 * stop the show and help them out.
 */
private static void reject(){
System.out.println("usage->java CreditCardValidator number");
System.exit(-1);
}

//start the app
public static void main(String[] args) {
if (args.length == 1) {
System.out.println("Is valid number? " +
validate(args[0]));
} else {
//user didn't enter a card number
reject();
}
}
}//eof
```

Result

The result of running the validator with different arguments is shown here.

Passing 12345 yields this result:

```
usage->java CreditCardValidator number
```

That's because the program rejects the argument if it is obviously not anything like a credit card number (between 13 and 16 characters).

Passing 456789098765432 yields this result:

```
Is valid number? false
```

Passing 4111111111111111 yields this result:

```
Is valid number? true
```

In the next few sections, we'll look at putting together a text editor, an RSS newsreader, and a drawing pad. It will be very thrilling and educational.

Application: SimpleTextEditor: Garage Pad

This program, the Garage Editor, is a simple text editor that allows users to open files, save their file, copy and paste, and display help information.

This is a terrific little application because it demonstrates a number of concepts important in the creation of real Java programs.

Demonstrates

Creating a basic GUI window, creating scrollbars on demand, creating a toolbar containing menus that contain menu items, how to use the JFileChooser to save and open files, cleanly closing an open document, exiting from an application, displaying informational messages to the user (like JavaScript alerts or Visual Basic MsgBox), using keyboard mnemonics (using key combinations such as Ctrl+S for Save), and using the preferences API to save user-specific preference data regarding how the application is run. In general, this is a pretty tight application, and does all its work in one file.

LUHN ALGORITHM FOR VALIDATING CREDIT CARDS

The LUHN (or MOD 10) algorithm used for validating credit card numbers is widely known and accepted. Here is how it works:

1. Start at the second digit from the right. Double each alternating digit. (Leave the last digit on the right alone since it is the check).
2. Add each individual digit comprising the products from step 1 to each of the unaffected digits.
3. If the product of step 2 ends in 0 (that is, can be divided by 10 with no remainder), the number is valid.

   For example, using 4111111111111111 (the common credit card test number):

   number: 4 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

   multiply: 2 2 2 2 2 2 2 2

   product: 8 2 2 2 2 2 2 2

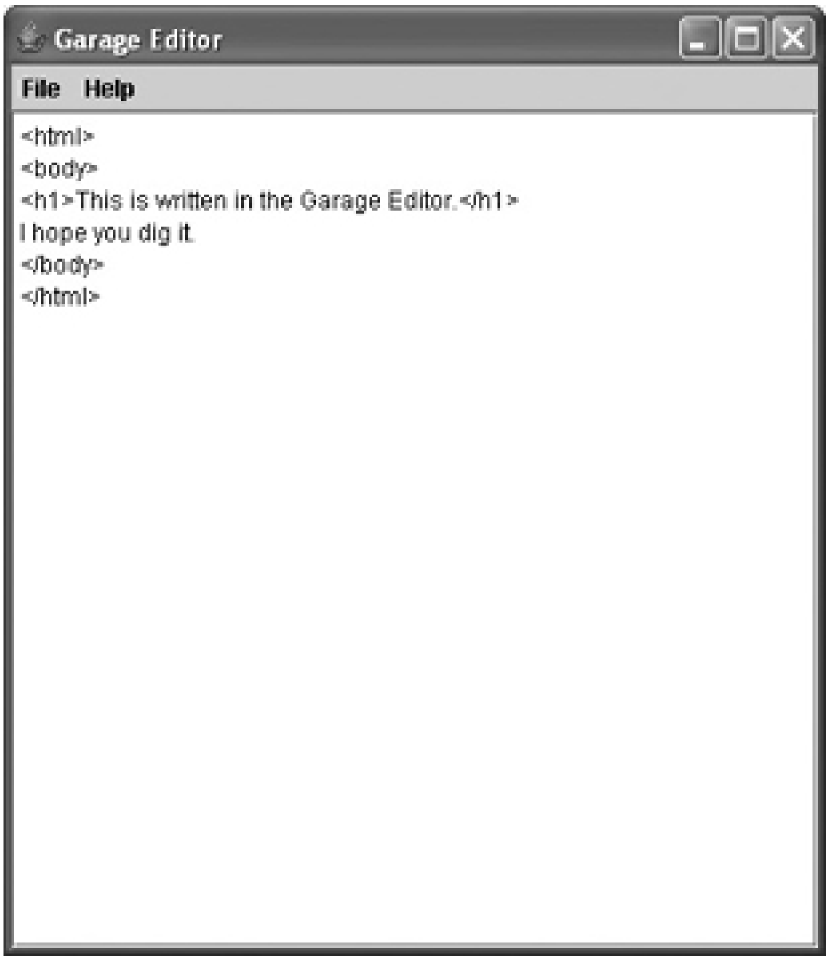
   Add them up (products are in parens):

   (8) + 1 + (2) + 1 + (2) + 1 + (2) + 1 + (2) + 1 + (2) + 1 + (2) + 1 + (2) + 1 = 30

   Divide by 10: 30/10 = 3. The number is good. (Because 30 MOD 10) = 0

Running the application and entering some text looks something like what you see in Figure 35.1.

Figure 35-1. The running text editor.



```
<html>
<body>
<h1>This is written in the Garage Editor.</h1>
I hope you dig it.
</body>
</html>
```

This looks like a lot of code, but there are also a lot of comments explaining what's going on.

GarageEditor.java

```
/**<p>
 * Presents a simple editor that allows
 * you to create a new file, open a file,
 * and save a file.
 * <p>
 * The chief benefit here is that you can see
 * how to interact with a meaningful program
 * without getting bogged down by layout manager
 * details.
 * <p>
 * Illustrates how to create a menu and menu items
 * <p>
 * Shows how to attach commands to be invoked when
 * an item is clicked.
 * <p>
 * Shows how to use common keystrokes for commands (ie,
 * press CTRL+S on the keyboard to Save the file).
 * <p>
 * Shows how to read files in and out for editing.
 * <p>
 * Shows how to use OK/Cancel dialogs
 * <p>
 * Shows how to use user preferences
 * </p>
 * Could benefit from threading to improve
 * responsiveness.
 * @author Eben Hewitt
 * @see JFrame
 * @see JTextArea
 * @see JOptionPane
 **/

import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.prefs.*;

public class GarageEditor extends JFrame {

   //holds user preferences
private Preferences prefs = null;

   //the user will type into this
   private JTextArea textArea;

   //lets you easily specify where you want
```

```
messages to go.
private static final int STDOUT = 0;
private static final int ALERT = 1;

//hold File menu commands
private static final String NEW_CMD = "New";
//default location at which to open window
private static final int DEF_WINDOW_X_LOCATION = 150;
private static final int DEF_WINDOW_Y_LOCATION = 150;

public GarageEditor() {
  // Add a menu bar and a JTextArea to the
  // window, and show it on the screen. The
  // first line of this routine calls the
  // constructor from the superclass to specify
  // a title for the window. The pack() command
  // sets the size of the window to
  // be just large enough to hold its contents.
      super("Garage Editor");

     //make a menu bar to hold menus
       JMenuBar menuBar = new JMenuBar();
     //add the File menu to it
       menuBar.add(makeFileMenu());
     //add the About menu to it
       menuBar.add(makeHelpMenu());

       //put the completed menu bar into
         the frame
        setJMenuBar(menuBar);

       //create a new JTextArea
       //give it some default text and set its size
        int rows = 25;
        int cols = 35;
       textArea = new JTextArea("choose
                 file...",
                 rows, cols);
     //white by default anyway, but highlights it
      textArea.setBackground(Color.WHITE);

      textArea.setMargin(new Insets(3,3,3,3));

       //make a set of scrolling controls to hold
       //the text area
       JScrollPane scroller =
                  new JScrollPane(textArea);

      //put the scroller into the content pane
      setContentPane(scroller);
      //stop app when user closes window
       setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
     pack();

   doLocation();
   //show the frame
   setVisible(true);
}

/**
* Sets up the location of the window using
* preferences data. Opens the editor in the
* last location that the user had it in. If
* not set, use a default value defined in
* class constants above.
*/
private void doLocation() {
  //get reference to preferences
  Preferences prefs = Preferences.userNodeForPackage(
  this.getClass());

  //retrieve the int. if there is no int stored
  //in prefs of the given string name, use the
  default
  int xLocation = prefs.getInt("WINDOW_X_LOCATION",
  DEF_WINDOW_X_LOCATION);
  int yLocation = prefs.getInt("WINDOW_Y_LOCATION",
  DEF_WINDOW_Y_LOCATION);

  //open the window at either the last
  //user location, or the default
  setLocation(xLocation,yLocation);
}

  /**
   * Makes the File menu
   * @return JMenu containing the New, Open, Save,
   * and Exit items
   */
  private JMenu makeFileMenu() {
     //get the action command from the menu
     //to determine what the user wants to do
     //and call a separate method to do the
     //dirty work
     ActionListener listener = new
             ActionListener() {
     public void actionPerformed(ActionEvent
             event) {
       String command = event.getActionCommand();
       if (command.equals("New")){
           doNew();
       }
       else if(command.equals("Open...")){
           doOpen();
       }
       else if(command.equals("Save...")){
           doSave();
       }
       else if(command.equals("Close")){
           doClose();
       }
       else if (command.equals("Exit")){
           doExit();
       }
     }
     };

   JMenu fileMenu = new JMenu("File");

      //NEW
   JMenuItem newCmd = new JMenuItem("New");
   newCmd.setAccelerator(KeyStroke.getKeyStroke
        ("ctrl N"));
   newCmd.addActionListener(listener);
   fileMenu.add(newCmd);

   //OPEN
   JMenuItem openCmd = new JMenuItem("Open...");
   openCmd.setAccelerator(KeyStroke
  .     getKeyStroke("ctrl O"));
   openCmd.addActionListener(listener);
   fileMenu.add(openCmd);

   //SAVE
   JMenuItem saveCmd = new JMenuItem("Save...");
   saveCmd.setAccelerator(KeyStroke
  .     getKeyStroke("ctrl S"));
   saveCmd.addActionListener(listener);
   fileMenu.add(saveCmd);

   //CLOSE
   JMenuItem closeCmd = new JMenuItem("Close");
   closeCmd.setAccelerator(KeyStroke
  .     getKeyStroke("ctrl L"));
   closeCmd.addActionListener(listener);
   fileMenu.add(closeCmd);

   //QUIT
   JMenuItem exitCmd = new JMenuItem("Exit");
   exitCmd.setAccelerator(KeyStroke
      .getKeyStroke("ctrl E"));
   exitCmd.addActionListener(listener);
   fileMenu.add(exitCmd);
```

```
        return fileMenu;
         }

      /**
       * Creates the About Menu
       * @return JMenu The Help menu containing the
       * About item.
       */
     private JMenu makeHelpMenu() {
        //get the action command from the menu
        //to determine what the user wants to do
        //and call a separate method to do the
        //dirty work
        ActionListener listener = new ActionListener() {
        public void actionPerformed(ActionEvent event) {
              String command = event.getActionCommand();
              if (command.equals("About"))
                        doAbout();
               }
         };
     JMenu helpMenu = new JMenu("Help");

       //ABOUT
     JMenuItem aboutCmd = new JMenuItem("About");
        aboutCmd.setAccelerator
                (KeyStroke.getKeyStroke("ctrl A"));
     aboutCmd.addActionListener(listener);
     helpMenu.add(aboutCmd);

     return helpMenu;
    }

    private void doAbout() {
     JOptionPane.showMessageDialog(this,
              "Java Garage Hardcore\nWhoever 2004");
     }

     private void doNew() {
```

```
 // Carry out the "New" command from the File menu
 // by clearing all the text from the JTextArea.
       textArea.setText("");
 }

private void doSave() {
   // Carry out the Save command by letting the user
   // specify an output file and writing the text
   // from the JTextArea to that file.
        File file; // The file that the user wants
                       to save.
   JFileChooser fd; // File dialog that lets the
                          //user specify the file.
        fd = new JFileChooser(new File("."));
        fd.setDialogTitle("Save As...");
        int action = fd.showSaveDialog(this);

        if (action != JFileChooser.APPROVE_OPTION) {
                //user cancelled, so quit the dialog
                return;
        }

        file = fd.getSelectedFile();
        if (file.exists()) {
                 //if file already exists, ask
                 before replacing it
                 action =
                 JOptionPane.showConfirmDialog(this,
                 "Replace existing file?");

                 if (action !=
                       JOptionPane.YES_OPTION)
                       return;
        }

        try {
           //Create a PrintWriter for writing to the
           //specified file and write the text from
           //the window to that stream.
```

```
            PrintWriter out = new PrintWriter(new
                        FileWriter(file));
            String contents = textArea.getText();
            out.print(contents);
            if (out.checkError())
                throw new IOException(
                   "Error while writing to file.");
                    out.close();
          }
          catch (IOException e) {
                // Some error has occurred while
                trying to write.
                // Show an error message.
                    JOptionPane.showMessageDialog(this,
                 "IO Exception:\n" +
                 e.getMessage());
          }
       }

        private void doOpen() {
           //open the file the user selected by
           //printing its contents to the text area

           //will hold the user's file
              File file;
              //creates the complete dialog the user
              //needs to select file
              JFileChooser fd;
              //use the current directory
              //(specified as ".")
              //as a starting place
              fd = new JFileChooser(new File("."));
              fd.setDialogTitle("Select a File...");
              int action = fd.showOpenDialog(this);

              if (action != JFileChooser.APPROVE_OPTION) {
                      return;
              }
```

```
    //set the file to what the user selected
    file = fd.getSelectedFile();
    try {
        //reset the text area to be blank
      textArea.setText("");
        //read in the selected file
      BufferedReader in = new BufferedReader(
                  new FileReader(file));
      String lines = "";

      int lineCt = 0;
      String str;

      while ((str = in.readLine()) != null) {
          lines += str + "\n";
      }

      textArea.setText(lines);
      in.close();

  } catch (Exception e) {
      JOptionPane.showMessageDialog(this,
      "Unable to open file:\n" + e.getMessage());
        }
}

private void doClose(){
      Object[] options = { "OK", "CANCEL" };
      int action =
           JOptionPane.showOptionDialog(null,
                "OK to close without
                saving?", "Closing",
                JOptionPane.DEFAULT_OPTION,
                JOptionPane.WARNING_MESSAGE,
                null, options, options[0]);

      if (action == JOptionPane.OK_OPTION) {
          //reset the text area
```

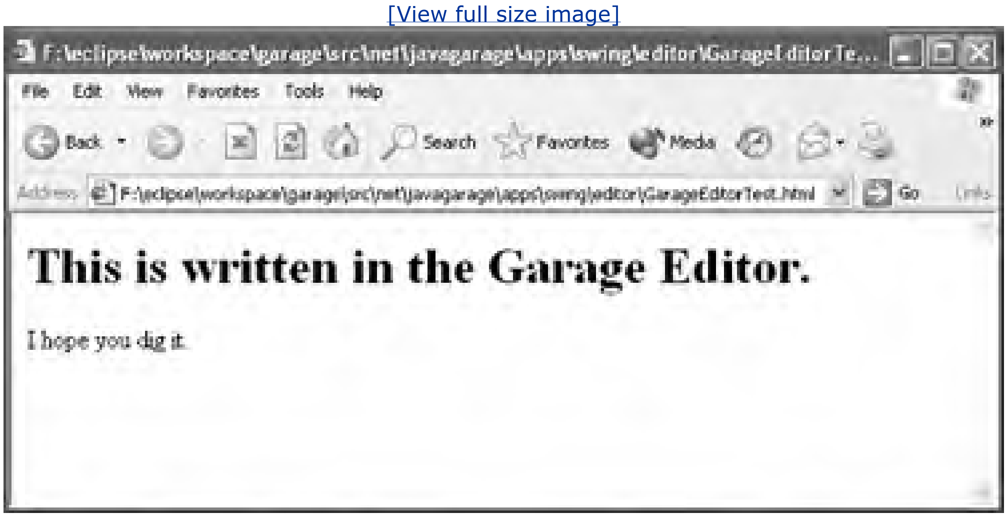
```
                textArea.setText("");
                return;
            } else {
                //do nothing and leave their
                file alone
                return;
            }
    }

