Learn Java the Hard Way

Graham Mitchell

Version 20130720.004

Version 20130720.004

10 09 08 07 06 05 XYZABC 04 03 02 01

Preface

I have been teaching beginners how to code for over fifteen years. More than 2,000 students have taken my classes and left knowing how to write simple programs that work. Some learned how to do only a little and others gained incredible skill over the course of just a few years.

I believe that nearly anyone can teach a child prodigy how to code. "I taught my 9-year-old daughter to code, and she made her first Android app six weeks later!" If you are the child prodigy, this book is not written for you.

I have also come to believe that there is no substitute for writing small programs. So that's what you will do in this book. You will type in small programs and run them.

"The best way to learn is to do."

– P.R. Halmos

Contents

- Exercise 46: Nested For Loops
- Exercise 47: Generating and Filtering Values
- Exercise 48: Arrays - Many Values in a Single Variable
- Exercise 49: Finding Things in an Array
- Exercise 50: Saying Something Is NOT in an Array
- Exercise 51: Arrays Without Foreach Loops
- Exercise 52: Lowest Temperature
- Exercise 53: Mailing Addresses (Records)
- Exercise 54: Records from a File
- Exercise 55: An Array of Records
- Exercise 56: Array of Records from a File (Temperatures Revisited)
- Exercise 57: A Deck of Playing Cards
- Exercise 58: Final Project - Text Adventure Game

Introduction: Java

Java is not a language for beginners. I am convinced that most "beginner" Java books only work on people who already know how to code or who are prodigies.

I can teach you Java, even if you have never programmed before and even if you are not a genius. But I am going to have to cheat a bit.

What I will teach you *is* Java. But it is not *all* of Java. I have to leave parts out because you're not ready for them. If you think you *are* ready for the more complex parts of Java, then 1) you're wrong, and 2) buy a different book. There are a great many books on the market that will throw all the complexity Java has to offer, faster than you can handle it.

In particular, I have one *huge* omission: I am going to avoid the topic of Object-Oriented Programming (OOP). I'm pretty sure that uncomfortable beginners can't learn how to code well and also learn object-oriented programming at the same time. I have almost never seen it work.

I plan to write a follow-up book that *will* cover Object-Oriented Programming and the more complex parts of Java. But you should finish this book first. I have been teaching students to program for many many years, and I have never had a student come visit me from college and say "I wish you had spent less time on the fundamentals."

What You Will Learn

- How to install the Java compiler and a text editor to write programs with.
- How to create, compile and run your first Java program.
- Variables and getting input from the user and from files.
- Making decisions with if statements
- Loops
- Arrays
- Records

In the final chapter you'll write a not-so-simple text-based adventure game with levels loaded from text files. You should also be able to write a text-based card game like Hearts or Spades.

What You Will Not *Learn*

- Graphics
- Object-oriented programming
- How to make an Android app

I like graphics, and they're not hard in Java compared to, say, C++, but I can't cover everything and teach the basics well, so something had to go.

Object-oriented programming has no place in an introductory book, in my opinion.

Android apps are pretty complex, and if you're a beginner, an app is way beyond your ability. Nothing in this book will *hurt* your chances of making an app, though, and the kinder, gentler pace may keep you going when other books would frustrate you into quitting.

Also, I hope to write two more books after this one. My second book will cover graphics and object-oriented programming in Java. My third book will cover making a simple Android app, assuming you have finished working through the first two books.

How to Read This Book

Although I have provided a zipfile containing the source code for all the exercises in the book, you should type them in.

For each exercise, type in the code. Yourself, by hand. How are you going to learn otherwise? None of my former students ever became great at programming by merely reading others' code.

Work the study drills, if there are any. Do the challenge assignments, if provided. And by the end you will be able to code, at least a little.

License

Exercise 0: The Setup

This exercise has no code but **do not skip it**. It will help you to get a decent text editor installed and to install the Java Development Kit (JDK). If you do not do both of these things, you will not be able to do any of the other exercises in the book. You should follow these instructions as exactly as possible.

Warning!

This exercise requires you to do things in a terminal window (also called a "shell", "console" or "command prompt". If you have no experience with a terminal window, then you might need to go learn that first.