  /**
   * Exit the application and stop the VM.
   */
  private void doExit() {
      //store the current location of the window
      //as a user preference so we can open it here
      //next time
      Preferences prefs =
          Preferences.userNodeForPackage
          (this.getClass());
      prefs.putInt("WINDOW_X_LOCATION", getX());
      prefs.putInt("WINDOW_Y_LOCATION", getY());
      // stop the application
      System.exit(0);
  }

    /**
     * Allows you to easily switch logging locations.
     * You could add a FILE case here too for moving
     * into production.
     * @param msg
     * @param type
     */
    private void log(String msg, int type){
        switch (type){
        default : //fall through
        case STDOUT:
        System.out.println("—>" + msg);
        break;
        case ALERT:
            JOptionPane.showMessageDialog(this, msg);
            break;
            }
        }

    public static void main(String[] args) {
        //start the application
            new GarageEditor();
    }
}
```

Results

We can enter some text, and then save the file just as you'd expect. Let's do that and then open it in a browser to prove that it works (see Figure 35.2).

Figure 35.2. Entering text into the editor is a breeze....

[View full size image]

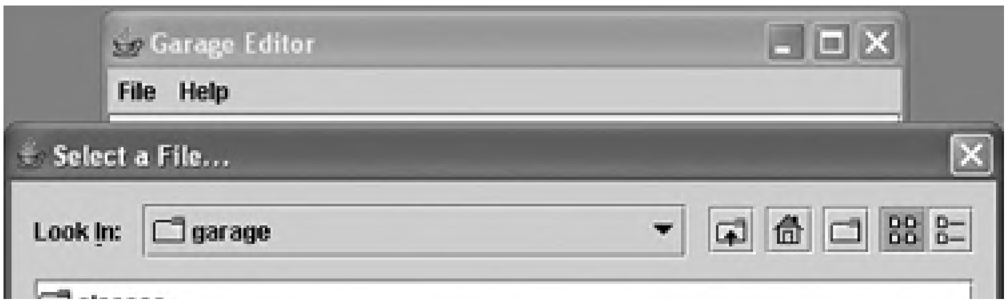

F:\eclipse\workspace\garage\src\net\javagarage\apps\swing\editor\GarageEditorTe...
File Edit View Favorites Tools Help
Back Search Favorites Media
Address F:\eclipse\workspace\garage\src\net\javagarage\apps\swing\editor\GarageEditorTest.html
Go Links

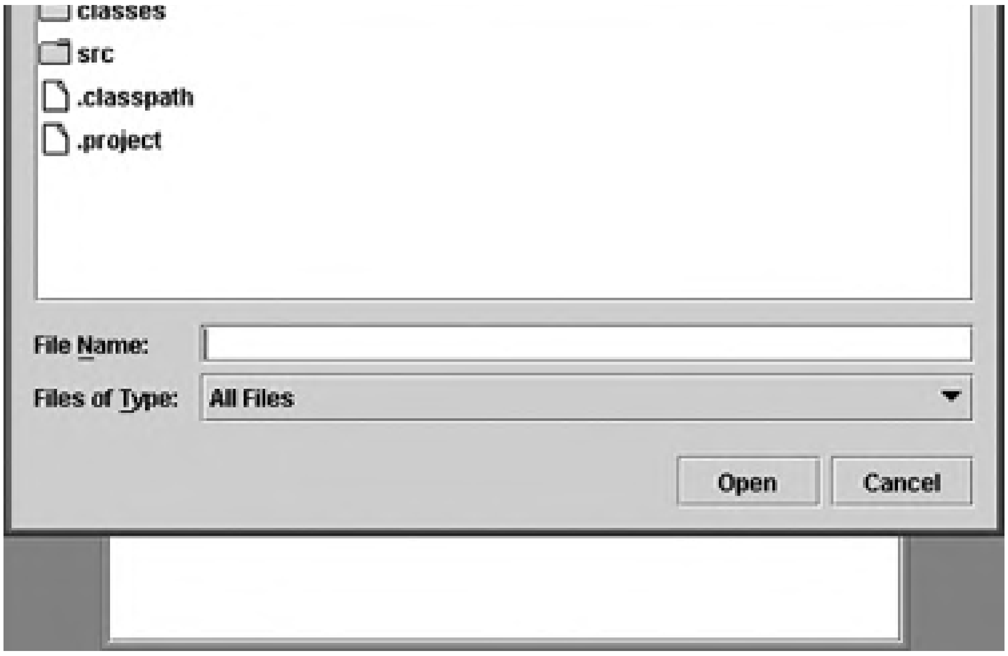
This is written in the Garage Editor.

I hope you dig it.

We also use the file chooser to browse to a local file (see Figure 35.3).

Figure 35.3. Using the JFileChooser.

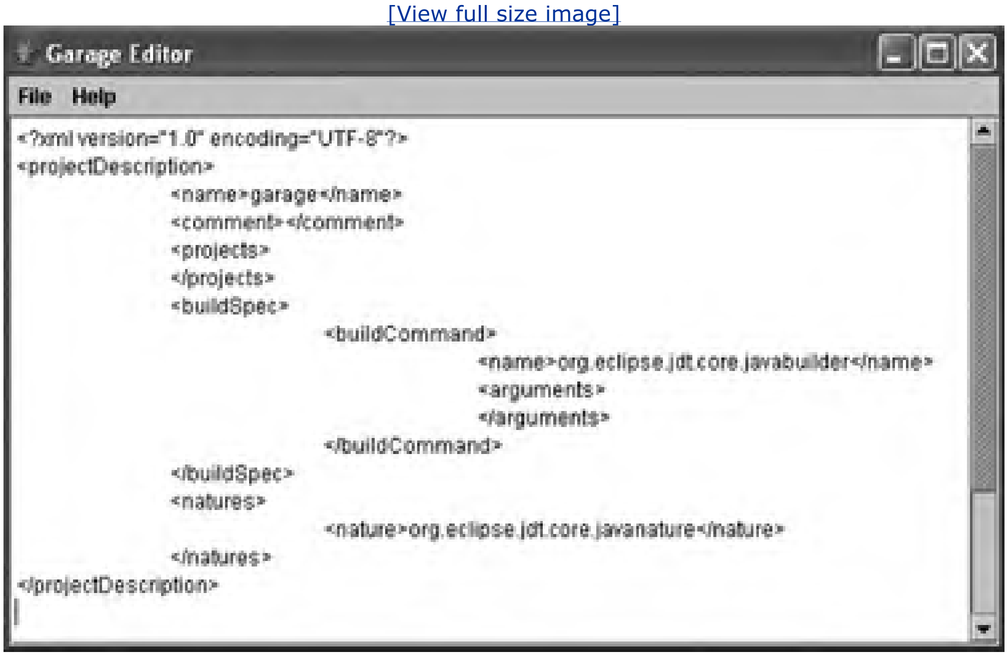




If we open an existing file written in XML, such as the Eclipse project file, we see that the program honors the line breaks, tabs, and so forth that were already present in the file (see Figure 35.4).

Figure 35.4. The editor honors formatting.

[View full size image]

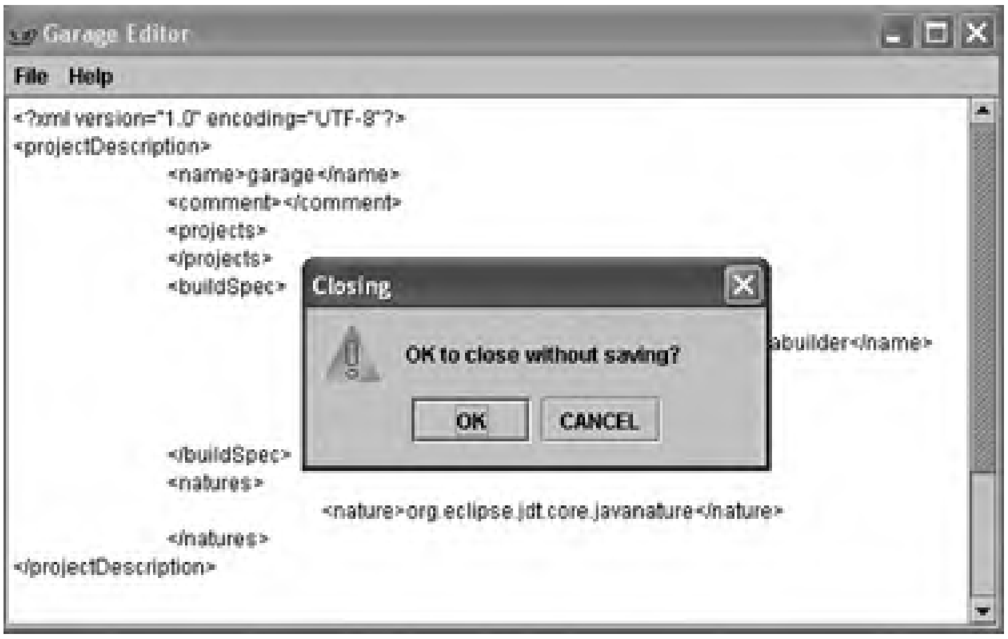


```
<?xml version="1.0" encoding="UTF-8"?>
<projectDescription>
		<name>garage</name>
		<comment></comment>
		<projects>
		</projects>
		<buildSpec>
				<buildCommand>
						<name>org.eclipse.jdt.core.javabuilder</name>
						<arguments>
						</arguments>
				</buildCommand>
		</buildSpec>
		<natures>
				<nature>org.eclipse.jdt.core.javanature</nature>
		</natures>
</projectDescription>
```

If you choose to close an open document without saving, the app asks if you're sure you know what you're doing. If you choose OK, changes are not saved; if you choose Cancel, you are returned to the editor (see Figure 35.5).

Figure 35.5. The JOptionPane at work.

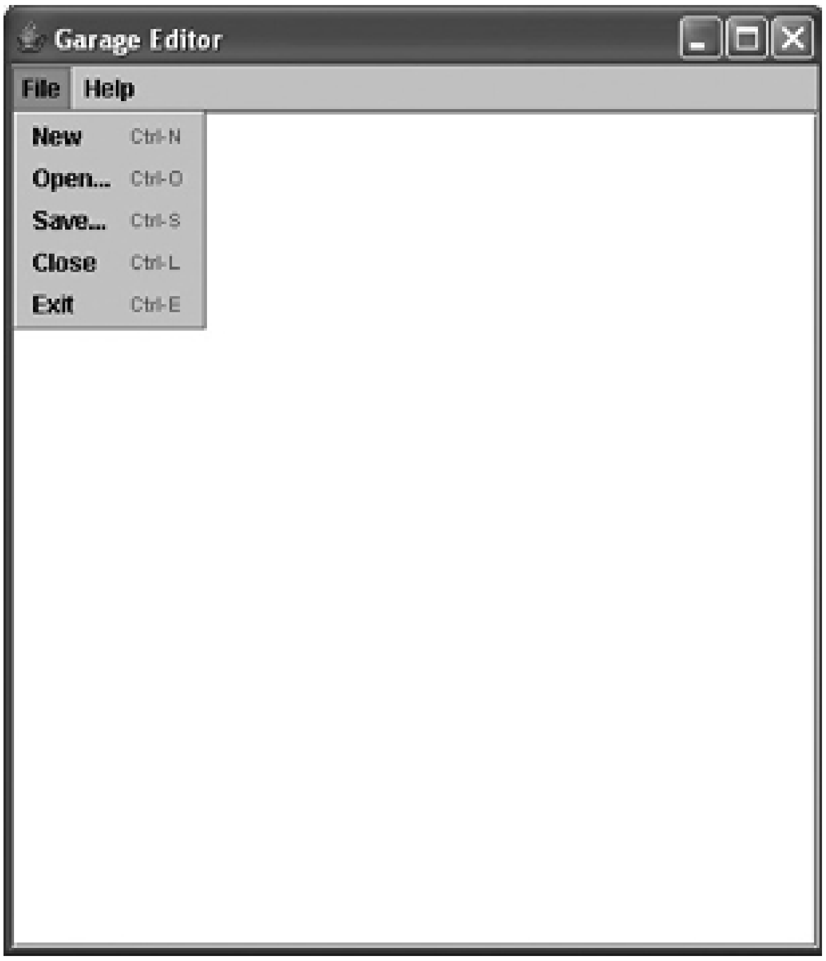
[View full size image]



```
<?xml version="1.0" encoding="UTF-8"?>
<projectDescription>
            <name>garage</name>
            <comment></comment>
            <projects>
            </projects>
            <buildSpec>
                                                  abuilder</name>
            </buildSpec>
            <natures>
                        <nature>org.eclipse.jdt.core.javanature</nature>
            </natures>
</projectDescription>
```

When you perform an action on the editor, you see the toolbar containing the File and Help menus. Choose a menu, and you see its items (see Figure 35.6).

Figure 35.6. The File menu and its items, including the mnemonics.



Application: RSS Aggregator

RSS is an acronym for Rich Site Summary (or, some will tell you, Really Simple Syndication). It is an XML-based technology for syndicating the content of news and blog Web sites. A program that is aware of RSS is called an aggregator. If you see a little orange XML icon in a Web page, it means that that content is available in an RSS feed.

In this episode, we'll write an RSS aggregator. It will pull together many of the technologies we have discussed, including GUI apps, and introduce some advanced topics we haven't discussed, such as networking, Collections, and caching.

Before we talk about what we'll write, let's figure out a bit more about RSS.

RSS is an XML format that is largely concerned with defining titles, news items, content, images, descriptions, and other structural information that news articles conform to.

There are many different versions of RSS. The first was .90, which was quickly obsolete. It was followed by version .91, wonderfully simple and straightforward to use, which makes it very popular for basic data. Version .92 allows for more metadata than previous versions. The common element between these versions of RSS is that they are each controlled by a single vendor (Netscape in the case of .90, and UserLand for the others). Version 1.0, however, is controlled by the RSS development working group. The current version is 2.0, and offers support for extensibility modules, a stable core, and active development. Note that this version is upwards compatible—that is, any valid 91 source is also a valid 2.0 feed.

The point in considering these different versions is that they are quite different indeed, and you are likely to encounter the many different versions on your travels. The application we write will allow you to create an XML file that defines the URLs of the RSS feeds you are interested in.

Here is what version .91 looks like:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<rss version="0.91">

<channel>

<title>WriteTheWeb</title>

<link>http://writetheweb.com</link>

<description>News for web users that write

     back</description>

<language>en-us</language>

<copyright>Copyright 2000, WriteTheWeb team.

     </copyright>

<managingEditor>editor@writetheweb.com</managingEditor>

<webMaster>webmaster@writetheweb.com</webMaster>

<image>

<title>WriteTheWeb</title>

<url>http://writetheweb.com/images/mynetscape88.gif</url>

<link>http://writetheweb.com</link>

<width>88</width>

<height>31</height>

<description>News for web users that write
```

```
      back</description>
</image>
<item>
<title>Giving the world a pluggable Gnutella</title>
<link>http://writetheweb.com/read.php?item=24</link>
<description>WorldOS is a framework on
which to build programs
that work like Freenet.</description>
</item>
</channel>
</rss>
```

Here is a version of RSS 2.0:

```
<?xml version="1.0" ?>
<!— RSS generated by UserLand Frontier v9.0 on 11/30/2003;
7:13:00 PM Eastern —>
<rss version="2.0">
<channel>
  <title>Scripting News</title>
  <link>http://www.scripting.com/</link>
  <description>All scripting, all the time, forever.</
description>
   <language>en-us</language>
   <copyright>Copyright 1997-2003 Dave Winer</copyright>
   <pubDate>Sun, 30 Nov 2003 05:00:00 GMT</pubDate>
   <lastBuildDate>Mon, 01 Dec 2003 00:13:00
          GMT</lastBuildDate>
   <docs>http://blogs.law.harvard.edu/tech/rss</docs>
   <generator>UserLand Frontier v9.0</generator>
   <managingEditor>dwiner@cyber.law.harvard.edu</
```

```
managingEditor>
   <webMaster>dwiner@cyber.law.harvard.edu</webMaster>
<item>
   <description><img
src="http://monster2.scripting.com/z/
images/archiveScriptingCom/2003/11/30/xmasTree.gif"
width="44"
height="66" border="0" align="right" hspace="15"
vspace="5"
alt="A picture named xmasTree.gif">Thanks to
Vitamin C, <a href="http://www.herbs.org/greenpapers/
echinacea.html">echinacea</a> and lots of rest, my first cold
of the winter appears to be over.</description>
   <pubDate>Mon, 01 Dec 2003 00:05:11 GMT</pubDate>
   <guid>http://archive.scripting.com/2003/11/
30#When:7:05:11PM</guid>
   <category>/Boston/Weather</category>
   </item>
</channel>
</rss>
```

This version is obviously more verbose, but you see that the basic elements—channel, item, image—are intact. What is added is extra metadata, such as publication date, last build date, and so on.

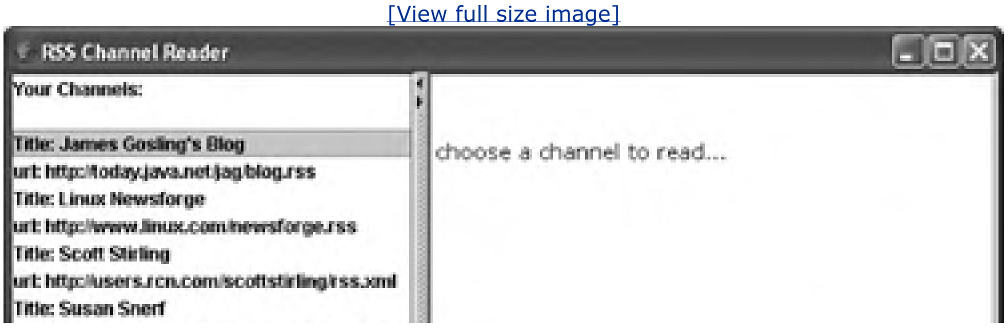
In our application, let's create an XML document that indicates the URLs of the RSS news feeds we want to view. Then, our Java app will display this list of links by reading in this XML document, and fetch the RSS data when we click on the link.

However, because it is native XML, it is not very readable in this format, and wasn't meant to be viewed this way. So, we use a basic XSL stylesheet that accounts for many of the common RSS tags in order to transform the source into HTML.

Our completed app will look like Figure 35.7 when we launch it.Click the URL on the left side. The app will open a network connection and retrieve the data at the other end, transform the data using the XSL stylesheet, and save the resulting HTML to a cache. Every subsequent time in this session that you click on that URL, the app retrieves the content from the cache.

Figure 35.7. The RSS Aggregator app displays channels to choose from for news reading.

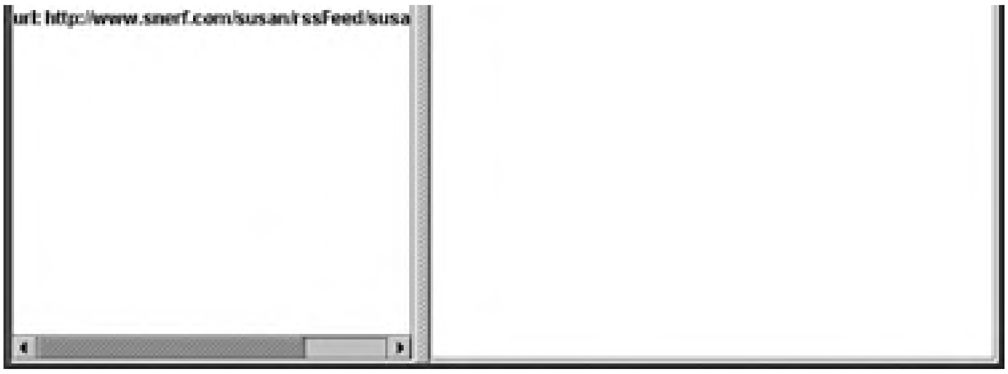
[View full size image]



Your Channels:

Title: James Gosling's Blog
url: http://today.java.net/jag/blog.rss
Title: Linux Newsforge
url: http://www.linux.com/newsforge.rss
Title: Scott Stirling
url: http://users.rcn.com/scottstirling/rss.xml
Title: Susan Snerf

choose a channel to read...

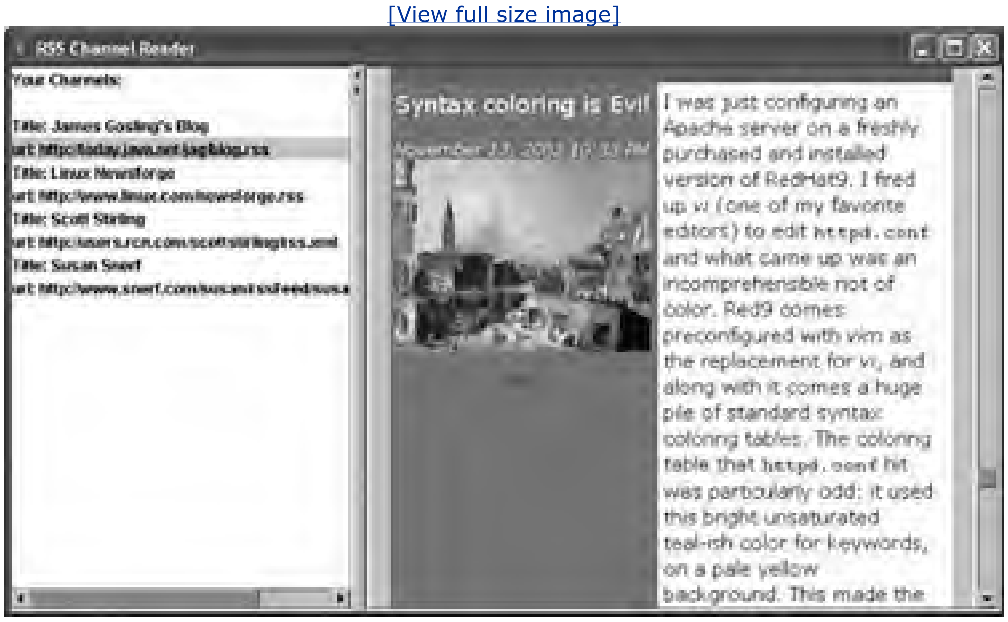


url: http://www.snerf.com/susan/rssFeed/susa

If the document contains a link, you can click on the link and the app will act as a browser, retrieve that content, and display it (see Figure 35.8). Note that to save space, the app does not check the version of the RSS document and do anything special based on that info, nor is the stylesheet anything more than rudimentary. For those reasons, most RSS feeds you throw at it should display, but they will likely display with varying degrees of beauty.

Figure 35.8. Browsing to a complete entry in James Gosling's blog.

[View full size image]



Your Channels:

Title: James Gosling's Blog
url: http://today.java.net/jag/blog.rss
Title: Linux Newsforge
url: http://www.linux.com/newsforge.rss
Title: Scott Stirling
url: http://users.rcn.com/scottstirling/rss.xml
Title: Susan Snerf
url: http://www.snerf.com/susan/rssFeed/susa

Syntax coloring is Evil

November 13, 2003 10:33 PM

I was just configuring an Apache server on a freshly purchased and installed version of RedHat9. I fired up vi (one of my favorite editors) to edit httpd.conf and what came up was an incomprehensible riot of color. Red9 comes preconfigured with vim as the replacement for vi, and along with it comes a huge pile of standard syntax coloring tables. The coloring table that httpd.conf hit was particularly odd: it used this bright unsaturated teal-ish color for keywords, on a pale yellow background. This made the

If you click on a title, your browser will clear. As mentioned earlier, clicking on the same title a second time in the same session retrieves data from the cache. The program's output to the standard out illustrates this:

```
F:\eclipse\workspace\garage\src\apps\rss\source\channels.xml

File

F:\eclipse\workspace\garage\src\apps\rss\source\cache\blog.rss

not in cache

F:\eclipse\workspace\garage\src\apps\rss\source\cache\blog.rss

Getting file
```

```
F:\eclipse\workspace\garage\src\apps\rss\source\cache\blog.rss

from in cache
```

Let's look at the files that make up the application in order of appearance.

First, NewsReader.java contains the main method. It starts the application by creating a new ReaderFrame and displaying it.

NewsReader.java

```
package net.javagarage.apps.rss;


/**

 * Starts the application by displaying the ReaderFrame.

 * @author eben hewitt

 * @see ReaderFrame

 */

public class NewsReader {


public static void main(String[] args) {

    start();

}


public static void start() {

    ReaderFrame frame = new ReaderFrame("RSS Channel

            Reader");

    frame.pack();

    frame.setVisible(true);

}

}
```

It's pretty straightforward. Because the main job of NewsReader.java is to create a ReaderFrame, let's take a look at his job.

ReaderFrame.java

```
package net.javagarage.apps.rss;

import java.awt.Dimension;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.Map;
import java.util.StringTokenizer;
import java.util.Vector;

import javax.swing.JEditorPane;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JList;
import javax.swing.JScrollPane;
import javax.swing.JSplitPane;
import javax.swing.ListSelectionModel;
import javax.swing.event.HyperlinkEvent;
import javax.swing.event.HyperlinkListener;
import javax.swing.event.ListSelectionEvent;
import javax.swing.event.ListSelectionListener;

/**
 * <p>
 * The frame that holds the content of the app.
 * It encapsulates the necessary JFrame code and
 * gathers the content.
 * Note in particular the getRSSData() method, which
 * demonstrates a pretty good use of caching. The
 * data is retrieved only when the user
 * requests it, and it is fresh the first time, but
 * subsequent reads of the same file in the same
 * session are read from the cache,
 *  which stores the transformed result.
 * </p>
 * <p>
 * Created: Sep 12, 2003 4:38:09 PM
 * <p>
 *
 * @author HewittE
 * @since 1.2
 * @version 1.0
 * @see
 */
public class ReaderFrame extends JFrame implements
ListSelectionListener, HyperlinkListener {
private Vector imageNames;
   private JLabel channelLabel = new JLabel();
private JEditorPane contentPane;
   private JList list;
   private JSplitPane splitPane;
   private String root = RSSKeys.RESOURCE_ROOT;

   /**
    * Constructor. See inline comments
    * @param title The app name that appears in the
    * window's upper left corner.
    */
public ReaderFrame(String title) {
super(title);
initFrame();

        //make the channel list
        list = new JList(getChannelList());
        //allow display of only one item at a time

list.setSelectionMode(ListSelectionModel.
                  SINGLE_SELECTION);
        //select the first item in the list
        //when the app starts
        list.setSelectedIndex(2);
        list.addListSelectionListener(this);
        //to hold list of available channels
        JScrollPane listScrollPane =
                   new JScrollPane(list);
try {
//create the right-hand side to show the transformed HTML
contentPane = new JEditorPane("file:///" +
RSSKeys.RESOURCE_ROOT + "start.html");
} catch (IOException ioe) {
System.out.println(ioe.getMessage());
}
//to hold the contents of selected channel
contentPane.setEditable(false);
contentPane.addHyperlinkListener(this);
JScrollPane channelScrollPane = new JScrollPane(contentPane);

        //make a split pane with two scroll panes
        splitPane = new
JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
                                  listScrollPane,
channelScrollPane);
        //allows the divider to be toggled on or off
        //with one click
        splitPane.setOneTouchExpandable(true);
/*set minimum sizes for the two components
in the split pane.
*/
Dimension minimumSize = new Dimension(300, 300);
splitPane.setMinimumSize(minimumSize);

//set preferred size for the split pane.
splitPane.setPreferredSize(new Dimension(650, 400));
//set the divider at the end of the channel list pane
```

```
splitPane.setDividerLocation(250);
//put the content into the frame
getContentPane().add(this.getSplitPane());
}

/**
 * do typical things to set up the window
 *
 */
public void initFrame(){
setDefaultLookAndFeelDecorated(true);
//shut down the app when the window is closed
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
//put window somewhere
setLocation(300,300);
}

/**
 * This method is separate from the parseList in
 * order to help keep the implementation-specific
 * details to a minimum in how we populate the list.
 * That is, we need a Vector of Strings.
 * @return Vector The list of channels as
 * user-friendly text.
 */
public Vector getChannelList() {
String channelList = XMLReader.transform(RSSKeys.LIST_XML,
RSSKeys.LIST_XSL);
return parseList(channelList);
}

/**
 * @param s
 * @return Vector The list items as parsed from the
 * channels.xml document
 */
protected static Vector parseList(String s) {
//default size of list will be 6
```

```
Vector channels = new Vector(6);
channels.add("Your Channels: ");
StringTokenizer tokenizer = new StringTokenizer(s, "\n");
while (tokenizer.hasMoreTokens() ) {
String channelName = tokenizer.nextToken();
channels.addElement(channelName);
}
return channels;
}

/**
 * Gets the split pane that holds the two
 * JScroll panes
 * @return JSplitPane The main frame
 */
public JSplitPane getSplitPane() {
  return splitPane;
}

/**
 * We are implementing the ListSelectionListener,
 * which means that we promise to do something when
 * the user clicks an item in the list.
 * <br>What we do is either display a clever message
 * if the user has clicked a channel name instead of
 * a URL. In the lucky event that the user clicks
 * that channel's URL instead, we pass that string
 * into the getRSSData method in order to get the
 * viewable page back.
 */
public void valueChanged(ListSelectionEvent evt) {
if (evt.getValueIsAdjusting())
   return;

//setup cache to hold transformed files
```

```
Map cache = Cache.getInstance().getCache();



JList theList = (JList)evt.getSource();
String content = "No channel selected";
String rssSite = "";
if (theList.isSelectionEmpty()) {
channelLabel.setText("Selection empty");
} else {
rssSite = (String)theList.getSelectedValue();
if (rssSite.startsWith("url:")) {
rssSite = rssSite.substring(4);
contentPane.setText(getRSSData(rssSite));
} else {
contentPane.setText("<code>");
}
}
  channelLabel.revalidate();
  }

  /**
   * The main workhorse of the app. Gets the url
   * passed in and gets its content from
   * the Internet the first time.
   * After that data has been read
   * once (per session) subsequent views are read
   * from the cache.
   *
   * @param rssSite The internet address of the XML
   * RSS feed that you want to read
   * @return The RSS transformed into HTML, which is
   * viewable in the JEditorPane
   * @see Cache
   */

  public String getRSSData(String rssSite) {
  String data;
  String result = "";
```

```
   try {
URL url = new URL(rssSite);
String xsl = RSSKeys.RSS_XSL;

//Note that the cache is a singleton
Map cache = Cache.getInstance().getCache();

BufferedReader in = new BufferedReader(new
InputStreamReader(url.openStream()));

//get only the file name, without any of the path
String fileName = url.getFile();
fileName = fileName.substring(fileName.lastIndexOf("/") + 1);

String savedXML = RSSKeys.CACHE + fileName;

if (cache.get(savedXML) == null) {
System.out.println("File " + savedXML +
       " not in cache");
PrintWriter out = new PrintWriter(new
       BufferedWriter(new
FileWriter(savedXML, false)));
//get data from URL
while((data = in.readLine())!= null) {
out.write(data);
}
out.close();
in.close();
//transform the file into HTML using XSL
result = XMLReader.transform(savedXML, xsl);
//cache the data
cache.put(savedXML, result);
} else {
System.out.println("Getting file " + savedXML +
        " from in cache");
result = (String) cache.get(savedXML);
}
```

```
   } catch (MalformedURLException mfe){
   System.out.println(mfe.getMessage());
   } catch (IOException ioe){
System.out.println(ioe.getMessage());
}

   return result;
   }
   /**
    * Receives a hyperlink event when user clicks on
    * a link that might be in one of the blog pages
    * and sets the page to that URL.
    */
   public void hyperlinkUpdate(HyperlinkEvent e) {
   URL url=e.getURL();

   if ((e.getEventType() ==
HyperlinkEvent.EventType.ACTIVATED) &&
   (url.toString().startsWith("http"))) {
try {
contentPane.setPage(url);
} catch (Exception ex) {
System.out.println(ex.getMessage());
}
   }
   }
}
```