Zed Shaw's excellent Command Line Crash Course is at http://cli.learncodethehardway.org/book/ and will teach you how to use PowerShell on Windows or the Terminal on OS X or "bash" on Linux.

Mac OS X

To complete this exercise, complete the following tasks:

1. Go to http://www.barebones.com/products/textwrangler/ with your web browser. Download the TextWrangler text editor and install it.
2. Put TextWrangler in your dock so you can reach it easily.
3. Find a program called "Terminal". (Search for it if you have to.)
4. Put your Terminal in your dock as well.
5. Launch the Terminal.
6. In the Terminal program, type javac -version and press RETURN. You should see a response like javac 1.7.0_04. It is okay if the number after javac is not exactly the same as long as it is 1.6 or greater. If you get an error message, however, you may need to install the JDK.
7. After this, you should be back at a prompt.
8. Learn how to create a folder (make a directory) from the Terminal. Make a directory so that you can put all your code from this book in it.
9. Learn how to change into this new directory from the Terminal. Change into it.
10. Use your text editor (TextWrangler) to create a file called test.txt and save it into the directory you just created.
11. Go back to the Terminal using only the keyboard to switch windows.
12. Back in Terminal, see if you can list the contents of the directory to see your newly created file.

OS X: What You Should See

I don't currently have access to a Mac, so here is Zed following the steps above on his computer in Terminal. Your computer would be different, so see if you can figure out all the differences between what he did and what you should do.

```
Last login: Fri Jul 19 00:56:54 on ttys001
```

```
~ $ javac -version
javac 1.6.22
~ $ mkdir javacode
~ $ cd javacode
javacode $ ls
# ... Use TextWrangler here to edit test.txt....
javacode $ ls
test.txt
javacode $
```

Windows

1. Go to http://notepad-plus-plus.org/ with your web browser, get the Notepad++ text editor, and install it. You do not need to be an administrator to do this.
13. Make sure you can get to Notepad++ easily by putting it on your desktop and/or in Quick Launch. Both options are available during setup.
14. Run PowerShell from the Start menu. Search for it and you can just hit Enter to run it.
15. Make a shortcut to PowerShell on your desktop and/or Quick Launch for your convenience.
16. Go to http://www.oracle.com/technetwork/java/javase/downloads/ with your web browser.
    A. Click the big "Java" button on the left near the top to download the Java Platform (JDK) 7u25. Clicking this will take you to a different page.
    B. On this page you will have to accept the license agreement and then choose the "Windows x86" version near the bottom of the list. Download the file.
    C. Once downloaded, run jdk-7u25-windows-i586.exe to install it. After you click "Next >" the very first time you will see a screen that says Install to: C:\Program Files (x86)\Java\jdk1.7.0_25\ or something similar. Make a note of this location; you will need it soon.
17. Once the JDK is installed you will need to find out the *exact* name of the folder where it was installed. Look on the C: drive inside the Program Files folder or the C:\Program Files (x86) folder if you have one. You are looking for a folder called Java. Inside that is a folder called jdk1.7.0_25 that has a folder called bin inside it. The folder name *must* have jdk1.7 in it; jre7 is not the same. Make sure there's a bin folder.
18. Once you are in this folder, you can left-click up in the folder location and it will change to something that looks like C:\Program Files (x86)\Java\jdk1.7.0_25\bin. You can write this down or highlight and right-click to copy it to the clipboard.
19. Once the JDK is installed and you know this location open up your terminal window (PowerShell). In PowerShell, type this:

[Environment]::SetEnvironmentVariable("Path",

"$env:Path;C:\Program Files (x86)\Java\jdk1.7.0_25\bin", "User")

Put it all on one line, though.

If you copied the folder location to the clipboard, then you can type everything up to the $env:Path; and then right-click in the PowerShell window and it should paste the folder name for you. Then you can just finish the line by typing ", "User") and pressing ENTER. If you get an error, you typed something incorrectly. You can press the up arrow to get it back and the left and right arrows to find and fix your mistake, then press ENTER again.