As you can see, ReaderFrame does the main work of the application. It relies on a number of other documents. The first is channels.xml, and it is our own custom XML file that defines the names and locations of the channels that we want to view. Obviously, you can change these values to ones that suit you better.

channels.xml

```
<?xml version="1.0"?>

<channel-list>

<channel>

<title>James Gosling's Blog</title>

<url>http://today.java.net/jag/blog.rss</url>

</channel>

<channel>

<title>Linux Newsforge</title>

<url>http://www.linux.com/newsforge.rss</url>

</channel>

<channel>

<title>Scott Stirling</title>

<url>http://users.rcn.com/scottstirling/rss.xml</url>

</channel>

<channel>

<title>Susan Snerf</title>

<url>http://www.snerf.com/susan/rssFeed/

susansnerfrss.xml</url>

</channel>

</channel-list>
```

This stylesheet is used to transform the contents of the channel list.

channels.xsl

```
<?xml version="1.0"?>

<xsl:stylesheet

xmlns:xsl="http://www.w3.org/1999/XSL/

Transform" version="1.0">

<xsl:output method="html" indent="yes"/>

<xsl:strip-space elements="*"/>

<xsl:template match="channel-list/channel/title">

Title: <xsl:apply-templates/> <xsl:value-of

select="title"/>

</xsl:template>

<xsl:template match="channel-list/channel/url">
```

```
<xsl:template match="channel-list/channel/url">
url: <xsl:apply-templates/><xsl:value-of select="url"/>
</xsl:template>
</xsl:stylesheet>
```

But how do we get the opening, default page? The JEditorPane reads in the start.html page, which looks like this.

start.html

```
<html>
<body>
<br>
<br>
<font face="Verdana">choose a channel to read...</font>
</body>
</html>
```

The file RSSKeys.java contains nothing other than String constants used by the application. This is an easy way here to use declarative style programming, where we don't store data that the app relies on (but which is specific to the environment) inside the program. This way, it is easy to change these values. Note that you probably want to do this business in a Properties file.

RSSKeys.java

```
package net.javagarage.apps.rss;
/**
 * @author eben hewitt
 * Holds constants that different classes need.
 * Note the naming convention of all uppercase and
 * underscores in between words for constants.
 */
public class RSSKeys {
```

```
public static final String RESOURCE_ROOT =

"F:\\eclipse\\workspace\\garage\\src\\apps\\rss\\

    source\\";

public static final String CACHE = RESOURCE_ROOT +

    "cache\\";

public static final String LIST_XML = RESOURCE_ROOT +

"channels.xml";

public static final String LIST_XSL = RESOURCE_ROOT +

"channels.xsl";

public static final String RSS_XSL = RESOURCE_ROOT +

"rssStyle.xsl";

}
```

Remember that on Windows, we need to escape the backslash character that is used as the file path separator.

The next important file is the XMLReader.java class. Its job is to read in the specified document as an input stream and transform it using the stylesheet. The XML document URL comes from the event generated in the GUI list (which itself was generated from the XML channels file), and the XSL stylesheet location is in the RSSKeys class.

XMLReader.java

```
package net.javagarage.apps.rss;


import java.io.FileInputStream;

import java.io.IOException;

import java.io.InputStream;

import java.net.MalformedURLException;

import java.net.URL;

import javax.xml.transform.Templates;

import javax.xml.transform.Transformer;

import javax.xml.transform.stream.StreamSource;

import javax.xml.transform.stream.StreamResult;

import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerFactory;

import java.io.StringWriter;

/**
 * <p>
 * Reads XML documents and transforms them using XSLT.
 * </p>
 * @author eben hewitt
 *
 */
public class XMLReader {

public static InputStream
        getDocumentAsInputStream(URL url)
throws IOException {
    InputStream in = url.openStream();
    return in;
  }

public static InputStream
getDocumentAsInputStream(String url)
    throws MalformedURLException, IOException {
    URL u = new URL(url);
    return getDocumentAsInputStream(u);
  }

  public static String transform(String xml,
          String xsl) {
  System.out.println(xml);
       TransformerFactory tfactory =
TransformerFactory.newInstance();
       try {

            // compile the stylesheet
            Templates templates =
tfactory.newTemplates( new
StreamSource(
            new FileInputStream(xsl)));
            StringWriter sos = new StringWriter();
            StreamResult out = new StreamResult( sos );

            Transformer transformer =
templates.newTransformer();
      transformer.transform(new StreamSource(xml), out);
            sos.close();
            String result = sos.toString();
            return result;
      }
      catch ( IOException ex ) {
          System.out.println(ex.getMessage());
      }
      catch ( TransformerException ex ) {
          System.out.println( ex.getMessage() );
      }
      return null;
   }
}
```

Here is the XSL stylesheet used to transform the content read in from a channel into pretty HTML.

rssStyle.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/
Transform" version="1.0">
<xsl:output method="html" indent="yes"/>

<xsl:strip-space elements="*"/>

<xsl:template match="rss">
    <table border="1">
    <xsl:apply-templates/>

    </table>

</xsl:template>


<xsl:template match="rss/channel">

<tr>

<td>

<a>

    <xsl:attribute name="href">




<xsl:value-of select="link"/>

</xsl:attribute>




<b>




<xsl:value-of select="title"/>




</b>

    </a>

</td>

</tr>

<xsl:apply-templates/>

</xsl:template>

<xsl:template match="rss/channel/link">

</xsl:template>


<xsl:template match="rss/channel/title">

</xsl:template>


<xsl:template match="rss/channel/description">

</xsl:template>


<xsl:template match="rss/channel/language">

</xsl:template>
```

```
<xsl:template match="rss/channel/copyright">
</xsl:template>

<xsl:template match="rss/channel/managingEditor">
</xsl:template>

<xsl:template match="rss/channel/webMaster">
</xsl:template>

<xsl:template match="rss/channel/image">
</xsl:template>

<xsl:template match="rss/channel/item">
<tr>
  <td>
  <a>


<xsl:attribute name="href">


<xsl:value-of select="link"/>


</xsl:attribute>


<b>


<xsl:value-of select="title"/>


</b>
  </a>
  </td>
  <td>
```

```
    <xsl:value-of select="description"/>

    </td>

</tr>

</xsl:template>


</xsl:stylesheet>
```

The Cache class here holds the generated content from previous visits to URLs. This is a fairly common way to implement a cache, and has the added benefit of demonstrating the Singleton design pattern.

Cache.java

```
package net.javagarage.apps.rss;


import java.util.HashMap;

import java.util.Map;

/**

 * @author eben hewitt

 * Singleton. Which means there can only be one

 * instance of this class per JVM.

 *

 * Holds a HashMap collection, which

 * allows easy retrieval of object values based on a

 * key name. So we can pass in the URL string of a

 * file name along with the content of the

 *  file, and get the file out later. The obvious

 * benefit is speed, also, singleton is a pretty

 * importantpattern and one that is used frequently

 * in a manner similar to this.

 */

public class Cache {

private Map cache = new HashMap();

private static Cache instance = new Cache();
private Cache() {}

protected Map getCache() {
return cache;
}
protected static Cache getInstance() {
return instance;
}
}
```

Application: DrawingPad: Garage Doodler

This Swing GUI application is comprised of two separate but related bits of functionality. The main thing it does is it allows the user to draw freehand on a canvas, and then captures and saves the doodle as an image file. To get this functionality, we use the ImageIO class, available since SDK 1.4. The second thing this app does is it allows you to open an image file from your hard drive and view it in the frame.

This application is worth looking over. It may seem long at first, but there are a lot of comments in this sucker. I think that Swing apps can be fairly confusing for a number of reasons. One reason is that Sun has frequently updated the APIs that are used to make GUIs. That means that you often see several different ways of doing effectively the same thing. Some of those ways may be more recent or efficient or stable, however. So you've got to be careful.

There are a lot of things that have to happen to make a Java GUI app go. If you have used Microsoft Visual Studio even for a few minutes, you can see that you can create a functional window in under a minute without writing a line of code. In Java, you have to do a lot of the writing yourself, and some components may not behave as expected. So I have commented just about every line of the application. Remember that the aim of the toolkit here is to give you something that works, so you can see how all of the pieces fit together, and something that you can build on if you want to.

Demonstrates

This application demonstrates a number of cool things, and a number of things that are important to doing real Java development. The following Swing classes are used: JFrame, JPanel, JComponent, JOptionPane, and JFileChooser. We also incorporate many classes from the older AWT package to handle coordinates, graphics, colors, events, and layout. These classes include Rectangle, Graphics, Point, Dimension, and Color. The application also demonstrates how to change the cursor from an arrow to a crosshair, and how to use keyboard shortcuts on your menus.

We see how to implement a menu with commonly required menu items that act differently than we have worked on previously. We see how to use the ImageIO class to read and write image data, and we subclass the FileFilter class to make sure that our JFileChooser only allows image file types to be opened. Use of JFileChooser to save an image is also presented.

This all means that we have to deal with mouse motion events, event handlers and listeners, see how anonymous classes are used, and how to do good subclassing. By using the JOptionPane, we show any exceptions to the user so that he can call your direct line to tell you exactly what the problem is; this is much better than burying the exception in an `out.println` statement.

Limitations and Extension Points

FRIDGE

Remember that one thing we're trying to do is make the code do the talking. That's why it's okay with me to have all of this code. I want the emphasis to be on the code itself, and let it do most of the communicating when possible. There are too many technical books that explain 20 things by showing you the API and then give you a lot of disconnected snippets of code; that often leaves you not knowing what to do when you sit down at the keyboard. It also makes it hard to integrate into a real app. This method is a key tenet in *Extreme Programming*, popularized by Kent Beck, which is an added bonus.

The application does not allow you to do a few things. First, you cannot change the background color of the pad, and you cannot change the color of the pen you use to draw with. This functionality can be added with some ease by looking into the JColorChooser API. This guy works somewhat like the JFileChooser, which should make it feel familiar after using the JFileChooser in this app. The JColorChooser provides a very rich set of controls that allow the user to pick a color using RGB values, an eyedropper, and more.

Another good extension point for this application would be to allow the user to choose a different line thickness. A simple way to do this would be to show a series of buttons on the tool bar that create the pen with the specified thickness. The way to do this is to create your desired thickness with a float value, and then call the `setStroke()`method on the Graphics2D object. That code would look something like this:

```
Graphics2D g2d = (Graphics2D)g;

float thickness = 5.0f;


// This is solid stroke, but you can also make dashed

BasicStroke stroke = new BasicStroke(thickness);

g2d.setStroke(stroke);
```

For a more user-friendly but complicated implementation, you could allow the user to choose the thickness with a JSlider control.

DrawingApp.java

```
package net.javagarage.apps.swing.draw;


import java.awt.BorderLayout;

import java.awt.Color;

import java.awt.Cursor;

import java.awt.Dimension;

import java.awt.Graphics;

import java.awt.Image;

import java.awt.Point;

import java.awt.Rectangle;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import java.awt.event.MouseAdapter;

import java.awt.event.MouseEvent;

import java.awt.event.MouseMotionAdapter;

import java.awt.image.BufferedImage;

import java.awt.image.RenderedImage;

import java.io.File;

import java.io.IOException;

import java.util.ListIterator;

import java.util.Vector;
import javax.imageio.ImageIO;

import javax.swing.ImageIcon;

import javax.swing.JComponent;

import javax.swing.JFileChooser;

import javax.swing.JFrame;

import javax.swing.JLabel;

import javax.swing.JMenu;

import javax.swing.JMenuBar;

import javax.swing.JMenuItem;

import javax.swing.JOptionPane;

import javax.swing.JPanel;

import javax.swing.KeyStroke;

import javax.swing.filechooser.FileFilter;


/**<p>

 * This application allows you to do a couple

 * different related things. First, it is a GUI

 * interface that allows you to use a JFileChooser

 * dialog to select an image file type to open

 * and view.

 * <p>

 * The second thing you can do is draw a doodle onto

 * a canvas and save it as an image file.

 * <p>

 * There are several classes in this file.

 * The main class is called DrawingApp, and it allows

 * you to open image files. Another class called

 * AnImageFilter extends FileFilter, and provides a

 * filter implementation that the  JFileChooser dia

 * log uses in order to make sure that only

 * image file types are opened by the user.<br>

 * There is a class called Doodler, that registers

 * mouse events to allow you to draw on the canvas.

 * <p>

 * You might extend this to include a JColorChooser

 * dialog to allow the user to change the background

 * color of the canvas, or to change the color
 * of the pen.
 * <p>
 * thank you to rockstar pawel zurek for his very
 * helpful ideas on this app.
 * @author eben hewitt
 **/
public class DrawingApp {
//the main window
protected JFrame frame;
//the panels hold the content
protected JPanel panel;
protected JPanel contentPane;
//holds the File drop-down menu
protected JMenuBar menuBar;
//this is where the user draws
protected Doodler canvas;

private boolean alreadyHasImage = false;

//start the app
public static void main(String[] args) {
//create instance by calling default constructor
new DrawingApp();
}
//constructor
public DrawingApp(){
//make the window pretty with static method
//which must be called before instantiating the window
JFrame.setDefaultLookAndFeelDecorated(true);
//now make the window
frame = new JFrame();
//this title will appear in upper-left of frame
frame.setTitle("Garage Draw Pad");
//initialize the frame to dispose of the jvm
//instance it is using when user closes the window
frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
//instantiate the panel
panel = new JPanel();
//give it a really simple layout type so we can
```

```
//add stuff to the panel
panel.setLayout(new BorderLayout());
//make it be 300x300 when it opens
panel.setPreferredSize(new Dimension(300,300));
//add the working area to the window
//in AWT we didn't have to do this.
//we have to do it in Swing because a JFrame
//has a number of layers, and the add call will be
//ignored on the root pane. this makes sure we get
//the layer of the frame where our stuff happens, and
//not where frame management is going on.
frame.getContentPane().add(panel);
/*
 * strangely, in my view, we have to set the
 * background color of the <i>frame</i> to white if
 * we want the drawing we make to be saved
 * with a white background; otherwise, the
 * background will be gray, instead of the white the
 * user sees when he clicks New. that's because of
 * the layers in a frame.
 */
frame.setBackground(Color.WHITE);

//create the menu
menuBar = new JMenuBar();
//call our method that makes the menu and then
//add the menu to the menuBar
menuBar.add(makeFileMenu());
//must do this to work with it
menuBar.setOpaque(true);
//add the menu bar to the frame
frame.setJMenuBar(menuBar);

//put it at this x,y location on the screen
frame.setLocation(350,350);
//the pack method is inherited from java.awt.Window.
//it makes the window displayable by sizing the
//components to fit into it. it then validates
//the layout.
```

```
frame.pack();

//show the window on the screen.

//the user can now interact with it, so return from

//the constructor and wait until the user does

//something.

frame.setVisible(true);

}


/**

 * Makes the menu by adding file items to the menu.

 * It also handles the job of creating a listener

 * for each of those items (such as New, Open, etc),

 * to tell the app what action to perform when the

 * user chooses that item.

 * @return The completed menu, ready to add

 * to the JMenuBar.

 */

private JMenu makeFileMenu(){

//get the action command from the menu

//to determine what the user wants to do

//and call a separate method to do the work

//ActionListener is an interface that extends EventListener

ActionListener listener = new ActionListener() {

//when an action event occurs (the user clicks a

menu

//item) the actionPerformed(ActionEvent) method is

//called. so we override it to do what we want.

public void actionPerformed(ActionEvent event) {

String command = event.getActionCommand();

//user clicked File > New

if (command.equals("New")){

//call our custom worker

doNew();

} else if(command.equals("Open...")){

doOpen();

} else if(command.equals("Save...")){

doSave();

} else if (command.equals("Exit")){

doExit();
}

}

};

//this is the File menu that will be added to

//the menubar. so we have to add each menu item

//to it first.

JMenu fileMenu = new JMenu("File");


//NEW

JMenuItem newCmd = new JMenuItem("New");

//the user can alternatively type the control key + N

//to generate the same event as clicking

newCmd.setAccelerator(KeyStroke.getKeyStroke("ctrl N"));

//register the listener

newCmd.addActionListener(listener);

//put this item onto the File menu

//items get added in order

fileMenu.add(newCmd);


  //other commands work same way...


//OPEN

JMenuItem openCmd = new JMenuItem("Open...");

openCmd.setAccelerator(KeyStroke.getKeyStroke("ctrl O"));

openCmd.addActionListener(listener);

fileMenu.add(openCmd);


//SAVE

JMenuItem saveCmd = new JMenuItem("Save...");

saveCmd.setAccelerator(KeyStroke.getKeyStroke("ctrl S"));

saveCmd.addActionListener(listener);

fileMenu.add(saveCmd);


//QUIT

JMenuItem exitCmd = new JMenuItem("Exit");

exitCmd.setAccelerator(KeyStroke.getKeyStroke("ctrl E"));

exitCmd.addActionListener(listener);

fileMenu.add(exitCmd);
```

```
return fileMenu;

} //end makeFileMenu()


/**

 * action on NEW

 */

private void doNew() {

//do work of "New" command from the File menu

try {


//if there is a canvas already, wipe it clean

//to draw a new picture

if( canvas != null ) {

//alert user

JOptionPane.showMessageDialog(null,"This will destroy current doodle");

//gets rid of the old panel

panel.remove(canvas);

}

//make this new object on which to draw

//and set its size to 300x300

canvas = new Doodler(300, 300);


//show a white background so the user sees it.

//note that since we won't be saving

//the <i>panel</i> to an image file,

//the background of the image wouldn't be white

//with only this!

panel.setBackground(Color.WHITE);


//put the canvas in the content panel

panel.add(canvas, BorderLayout.CENTER);


//create a crosshair cursor to draw with

Cursor cursor =

Cursor.getPredefinedCursor(Cursor.CROSSHAIR_CURSOR);

//make the canvas use the cursor we just created
```

```
canvas.setCursor(cursor);

//call this to repaint the window

frame.validate();

} catch (Exception e){

System.out.println("Exception in doNew():

        " + e.getMessage());

System.exit(1);

}

}


/**

 * action on OPEN

 */

private void doOpen(){




//create new file chooser dialog box

//the dialog will use the user's home directory

//as a starting place

JFileChooser fileChooser = new JFileChooser();

//if you wanted to specify a different dir to start

//in, you could do this for example in linux:

//JFileChooser f = new JFileChooser(new

        File("/home/dude"));


//set the file chooser to only see .gif file types

//see the AnImageFilter class here for details

fileChooser.addChoosableFileFilter(new

        AnImageFilter());


fileChooser.setFileSelectionMode(JFileChooser.FILES_

        AND_DIRECTORIES);


//open the file chooser and make the frame its

//parent which means, if you close the frame, the //dialog closes too

int returnValue = fileChooser.showOpenDialog(frame);


//when user clicks open, do this

if(returnValue == JFileChooser.APPROVE_OPTION) {
```

```
//create a file object based on the chosen file
File file = fileChooser.getSelectedFile();

try{

if(alreadyHasImage) {
JOptionPane.showMessageDialog(frame,"Opening removes the current image");
//gets rid of the old panel
frame.getContentPane().remove(panel);
//make a new clean one
panel = new JPanel();
//add the panel to the content pane
frame.getContentPane().add(panel);
}

//read it in as an image
BufferedImage image = ImageIO.read(file.toURL());
//use the file to create an ImageIcon object
//and make it the value of a label that we can
//easily display on the panel
JLabel imageLabel = new JLabel(new ImageIcon(image));

/*
 * Note that by adding a doodler object to the
 * canvas, we will be able to draw on any open image
 * that is a previously saved doodle.
 * But a photo we won't be able to draw on.
 */
canvas = new Doodler(300, 300);
panel.add(canvas);

panel.add(imageLabel);

alreadyHasImage = true;
frame.validate();

} catch (Exception e){
System.out.println("Exception in doOpen(): " + e);
System.exit(1);

} //end catch

} //end if

} //end doOpen


/**

 * action on SAVE

 */

private void doSave() {

//file to be saved

File outFile;

//lets user specify where to save file

JFileChooser fileChooser = new JFileChooser();

//specify some things about the file chooser dialog

fileChooser.setDialogTitle("Save As...");

   //show the dialog and make the frame its parent

int action = fileChooser.showSaveDialog(frame);

if (action != JFileChooser.APPROVE_OPTION) {

//user cancelled, so quit the dialog

return;

}

//point the file the user chose to this object

outFile = fileChooser.getSelectedFile();


if (outFile.exists()) {

//if file exists already, make sure that the

//user wants to replace it

action = JOptionPane.showConfirmDialog(frame,

  "Replace existing file?");

if (action != JOptionPane.YES_OPTION)

return;

}

try {

//get the coordinates of the corners of the

//canvas so we have the part we want to save

Rectangle rect = canvas.getBounds();


//create an image ready for double-buffering
```

```
//from the screen area bound
//by that rectangle. This method returns null
//if the component is not displayable.
Image image = canvas.createImage(rect.width, rect.height);

//creates the context required for drawing the image
Graphics g = image.getGraphics();

//paint onto the canvas the lines created by the user
canvas.paint(g);

//save the image file using jpeg compression format
ImageIO.write((RenderedImage)image, "jpg", outFile);

    //catch any problems encountered during the save
} catch (IOException e) {
//print any exception to a messagebox (like a JS alert
//or like MessageBox.Show(...) in C#)
JOptionPane.showMessageDialog(frame,
"IOException in doSave(): " + e.getMessage());
System.out.println(e.getCause().getMessage());
System.exit(1);
}
}

/**
 * Exit the application and stop the VM.
 */
private void doExit() {
// stop the application
System.exit(0);
}
}//end DrawingApp class

/**
 * class to define the images only filter
 * we must subclass FileFilter
 */
class AnImageFilter extends FileFilter {

//find the extension of this file type
//so we know whether or not to include it
public static String getExtension(File f) {
String extension = null;
String fileName = f.getName();
int i = fileName.lastIndexOf('.');

if (i > 0 && i < fileName.length() - 1) {
extension = fileName.substring(i+1).toLowerCase();
}
return extension;
}

/* define the file types we are willing
 * to accept. also show directories
 * so the user can navigate into them
 */
public boolean accept(File f) {
if (f.isDirectory()) {
return true;
}
//we're going to save drawings with jpg compression,
//so this is for show really
String extension = getExtension(f);
if (extension != null) {
if (extension.equals("tiff") ||
extension.equals("tif") ||
extension.equals("gif") ||
extension.equals("jpeg") ||
extension.equals("jpg") ) {

return true;
} else {
return false;
}
}
```

```
return false;

}

//what the user will see in "Files of Type..."

public String getDescription(){

return "*.tif, *.tiff, *.gif, *.jpeg, *.jpg";

}

} //end AnImageFilter




/**

 * Doodler class uses AWT canvas to draw on.

 * Note that while you may instinctively want to

 * extend the AWT Canvas class here, it is not

 * necessary and will mess things up: it will make

 * your File menu inaccessible! The reason is that

 * Canvas is an AWT (meaning heavy-weight)

 * component and JMenuBar is a lightweight Swing

 * component. Remember: AWT components will ALWAYS

 * cover Swing components.

 * JPanel would also have worked here.

 */

class Doodler extends JComponent {

//these ints hold the coordinates

private int lastX;

private int lastY;

private Vector plots = null;


private Vector pointData = null;

//declare two ints to hold coordinates.

//remember that because these are class-level variables,

//they will be initialized to their default values

//(0 for int).

private int x1, y1;

private Graphics graphics = null;


//constructor

public Doodler (int width, int height) {

//call the default constructor of the superclass
```

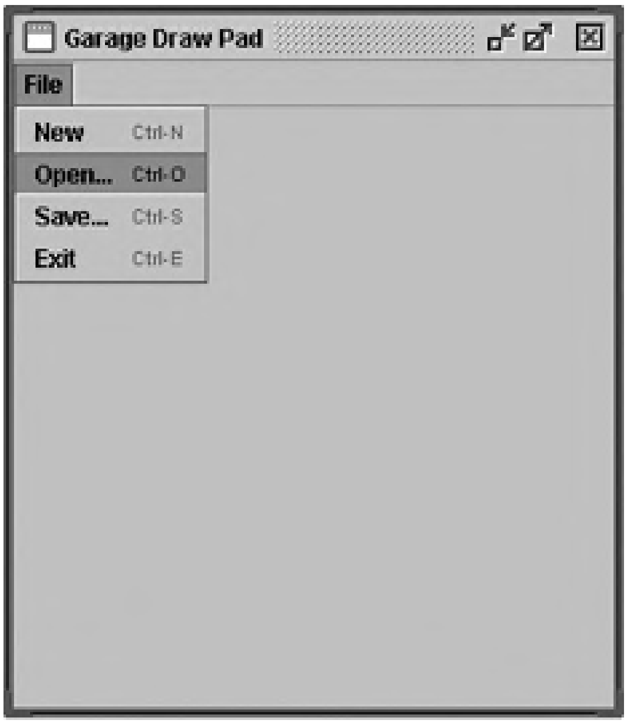
```
//(JComponent)—<i>not</i> Canvas!

super();

this.setSize(width,height);

plots = new Vector();

pointData = new Vector();

/*

 * Listen for the event that is fired when the

 * mouse button is pressed, because we don't want

 * to draw whereever the mouse goes—only when it

 * is pressed.

 */

addMouseListener(new MouseAdapter() {

public void mousePressed(MouseEvent e) {

x1 = e.getX(); y1 = e.getY();

pointData.add(new Point(x1, y1));

}

public void mouseReleased(MouseEvent e) {

plots.add((Vector)pointData.clone());

}

});

/*

 * Listen for the movement of the mouse being

dragged

 * so that we can capture each point across which it

 * was dragged and add it to our vector that stores

 * all the points; then let the drawLine method of

 * the Graphics class connect the dots.