1. Once the setEnvironmentVariable command completes without giving you an error, close the PowerShell window by typing exit at the prompt. If you don't close it the change you just made won't take effect.
20. Launch PowerShell again.
21. Type javac -version at the prompt. You should see a response like javac 1.7.0_25. Congratulations! If you got that to work, the rest of this book ought to be relatively easy.
22. After this, you should be back at a blinking PowerShell prompt.
23. Learn how to create a folder (make a directory) from the terminal window (PowerShell). Make a directory so that you can put all your code from this book in it.
24. Learn how to change into this new directory from the prompt. Change into it.
25. Use your text editor (Notepad++) to create a file called test.txt and save it into the directory you just created.
26. Go back to the terminal using only the keyboard to switch windows.
27. Back in the terminal, see if you can list the contents of the directory to see your newly created file.

Windows: What You Should See

```
Windows PowerShell
Copyright (C) 2009 Microsoft Corporation. All rights reserved.

PS C:\Users\Graham_Mitchell> javac -version
javac 1.7.0_25
PS C:\Users\Graham_Mitchell> mkdir javacode


    Directory: C:\Users\Graham_Mitchell\javacode


Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d----         7/19/2013   7:39 PM          javacode


PS C:\Users\Graham_Mitchell> cd javacode
PS C:\Users\Graham_Mitchell\javacode> ls
PS C:\Users\Graham_Mitchell\javacode>
... Here you would use Notepad++ to make test.txt in javacode ...
PS C:\Users\Graham_Mitchell\javacode> ls


    Directory: C:\Users\Graham_Mitchell\javacode
```

```
Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---         7/19/2013   7:45 PM          4 test.txt


PS C:\Users\Graham_Mitchell\javacode>
```

You will probably see a different prompt and other slight differences but you shouldn’t get any errors and this is the general idea.

Linux

There are a lot of different versions of Linux out there, so I am going to give instructions for the latest version of Ubuntu. If you are running something else, you probably know what you are doing well enough to figure out how to modify the directions for your setup.

1. Use your Linux package manager to install the gedit text editor (Which might just be called "Text Editor".)
28. Make sure you can get to gedit easily by putting it in the Launcher.
29. Run gedit so we can change some of the defaults to be better for programmers:
    D. Open *Preferences* and select the *Editor* tab.
    E. Change Tab width: to 4.
    F. Put a check mark next to "Automatic indentation"
    G. Open the *View* tab and turn on "Display line numbers."
30. Find your Terminal program. It could be called GNOME Terminal, Konsole, or xterm.
31. Put your Terminal in the Launcher as well.
32. Use your Linux package manager to install the Java JDK. I use openjdk-7-jdk, but if you prefer the Oracle one that will work, too.
33. Launch your Terminal if you haven't already.
34. Type javac -version at the prompt. You should see a response like javac 1.7.0_25. If not, make sure the JDK is installed and that the bin folder containing the executable javac is in your PATH.
35. Learn how to create a folder (make a directory) from the terminal. Make a directory so that you can put all your code from this book in it.
36. Learn how to change into this new directory from the prompt. Change into it.
37. Use your text editor (gedit) to create a file called test.txt and save it into the directory you just created.
38. Go back to the terminal using only the keyboard to switch windows. Look it up if you don't know how.
39. Back in the terminal, see if you can list the contents of the directory to see your newly created file.

Linux: What You Should See

```
mitchell@graham-desktop:~$ javac -version
javac 1.7.0_25
mitchell@graham-desktop:~$ mkdir javacode
mitchell@graham-desktop:~$ cd javacode/
mitchell@graham-desktop:~/javacode$ ls
mitchell@graham-desktop:~/javacode$
... Here you would use Notepad++ to make test.txt in javacode ...
mitchell@graham-desktop:~/javacode$ ls
test.txt
mitchell@graham-desktop:~/javacode$
```

You will probably see a different prompt and other slight differences but you

shouldn’t get any errors and this is the general idea.

Warnings for Beginners

You are done with the first exercise. This exercise might have been quite hard for you depending on your familiarity with your computer. If it was difficult and you didn't finish it, go back and take the time to read and study and get through it. Programming requires careful reading and attention to detail.

If a programmer tells you to use vim or emacs or Eclipse, just say "no." These editors are for when you are a better programmer. All you need right now is an editor that lets you put text into a file. We will use gedit, TextWrangler, or Notepad++ (from now on called "the text editor" or "a text editor") because it is simple and the same on all computers. Professional programmers use these text editors so it's good enough for you starting out.