 */

addMouseMotionListener(new MouseMotionAdapter() {

public void mouseDragged(MouseEvent e) {

int x2 = e.getX(); int y2 = e.getY();

pointData.add(new Point(x2, y2));

graphics = getGraphics();

graphics.drawLine(x1, y1, x2, y2);

x1 = x2; y1 = y2;
}
});
}
/*
 * Paints our doodle line by storing each point over
 * which the pressed mouse passes in a vector.
 * @see java.awt.Component#paint(java.awt.Graphics)
 */
public void paint(Graphics g) {
//returns a way to iterate over
//all of the elements in this list
ListIterator it = plots.listIterator();

while (it.hasNext()) {
Vector v = (Vector)it.next();
Point point1 = (Point)v.get(0);
//remember you can do more than one thing in a for
//loop's first statement
for (int i=1, size = v.size(); i < size; i+=2) {
Point point2 = (Point)v.get(i);
g.drawLine(point1.x, point1.y, point2.x, point2.y);
point1 = point2;
} // end for
} // end while
} //end paint override

}//end Doodler class
```

As you can see by reading over it, there are a few different class files in this one source file. Recall that only one of the classes may be public, and it must be named the same as your source file.

Let's walk through the code and see what kinds of things happen on your screen when you execute the program.

Figure 35.9 shows what the app looks like when you first load it and click on the File menu.

Figure 35.9. The Doodler application when the application is executed and the File menu is clicked. Notice that the frame is decorated—not just the Windows default.



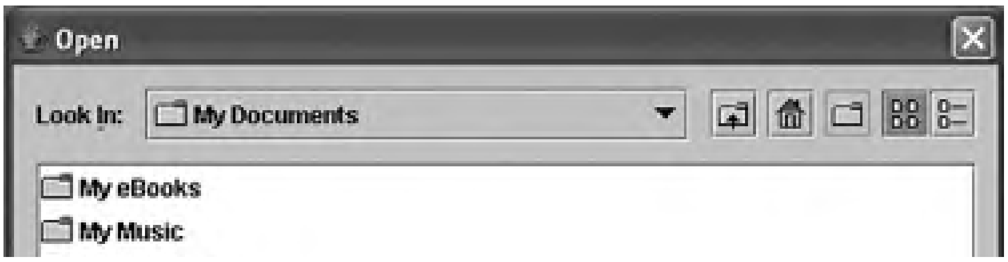
First, let's read in an image. When we type Ctrl + o on the keyboard, or click the Open menu item, this executes the doOpen() method. We did not have to put this functionality into a separate method, but it does make the JMenu business a little easier to port to a new app, and is helpful for the code reader—if she's not interested in the doOpen() method right now, it's easy to skip by it and find what she's looking for. As we'll see later with the doNew() method, there are times that you want to create a whole new object, or even a new thread, and let that guy take over for a while.

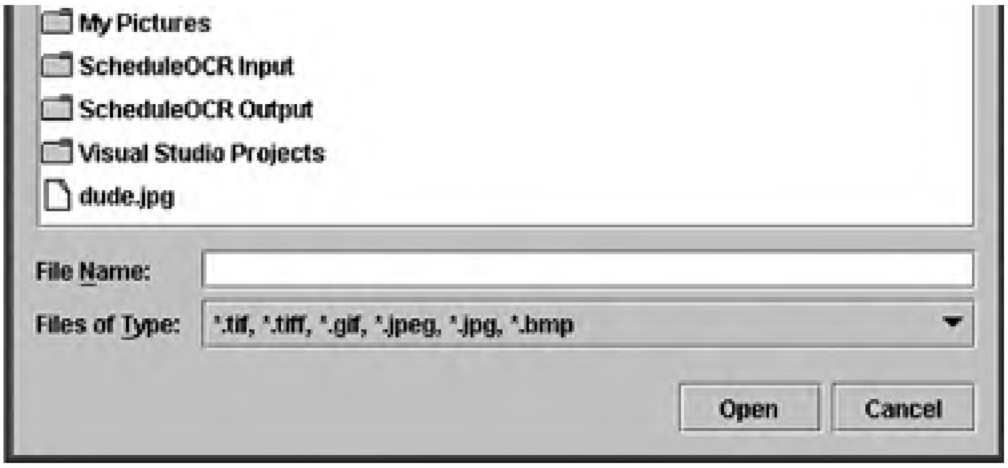
FRIDGE

The DrawingPad app won't read in a bitmapped image. The ImageIO class gets an ImageReader to decode the URL passed to it. The way ours is written, we don't decode a .bmp, so null will be returned by the static call to ImageIO.read(file.toURL());. As a consequence, a NullPointerException will be thrown if you try do that.

But back to our regularly scheduled program. We were opening a file. To do this, we get a JFileChooser dialog, which is a standard part of the API (see Figure 35.10). This is a really terrific component because it is lightweight and has a lot of functionality. After the user selects the file he's interested in, we use a BufferedImage reference to read in the data using an ImageInputStream and create it as an ImageIcon, which we pass to the JLabel constructor. We then add the JLabel to the panel, and ask the container to redraw itself.

Figure 35.10. The JFileChooser is a great component for both opening and saving files of all types. Notice that the filter is applied, and so we only see folders, and files of one of our pre-defined extension (image) types.

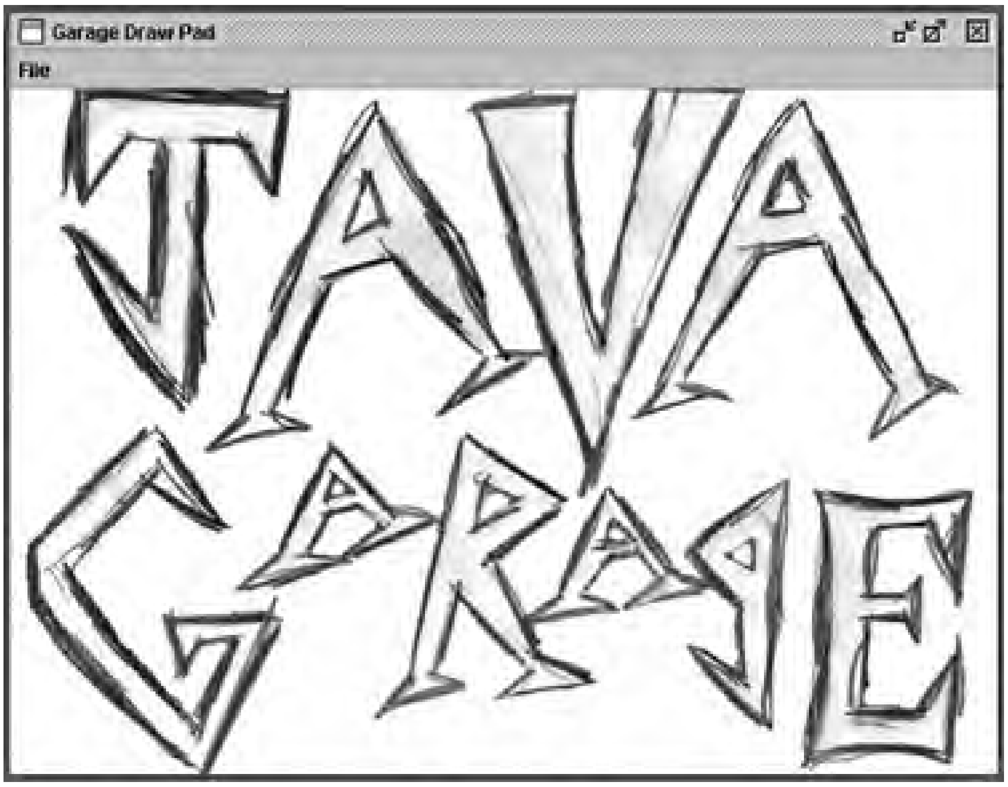




Note that the JFileChooser opens in the default user location; in the case of Windows, this is My Documents. We could also specify a different location in the constructor.

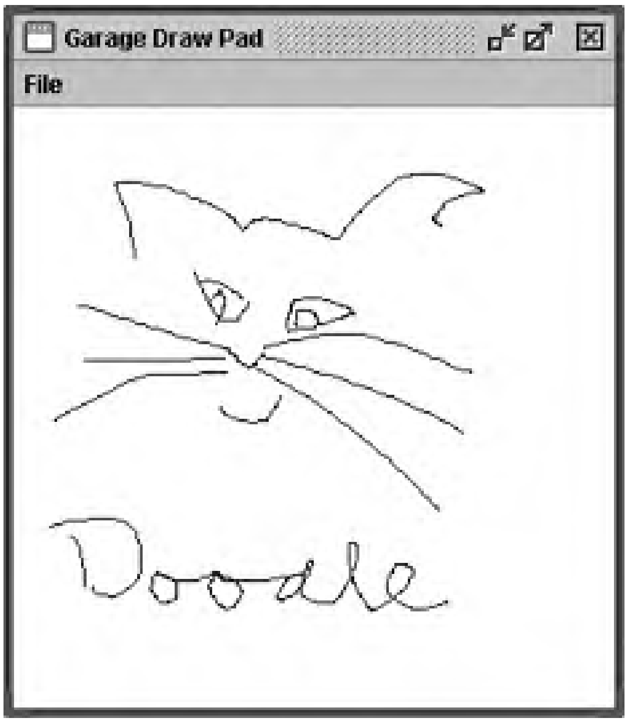
Now, you'll notice that the image may be larger than our 300x300 frame size, in which case we can just drag a corner of the window, and it will automagically resize itself to fit the image (see Figure 35.11). We could have used a JScrollPane to allow the user to view only part of the image, but that seemed like not-very-useful functionality for the added complexity. Also, we've already seen how to use JScrollPane in the RSS newsreader app.

Figure 35.11. The image you open may be larger than the frame, in which case you can drag any one side of the frame to resize the viewing area to snugly fit the image.



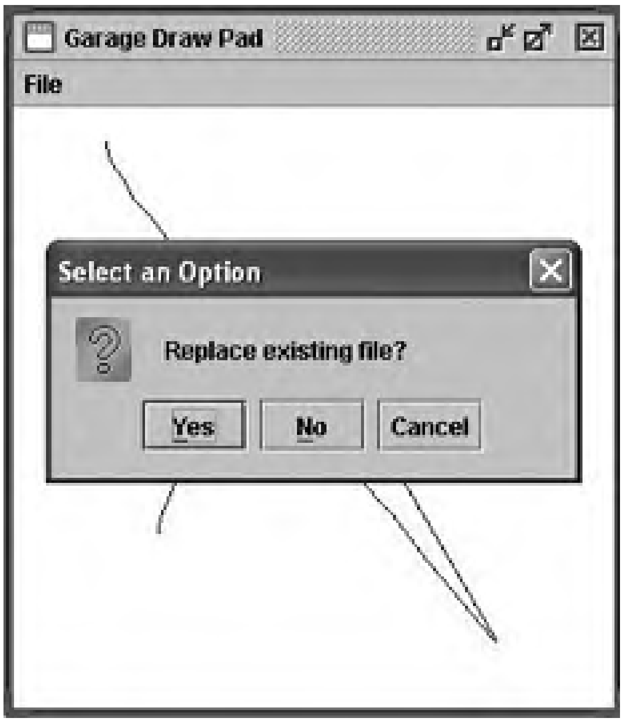
It would be nice if we could draw on a photographic image that we open, but we cannot do that. However, if you create a new doodle in this app, save it, and then open it later, you will be able to continue drawing on it (see Figure 35.12).

Figure 35.12. This is a picture of my kitty cat Doodlehead. It's an original artwork.



If you attempt to save a file using the name and path of a file that already exists, the application detects this and warns you that you are about to overwrite the existing file (see Figure 35.13). If you don't want to do that, you can hit Cancel and nothing happens. You can then click Save again and choose another name or path.

Figure 35.13. The drawing pad uses JOptionPane's showConfirmDialog method to create a dialog alert with a Cancel button.



Replace existing file?

Of course, clicking the Exit button on the menu cleanly exits the application and shuts down the Java Virtual Machine.

Ok. Thanks a million. I hope that you find a load of stuff in the toolkit you can pillage and use in your own apps.

Chapter 36. SYSTEM.EXIT…

dude yt?

ya

is that it?

huh?

this java garage is all ovah.

ya

what a rip off. i hate technical books.

da whole reason I got this Garage in da first place is so I could jazz it up my flux capacitor. I need two point two one jiggawatts!

YOU LOVE BIG PIPES I KNOW YOU DO DON"T DENY IT!

^_^

no dude my mind is like a blue screen o death on this java schlepp.

nod

uh...I remember some stuff though. we can use statements, design classes, write a console application, and write a GUI app. can read and write and images, and use regular expressions and exceptions and Streengs and formatting. and yadda yadda.

all right. that's true.

thank you.

so what happens next/

next you get More Java Garage.

ack a sequel. Java Revolutions.

that covers database access with Generics, Collections, JDBC, multithreading, internationalization, reflection, tuning, design patterns and more yee-haw like that.

but what about the web?

that's the next part. it is really a good idea to know what you're doing in java when you start working with web apps. it is a whole different deal. but everything you learned in this book will help you on the web stuff.

oh yeah? all that crepe about GUI programming???? :(

there are applets, you know. while you don't have access to the local file system in an applet, there are other ways to put that GUI programming to work on the web. for instance, web start.

whats webstart?

it is a way of packaging apps for deployment from the web that lets the user run them locally. so it is like the best of both worlds: you get all the richness of a fat client, and none of the maintenance headaches.

but what about JSPs and servlets?

that is the logical next step. the thing to do is start applying your java knowledge to JSPs. they allow you to write html code intermingled with java code and tags that execute java code so you can have dynamic applications that run on the web. now with jsp 2.0 it is a lot easier to get up and running than before. they have added a lot of tags to the regular distribution.

can I start doing web apps with the Java 5 SDK?

not the standard edition. you have to get the enterprise edition right now it's J2EE 1.4

cool. thanx

np.

you wanna get some lunch?

i was going to get me some Commander Taco.

dude. you got to stop eating in your car.

:P

need some company?

yeah, come and stuff.

i promise not to lecture you anymore about EPL soccer rules

ha. dude. you can yammer whatever the fook you wants

cool

come down.

gimme a sec with this patch install. ok. cya

ok cya

CHAR		HEX	DEC
(ETX)	^C	0x03	&#003; END OF TEXT

Copyright

Library of Congress Cataloging-in-Publication Data

A CIP catalog record for this book can be obtained from the Library of Congress

Acquisitions Editor: John Neidhart

Editorial Assistant: Raquel Kaplan

Marketing Manager: Stephane Nakib

Marketing Specialist: Jen Lundberg

Publicity: Kerry Guiliano

Managing Editor: Gina Kanouse

Project Editor: Michael Thurston

Cover Design: Anthony Gemmellaro

Interior Design: Wanda Espana

Manufacturing Buyer: Dan Uhrig

Dedication

dedication.bas

```
10 PRINT "For Alison, of course"

20 GOTO 10

RUN
```

Eben Hewitt

Eben is a Sun Certified Java Programmer, Sun Certified Web Component Developer, Sun Certified Java Developer, and Senior Java Programmer/Analyst for Discount Tire Company in Scottsdale, AZ. He is the Garage Series Editor, and the author of four other programming books, including the acclaimed *Java for ColdFusion Developers*, and the forthcoming *More Java Garage*. He has been an invited speaker on Java at regional user groups.

Eben has a Master's Degree in Literary Theory, and had his first full-length, original play produced in New York City in 1996. Questions haunt him in this Champagne-light and Orwellian time: Who will win, Struts or Faces? What if they trade Johnson to the Yankees? At long last, have we left no sense of decency?

He and his family, Alison, Zoë, Mister Apache Tomcat, Doodle, and Noodle, burn to ashes every summer.

JAVA GLOSSARY ON STEROIDS

A

abstract

Used to specify a class that cannot be instantiated. Subclasses may inherit from an abstract class, that is, *extend* them. Also, modifies a method whose implementation you wish to defer to another class.

Abstract Windowing Toolkit (AWT)

Package of graphical user interface classes implemented in native platform component versions. AWT classes allow for the creation of buttons, frames, event handlers, and text in application interfaces. AWT's popularity has ebbed significantly in favor of the newer, more portable and lightweight Swing.

API

Application programming interface. The specification that dictates how a programmer must write to access the state and behavior of objects and classes.

applet

A (generally small) program written in Java that extends the java.applet.Applet class. By extending this class, the program becomes capable of executing in a Web browser or other device that supports applets in general. Applets helped create tremendous popularity for the Java programming language early in its life because they allow programs to be dynamically downloaded and securely executed. Applets can be viewed from the command line with appletviewer, a program of the standard SDK. Alternatively, applets can be viewed in a Web page by having the user download a plugin and referencing the applet with either the <object> or <applet> tags, both part of standard HTML. Microsoft's illegal incorporation of proprietary technology into its Java Virtual Machine seems to have had significant negative impact on the popularity of applets, as the necessary 5MB plugin download from Sun makes for a thicker client and additional user support.

argument

An argument is an item of data passed into a method. An argument may be an expression, a variable, or a literal.

array

An ordered set of data items, all of which must be of the same type. Each item in the array can be referenced by its integer position. The first element of a Java array is 0.

ASCII

Acronym for American Standard Code for Information Interchange. An ASCII code is the 7-bit numerical representation of a character, which can be understood by many different computing platforms.

B

bean

Beans are standard Java classes that conform to certain design and naming conventions (private variables and getter and setter methods). When people say, "JavaBean," they mean something that comes from the java.beans package. JavaBeans and Enterprise JavaBeans have very little relation (no more than, say, Java and JavaScript). The original purpose of beans was to make it easy for software vendors to write programs (such as IDEs) to allow users to visually manipulate the software component represented by the bean (like in VB). To achieve this, beans all share the following characteristics: Properties, Customization, Persistence, Events, and Introspection.

bit

Contracted form of *binary digit*, the smallest unit of information in a computer. A bit is capable of holding one of two possible values: on or off, commonly represented as true or false, or 0 or 1. The term is said to have been coined by the mathematician Claude Shannon.

bitwise operator

An operator that manipulates individual bits within a byte, generally by comparison, or by shifting the bits to the left or right.

block

Any Java code between curly braces. For instance: `{ int i = 0; }` is a block of code. Colloquially, developers refer to a "try/ catch block."

Boolean

The object wrapper for the primitive `boolean`.

boolean

One of the primitives in Java, a `boolean` represents either a true or false value. The default value is false. Note that boolean represents not a 1 or 0, but the literal value true or false.

bounding box

Used in graphical user interface (GUI) programming, refers to the smallest geometric area surrounding a shape.

bounds

Used to constrain the types that you can invoke on wildcards in generic method definitions. A lower bound means that the runtime value supplied for the wildcard must be higher up the inheritance hierarchy than the wildcard lower bound. An upper bound indicates that when you supply an actual type at runtime, it must be a subclass of the upper bound type. A lower bound is expressed with `? extends T`. Say you wanted to allow your wildcard to be "replaced" at runtime with only classes that are subclasses of java.lang.Exception. You would write `? Extends Exception` in your method definition. An upper bound is expressed with `? super T`. You can also use a parameter type as the bound of your wildcard, like this: `? extends E`. This means, "the argument supplied to this method at runtime must be a subclass of whatever the actual parameter type supplied is."

See also **[wildcard]**

break

A keyword in Java used to resume program execution at the next statement immediately following the current statement, except when a label is used, in which case execution returns to the label.

Byte

An object wrapper for a byte primitive.

byte

A Java keyword used to declare a primitive variable of type byte. Eight bits form one byte. A byte is a signed integer in Java; its minimum value is -27 and its maximum positive value is 27-1.

bytecode

Machine-independent code that is generated by the Java compiler and executed by the Java interpreter. The code in a .class file is bytecode, serving as input to the Java Virtual Machine. Bytecode can also be converted to native machine code by a Just In Time compiler.

C

case

A Java keyword, case is used inside a switch block to define a set of statements to be executed conditionally, depending on whether the case value matches the supplied switch value.

cast

To convert one data type to another. There are implicit primitive casts, in which you do not write the code to perform the conversion—it is done automatically. An explicit primitive cast demands that you write the code to perform the cast, because you are narrowing the primitive (putting it into a smaller container—for example, casting a long into an int). Narrowing conversions can cause loss of precision, which is why you must explicitly perform it.

catch

A Java keyword used to declare a block of statements to be executed in the event that an exception is thrown from code in the corresponding try block.

char

A Java keyword used to declare a primitive variable of type char. A char is used to represent Unicode character values. It is declared with single quote marks around the value, as in: char grade = 'a'; chars are considered 16-bit unsigned integer primitives, which means that Java allows direct conversion between char and other primitive integers. Its possible values range from 0 to 65535 (or 216-1).

class

A Java keyword representing the declaration of an implementation of a particular object. Classes are used to create instances of objects. Classes define instance and static variables and methods. They specify the interface that the class implements, and the immediate superclass that the class is extending. All behavior in Java programs occurs inside classes. Every class implicitly extends the class java.lang.Object. The class java.lang.Class can provide the runtime type of an object.

class file

The file created when a .java source code file is compiled. The name of the generated file will match the name of the source code file. The class file contains bytecode to be executed by the runtime.

class method

More commonly called a static method, a class method is one that can be invoked without reference to any particular object. Defines static methods when the execution will always be the same, regardless of the particular object invoking it. For example, a method that makes a connection to a datasource is often defined statically, because it is always performed in the same way, and is always returned the same result (a database connection) regardless of what instance calls it. Invoke class methods on the class itself. For example, the Collections class contains a number of static methods for convenience, such as sort(). You invoke it like this: Collections.sort(myArrayList);. The java.lang.Math class is another class that defines only static methods (which makes sense—the pow() method, which calculates the result of the first argument raised to the second argument, will always perform the same action).

Classpath

An environment variable indicating the location of class libraries used by the Java Virtual Machine. If you write a program that uses a particular library (for example, a JAR file that contains classes that help you convert data into PDF format), you need to put that JAR on your classpath so that the JVM can find it when it executes your code that references the classes in that JAR. Here's another common example: If you want to send an e-mail from a Java program, you can't do it using just the J2SE; you need to download J2EE, put the j2ee.jar file on your classpath, and then you can use the classes in the javax.mail package to send the e-mail.

class variable

Also known as a static variable, a class variable is defined at the class level, using the static modifier, and does not rely on any particular instance of the class to be invoked.

comment

Text that explains to yourself or other programmers what the purpose of this code is, what other code it affects, and other metadata such as the author's name and the created date. Single-line comments begin with //. Multi-line comments begin with /* and end with */. Javadoc comments, which the javadoc program uses to generate HTML documentation, use /**.

compile-time

The time at which a source file is compiled into a class file. Some errors are caught during compile-time.

compiler

The program distributed with J2SE called javac, which translates Java source code (in .java files) into Runnable bytecode (.class files).

constructor

A constructor is defined in a class file, and is used to create instances of the class. Constructors have the same name as their class, and are invoked using the new keyword. A Java class may define zero or more constructors. If no constructor is defined, the constructor for the superclass is automatically invoked. Additional constructors must specify a unique argument list, just like overloaded methods.

continue

A Java keyword used inside loops. Invoking it causes program execution to resume at the end of the current loop. As with break, if a label is used, execution resumes at the label.

D

deadlock

Deadlock is an unrecoverable state sometimes encountered in thread programming. It occurs under the following circumstances: thread A has a lock on object X, and is waiting for the lock to object Y; meanwhile, thread B has the lock on object Y, and is waiting for the lock on object X. In the event of a deadlock, program execution will simply suspend indefinitely, making it difficult to debug.

declaration

A statement that creates an identifier and associates attributes with it. Declaring a data item may or may not reserve memory space. A method declaration may or may not also provide the implementation.

default

A Java keyword used within a switch block to indicate the code block to execute if no case value matches the switch value.

do

A Java keyword used to declare a style of loop.

DOM

Document Object Model. An XML specification for representing data as an object tree.

Double

An object wrapper for the primitive double.

double

A Java keyword used to declare a 64-bit primitive of type double. Floating point numbers have ranges dependent upon the size of the mantissa and exponent fields. Double.MAX_VALUE and Double.MIN_VALUE indicate these for the IEEE double-precision data type. These represent the largest and smallest possible positive values.

double precision

A double precision variable is an IEEE, standard, floating point variable that is capable of representing numbers in the range 4.9 x 10-324 to 1.8 x 10308.

E

Encapsulation

A concept in object-oriented programming that dictates that the implementation of data and behavior within an object must be hidden, exposing only a public (client-facing) view of the interface, thereby enforcing a contract for acceptable client interaction with the object. Encapsulation of your code increases portability and reuse, and is a very good thing to do. Ways of encapsulating your code include making private variables, with public getX() and setX() methods.

error

A state from which a program cannot recover. java.lang.Error is a subclass of java.lang.Throwable, and is the parent class of all Java error classes. A runtime error can occur in the following circumstances: when an exception is not caught by any catch block (in which case the controlling thread invokes the uncaughtException method and terminates), when you try to cast a class into a type that it cannot be cast to (in which case a ClassCastException is thrown), or when you try to store an object in an array meant to hold different object types (in which case an Array StoreException is thrown). There are also LinkageErrors, which are thrown when a loading, linkage, preparation, verification, or initialization error occurs in the classloader.

exception

A programmatic state that subverts the normal flow of execution. Exceptions should be handled by anticipating the possibility of their occurrence, and surrounding the potentially problematic code with a try/catch block.

extends

A Java keyword to indicate that the current class is a subclass of the class it extends. By extending a class, a class inherits the functionality defined in its parent class (also called the superclass). This functionality can then be added to or, if the parent data item is not declared final, overridden.

F

field

A variable defined in a class, also called a member.

final

A Java keyword indicating a data item (class, field, or method) that cannot be overridden. A final class cannot be subclassed. For example, the following statement is illegal: `public class SuperString extends String {}`.

finally

A Java keyword indicating a block of statements that should execute whether or not the code within a corresponding `try` block threw an exception. A `finally` clause is not required, and is typically used to clean up resources (such as to close a connection to a database or socket, or to close a file opened in a corresponding `try` block).

Float

An object wrapper for the primitive `float`. This wrapper class defines a number of possibly useful constants, including `NaN` (Not a Number), `MAX_VALUE`, and `POSITIVE_INFINITY`.

float

A Java keyword used to declare a primitive 32-bit floating point number variable. Its numerical range is (1.4 * 10^-45) (Float.MIN_VALUE) through (3.4028235 * 10^38) (Float.MAX_VALUE).

for

A Java keyword used to declare a kind of loop that iterates until a specified condition is met.

G

Garbage Collection

Name for the process by which the Java Virtual Machine frees memory by reclaiming resources that are no longer referenced or accessible by the program. This feature of Java makes it easier to program in than other languages in which the programmer must explicitly free objects. The garbage collector executes on the heap, because that is what object data is stored.

generics

The name given to the collective functionality that allows a programming language to provide generic types and generic methods. In Java, generics are often compared to C++ templates. This comparison is understandable, as templates and generics aim to provide similar kinds of functionality. However, you should probably abandon this comparison readily, as the two are implemented very differently.

See also **[Generic Type]**
See also **[raw type]**
See also **[wildcard]**

Generic Type

Generic types include both generic classes and generic interfaces. These declare one or more variables, called type parameters, and are special because they are of unknown type until runtime. An example of a generic type is ArrayList<E>. The <E> indicates that you can create an instance of the ArrayList class by passing a specific type to it at runtime, like this: ArrayList<String>. This invocation means that this ArrayList will hold only objects of type String. The runtime will know that, and then you won't have to cast down from Object when you get the elements back out.

See also **[raw type]**
See also **[parameterized type]**

GUI

Graphical user interface. Pronounced "gooey." Components that allow the user to visually interact with a program. In Java, GUIs are created using (primarily) the Swing library.

H

heap

The heap is where Java objects are stored. Sometimes it is called the garbage collectible heap because this is the only area in which the garbage collector runs, ensuring that there is as much free memory as possible.

Hexadecimal

Base 16 numbering system in which 0-9 and A-F represent the numbers 0 through 15. Java hexadecimals are prefixed with 0x. Note that the letters used to represent a hexadecimal number are not case sensitive—that is, 0xAFC is identical to 0xafc.

I

identifier

A name used to represent a data item in a Java program such as a field, method, or class.

implements

A Java keyword used in the class declaration to indicate the name of an interface that this class provides an implementation for. A class can implement multiple interfaces; when this is done, separate interface names with a comma, as in this example: public final class String extends Object implements CharSequence, Comparable, Serializable.

import

Allows the programmer to reference individual class names in code without using the fully qualified class name (that is, without using the package name). Only classes in the java.lang package are imported automatically—and because of this, we can simply type String in our code, instead of java.lang.String. Note that as of Java 1.5, you can perform static imports, which means that you can import the static methods and fields of a class so that you don't have to prefix the class name when you invoke them.

Inheritance

In object-oriented programming, the concept that a class automatically receives data and functionality (variables and methods) from its parent class. In Java, all classes inherit from java.lang.Object. The parent class is also referred to as the superclass, in which case, the inheriting class is referred to as the subclass.

Instance

One particular object of a class—that is, the object referred to by this. To create an instance of an object, use the new keyword. For example, this code creates a new instance of the Integer class: Integer x = new Integer(42);.

instanceof

A Java keyword indicating the operator that determines if a reference is of a certain type. The operation returns true if the reference passed can legally be cast to the given type. For example, the statement return "Hey!" instanceof Object; will return true, because String extends Object. Note that it also works when the reference type is an interface: return (new ArrayList() instanceof Collection) also returns true.

int

A Java keyword used to declare a 32-bit signed integer primitive in the range of 231 to 231-1 (or -2,147,483,648 to 2,147,483,647).

interface

A Java keyword used to define a set of methods and constants without implementation. A class that implements the interface must provide an implementation for each method defined in the interface. An interface can be used as the reference type for a class that implements the interface. For example, ArrayList implements the methods of the Collection interface, so we can legally write this: Collection myArrayList = new ArrayList();. We can then use our myArrayList object anywhere that a Collection is called for.

Iterator

An interface that offers a way to allow a caller to access each element in a collection without exposing the collection itself to the caller. The benefit is one of increased encapsulation. Iterator is an improvement over the older Enumeration, in that they feature clearer, more precise method names, and offer a way to remove elements from the collection.

J

Java Virtual Machine

Part of the Java Runtime Environment, the JVM is the program that interprets and executes bytecode in compiled Java class files. The JVM is invoked with the java command and supplies programs for a context in which to run, providing resources such as memory management via garbage collection, network access, and security. The JVM acts as an abstraction of the hardware layer, enabling programs written in Java to run on any platform with a Java Virtual Machine. There are different JVM specifications, including one for smart cards and micro devices.

J2EE

Java 2, Enterprise Edition is a free download, separate from the Standard Edition, that offers APIs for database, advanced networking, server-side, and distributed functionality.

J2ME

Java 2 Micro Edition is required for writing small footprint programs runnable micro devices including smart cards, set-top boxes, and handhelds.

J2SE

Java 2 Standard Edition provides the basic development environment for creating portable, secure, network-ready software applications. When declaring an interface, you may only use the public and abstract modifiers. That is, you cannot declare a static method in an interface.

JAR

A Java ARchive is a file format used to aggregate and compress a number of files into one. Because the jar utility, which is included with the standard SDK, uses the same compression algorithm as .zip files, JARs can be manipulated using Zip utilities.

JDBC

Java Database Connectivity is an API that allows programmers to access tabular data stored in relational and object databases, text files, and spreadsheets. JDBC is included with the J2SE.

JDK

Java Development Kit also referred to as SDK (Software Development Kit).

JFC

Java Foundation Classes. Set of Java class libraries, included with J2SE, that support the building of rich Graphical User Interface applications. The JFC includes AWT (Abstract Window Toolkit), Drag and Drop APIs, Java 2D, Swing, Accessibility APIs, and Internationalization support.

JIT Compiler

A Just In Time compiler takes the bytecodes of a .class file at runtime and compiles them into code native to the machine on which it's running. A JIT compiler therefore replaces the functionality of the Java bytecode interpreter. This happens on-the-fly, method-by-method (hence, "just in time").

JNDI

Java Naming and Directory Interface is a set of APIs that support interaction with directory services such as LDAP.

JNI

Java Native Interface allows code that runs in a JVM to interact with code written in other languages, such as C, C++, and assembly. JNI, which is part of the J2SE, is used by programmers when they cannot or do not wish to write their entire application in Java. This situation would occur when you already have some native code written that you'd like to reuse, or if Java does not support the operation you need to perform.

Java RMI

Java Remote Method Invocation is for use in distributed environments, and allows an object in one virtual machine to call methods on an object in another virtual machine.

Java Runtime Environment

The JRE is a subset of the SDK. It includes the Java HotSpot runtime, client compiler, the Java Plugin, and nearly twenty standard libraries (the core classes). It does not include the compiler, the debugger, or other such tools. The JRE is the minimum runtime required for executing Java applications; if your machine ain't got a JRE, it won't run your Java program. That's why JREs are freely distributable, because developers often need to package one along with their applications. Note that the SDK is not freely distributed by developers.

K

keyword

A word reserved by the Java programming language for its exclusive use, which is therefore not available for developer use in naming variables or methods. For example, `String for = "x";` is illegal because `for` is a keyword. Note that two terms, `const` and `goto`, are reserved but not used.

L

Local inner class

A local inner class is a class defined within the scope of a method. Local inner classes are often also anonymous classes. An important caveat regarding their use: Local inner classes cannot see variables defined in their enclosing method *unless* those variables are marked final.

Local variable

A variable declared within a method. Local variables are sometimes (rarely) referred to as automatic variables. Local variables are distinct from class variables in that they must be initialized before you can use them. Local variables' visibility is only the life of the method—when the method in which they're defined returns, local variables vanish.

Long

Object wrapper class for long primitives.

long

Java keyword used to declare a primitive variable of type long, which uses 64 bits to represent integral values ranging from -263 to 263-1 (or -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807). Yes, that's over 9 *quintillion*.

M

members

The data elements that make up a class: inner classes, methods, and variables.

method

An element defined inside a class that performs a function, can accept arguments, and return a single result. A method body consists of the block of statements inside curly braces.

modifier

A keyword placed in the definition of a class, method, or variable that changes how the element acts. There are visibility modifiers (typically called access modifiers), such as `public`, `protected`, and `private`, and other modifiers, such as `final` or `native`.

modulus

The operator that returns the remainder of an integer division operation, represented in Java as `%`. For example, `5 % 2` returns 1.

multiple inheritance

Refers to the capability of a programming language to support a class extending more than one class. This is possible in C++, but not in Java. This decision was made on purpose in order to help code clarity and encourage good design.

multithreaded

A program that executed multiple threads concurrently.

N

NaN

Not a Number. A floating point constant representing the result of attempting to perform an erroneous mathematical operation (such as trying to divide by zero). Referencing the static variable Double.NaN prints "NaN", and calling the non-static method isNaN() on a Float or Double object returns a boolean. For example: return new Float(3.14).isNaN() returns false.

null

Represents the absence of a value, using the ASCII literal null.

O

Object

An object is an instance of a particular class, making it the fundamental unit of an object-oriented application. Objects typically store data in variables, and provide methods with which to operate on that data.

OOP

Object-oriented programming. Important concepts in OOP include Encapsulation, Inheritance, and Polymorphism, all of which Java supports.

overloading

A method is overloaded when the same class defines multiple methods with the same name but different argument lists. Constructors may also be overloaded.

overriding

A method is overridden when a subclass defines a method with the same name, return type, and parameter list as a method in a superclass.

P

package

A Java keyword that is used to indicate the package name to which this class belongs. Conceptually, a package is a group of classes with a logically similar purpose.

parameterized type

A generic type, with actual arguments supplied for its type variables. For example, ArrayList<E> is a generic type. You can invoke it as ArrayList<Product> and then that is called the parameterized type.

primitive

A Java primitive is a simple type that evaluates to the single value stored in that variable (as opposed to a reference type, whose value is its memory address). Primitives cannot have methods or hold any other data than their single value. You cannot cast a primitive type to an object reference, or vice versa.

private

A Java keyword used as an access modifier to indicate that this member is not visible to any other class but the one in which it is defined.

process

A space in a virtual addressing system containing one or more threads of execution.

protected

A Java keyword used as an access modifier to indicate that this member is visible only to the class in which it is defined, any subclass of that class, or any class in the same package.

public

A Java keyword used as an access modifier to indicate that this member can be accessed by any other class.

R

raw type

A parameterized type invoked without its parameter. ArrayList<E> is a parameterized type. You can either supply a parameter for E, or you can forget about it, in which case your invocation is ArrayList aList = new ArrayList();, and aList is then referred to as a raw type.

reference

A data element (object or interface) that serves as a pointer to an address in memory where the object is stored. A reference variable is a name used to reference a particular instance of a Java class.

return

A Java keyword used to indicate that a method has completed execution, and should give control back to the caller. return is optionally followed by a value or expression as required by the method definition. The following are all legal: return;, return "choppers";, return (x < (7 + y));, and return new Double(42);.

runtime

though "at runtime" is sometimes used to mean "during execution of the program."

See also **[Java Runtime Environment]**

S

scope

A member's scope dictates its visibility. An automatic (local) variable is one defined inside a method body; its scope is the method itself, and it is not visible to code outside the defining method.

shift operator

The shift operators move the bits of an integer number to the right or the left, which results in another number.

short

A Java keyword used to declare a 16-bit signed primitive integer type in the range of -215 to 215-1 (or -32,768 to 32,767).

signature

A method's name and the type and order of its parameter list.

stack

The place where Java stores references to objects. The object data itself is stored on the heap. The stack is set to an initial fixed size, whereas the heap can grow to consume all available system memory. When a JVM thread is created (when a program starts), a separate, private stack is created for that thread. The stack stores frames, which contain the data.

stack trace

A list of called methods, in descending order in which they were called. The list represents the threads and monitors in the Java Virtual Machine, typically that led to an exceptional state. If the JVM experiences an internal error (say, because an exception is thrown), it will signal itself to print out the stack trace. Useful in debugging, the method call on the top of the stack trace should be the last method called when the program encountered an error.

static

A Java keyword used to define a variable, method, or standalone block of code as being executable without an instance of the class. Because static variables are called on a class, there is only one copy of a static variable, regardless of how many instances of the class there may be. Static methods may only operate on static variables. Notice that you can just write inside a class `static{...}`, which declares a block of code (not a method) that executes when the class loads (it would be polite of us to notice too that this technique is sometimes frowned upon, as it is obdurately defies object orientation). It is useful in invoking libraries used by native method calls.

static method

A method called directly on a class, not an instance of the class. Static methods do not have knowledge of instance variables. For example, the following statement is a static method declaration: `public static void main(String[] args)`. It makes sense that the famous `main` method would be static—how could you start your application if you had to make an instance of some object first in order to call the `main` method? You would be making objects, in which case you would have already started your application. The following example shows using a class name to call one of its static methods without an instance of it: `System.exit(0);`. Methods declared in an interface may *not* be declared static.

static variable

A variable available directly from the class itself, not an instance of the class. Also known as class variables, static variables will have the same value across every instance of the class.

stream

Data read in or written out as a sequence of bytes or characters.

String literal

A sequence of characters contained within double quotes.

subclass

A class that extends another class, either directly (by using the keyword extends in its class declaration) or indirectly. All Java classes are subclasses of java.lang.Object. Sometimes subclasses are called child classes.

super

Java keyword used to call the constructor with the matching parameter list of a parent class.

superclass

A class from which another class is derived. Also called the parent class. java.lang.Object is the superclass of all Java classes.

Swing

A set of lightweight GUI components that run uniformly on any platform that supports the Java Virtual Machine.

switch

Java keyword used to evaluate a variable of integral primitive type (either an int or a type that can be implicitly cast to an int, such as byte, char, short, or int) in order to match its value against values declared in case statements. If the switch value does not match any case, a default statement is executed, if implemented.

synchronized

Java keyword indicating that the method or code block that it contains is guaranteed to be executed by at most one thread at a time. Notice that you can write a perfectly legal (though useless as written here) block of code synchronized (new Object()){}, which demonstrates that you can synchronize access to an object without a method.

T

this

Java keyword referring to the current object instance.

Thread

The java.lang.Thread class defines the behavior of a single thread of control inside the JVM. It can be subclassed to create threads within your application. The other way to create a thread is to write a class that implements the java.lang.Runnable interface.

thread

A thread represents the basic unit of execution in an application. One process may have multiple threads running at once, with each performing a different function. When a thread is executed, local variables are stored in a separate memory space, so that different instances do not confuse or overwrite each other's values.

throw

Java keyword used when the programmer means to generate a new instance of an exception type for handling by the caller. For example: throw new GarageException("Houston, we have a problem:");.

throws

Java keyword used in method declarations to indicate that the method will pass the specified exceptions up to the caller to deal with.

transient

Java keyword indicating that a field should not be considered part of the persistent state of an object, and should therefore not be serialized in an ObjectStream. Does not apply to any element but variables.

try

Java keyword specifying a block of code as one in which an exception might occur, and which consequently must be accompanied by one or more catch blocks to deal with the exception.

U

unary

A unary operator is one that affects only a single operand. For example, + and - are unary operators used to signal the positive or negative value of an integer.

Unicode

Unicode defines a unique number for every character in every language, on every platform. It is the 16-bit character set that serves as the official implementation of ISO10646, and is capable of representing characters from many languages. Although the standard Latin character set used in English requires only 7 bits to represent, more complex characters, such as those written in Arabic, Japanese, Hebrew, Georgian, Greek, and Korean require 16 bits. Unicode is used not only in Java, but also in XML, LDAP, CORBA, and others.

V

Virtual Machine

The Java Virtual Machine serves as the main interpreter for Java bytecodes, and consists of a bytecode instruction set, a set of registers, a stack, a heap, and a method storage area.

Visibility

The level of access that methods and instance variables expose to other classes and packages. You define visibility using the modifiers `public`, `protected`, and `private` (note that there is also `default` visibility, though you don't write `default` explicitly).

void

Java keyword used to indicate in a method declaration that the method does not return any value. Methods returning `void` return implicitly when they complete the final statement. Note that this does *not* mean that use of the `return` keyword is mutually exclusive with use of the `void` return type—that is, the following is perfectly legal (but probably useless): `public void dumbMethod(){ return; }`.

volatile

A modifier indicating that a variable can be accessed and modified asynchronously by concurrent threads. For example: `volatile int myNumber;`.

W

widening conversion

What happens when a value of one type is converted to a wider type—that is, a type with a wider range of possible values. Converting from a short to an int is a widening conversion; converting from a char to a byte is not. The Java runtime knows that your variable couldn't suffer any loss of data by putting its value in a container bigger than the one it already has. For this reason, widening conversions happen automatically. You can lose data when performing a widening conversion that is done automatically. Numerical range is the important thing. Example (float's numerical range encompasses long's, but long (64-bits) can hold more digits than float (32-bits)):

```
public class ConversionTest {

public static void main(String[] args) {

    float a;

    long b = 4203020102023324L;

    a = b;

    System.out.println("float: " + a);

    System.out.println("long: " + b);

    }}
```

Output:

```
float: 4.2030201E15

long: 4203020102023324
```

wildcard

Used in regular expressions to mean any character. In generics, the wildcard is represented as ?. This is used in a generic method declaration to indicate an unknown type. The wildcard can be type-bounded by either upper or lower limits.

See also **[bounds]**

wrapper class

The primitive wrapper classes in the java.lang package correspond to each of the eight primitive variable types: the Boolean primitive wrapper class is java.lang.Boolean; the char wrapper class is Character, and so forth. The purpose of the primitive wrappers is to provide utility methods (such as Double.parseDouble()), constants (such as Boolean.TRUE and Float.POSITIVE_INFINITY), and object encapsulation for primitives.

PREV < Day Day Up >

JAVA GARAGE

By Eben Hewitt

Publisher: Prentice Hall PTR
Pub Date: August 12, 2004
ISBN: 0-321-24623-3
Pages: 480

Enter your *Java Garage*... where you do your work, not somebody else's. It's where you experiment, escape, tinker, and ultimately succeed.

Java Garage is not your typical Java book. If you're tired of monotonous "feature walks" and dull tutorials, put down those other Java books and pick up *Java Garage*. Java guru Eben Hewitt takes a fresh look at this popular programming language, providing the insight and guidance to turn the regular programmer into a master. The style is straightforward, thought-provoking and occasionally irreverent.

You'll learn the best ways to program with everything that matters: J2SE 5.0 classes, inheritance, interfaces, type conversions, event handling, exceptions, file I/O, multithreading, inner classes, Swing, JARs, etc. Hewitt provides real working code and instructions for making usable applications that you can exploit and incorporate into your own personal projects with ease. Need answers quickly? The book also includes FAQs for speedy reference and a glossary on steroids that gives you the context, not just the definition.

With *Java Garage*, you'll learn the best way to create and finish projects with finesse. Think 'zine. Think blog. But, please, do not think of any other Java book you have ever seen.

ACKNOWLEDGMENTS

I am grateful for the help of many people during the making of this book.

First, I must thank my editor, John Neidhart (Iron Chef Garage), for being a standup guy through the many labyrinthine conversations that helped shape this new series. I appreciate your high-availability, ease of maintenance, and extensibility. Here's to more sushi at Bond Street.

I am grateful not only to John, but to the many good and hard-working people at Prentice Hall for the opportunity to work on this series. Thank you to production editor Michael Thurston, and Kelli Brooks who worked very hard to root out the errors in this book and bring it all together. Special thanks to Anthony Gemmellaro and the entire production team for the stunningly cool artwork and interior. I am also grateful to the good people of the Salt River Prima-Maricopa Indian Community; may you live long and prosper.

The efforts of my technical reviewers were tremendous. Thank you to Pawel Zurek, a terrific software developer with an eagle eye. I appreciate very much the careful and insightful comments of J. Benton: it's really fun to talk with you about searching, and planning, and algorithms (all important things in life). Thank you also to Vic Miller for your terrific work in technically reviewing this book. You caught a number of errors; thanks for helping me develop with pleasure.

I am a lucky sonofagun to be married to Alison Brown, my favorite thing in this world. She has also been very helpful in shaping this series. Thank you, Alison, for your smashing ideas, perseverance, and hard work—and for helping keep me monotonously on-message.

Eben Hewitt
July 2004
Scottsdale, Arizona

ABOUT THE AUTHOR

- Table of Contents
- Index

JAVA GARAGE

By Eben Hewitt

Publisher: Prentice Hall PTR
Pub Date: August 12, 2004
ISBN: 0-321-24623-3
Pages: 480