A programmer will eventually tell you to use Mac OS X or Linux. If the programmer likes fonts and typography, he'll tell you to get a Mac OS X computer. If he likes control and has a huge beard, they'll tell you to install Linux. Again, use whatever computer you have right now that works. All you need is an editor, a Terminal, and Java.

Finally, the purpose of this setup is so you can do three things very reliably while you work on the exercises:

- Write exercises using your text editor (gedit on Linux, TextWrangler on OSX, or Notepad++ on Windows).
- Run the exercises you wrote.
- Fix them when they are broken.
- Repeat.

Anything else will only confuse you, so stick to the plan.

Frequently-Asked Questions

Do I have to use this lame text editor? I want to use Eclipse!

Do not use Eclipse. Although it is a nice program it is not for beginners. It is bad for beginners in two ways:

1. It makes you do things that you don't need to worry about right now.
2. It does things for you that you need to learn how to do for yourself first.

So follow my instructions and use a decent text editor and a terminal window. Once you have learned how to code you can use other tools if you want, but not now.

Can I work through this book on my tablet? Or my Chromebook?

Unfortunately not. You can't install the Java development kit (JDK) on either of those machines. You must have some sort of traditional computer.

Exercise 1: An Important Message

In this exercise you will write a working program in Java to display an important message on the screen.

If you are not used to typing detailed instructions for a computer then this could be one of the harder exercises in the book. Computers are very stupid and if you don't get *every* detail right, the computer won't understand your instructions. But if you can get this exercise done and working, then there is a good chance that you will be able to handle every exercise in the book as long as you work on it every day and don't quit.

Open the text editor you installed in Exercise 0 and type the following text into a single file named FirstProg.java. Make sure to match what I have written exactly, including spacing, punctuation, and capitalization.

```
1 public class FirstProg
2 {
3    public static void main( String[] args )
4    {
5       System.out.println( "I am determined to learn how to code." );
6       System.out.println( "Today's date is" );
7    }
8 }
```

I have put line numbers in front of each line, but do not type the line numbers. They are only there so I can talk about the lines. Also, depending on whether or not you have saved the file yet, the different words may not be colored at all. Or if they are colored, they might be different colors than mine. These differences are fine.

I'm going to walk through this line-by-line, just to make sure you typed everything correctly.

The first line starts with the word public followed by a single space then the word class followed by a single space and then the word FirstProg. The 'F' in "First" is capitalized, the 'P' in "Prog" is capitalized. There are only two capital letters in the first line. There are only two spaces.

The second line is just a single character: a "brace". You get it to show up by holding down SHIFT and then pressing the '[' key which is usually to the right of the letter 'P'.

Before I go on to the third line of the program, I should tell you what programmers usually call each funny symbol that appears in this program.

> (and) are called "parentheses" (that's plural). Just one of them is called "a parenthesis", but some people just call them parens ("puh-RENZ"). This one ("(") is sometimes called a "left paren" and the other (")") is called a "right paren" because parentheses usually come in pairs and one is usually to the left of the other. The left parenthesis ("(") is also often called an "open paren" and the right one is called a "close paren" for similar reasons.
>
> There's an open paren on line 3 and a close paren, too, and no

other parentheses in the whole file.

[and] are called “brackets”, but many programmers call them “square brackets” to make sure there’s no confusion. In Java, parentheses and square brackets are *not* interchangeable. Brackets come in pairs and they are called “left bracket” or “open bracket” and “right bracket” or “close bracket”.

There’s an open and close square bracket right next to each other on line 3.

{ and } are called “braces”, and some programmers call them “curly braces”. These also always come in pairs of left and right curly braces / open and close braces.

" is called a “quotation mark”, often just abbreviated “quote”. In Java, these always come in pairs. The first one in a pair is usually called an “open quote” and the second one is a “close quote” even though it’s the exact same character in both places. But the first quote serves to begin something and the second one ends that thing.

' is technically an “apostrophe”, but almost all programmers call them “single quotes”. For this reason a quotation mark is often called a “double quote”. In some programming languages, single quotes and double quotes are interchangeable, but not in Java. Java *does* use single quotes sometimes, but they’re going to be pretty rare in this book.

. is technically a “period”, but almost all programmers just say “dot”. They are used a lot in programming languages, and they are usually used as separators instead of “enders”, so we don’t call them periods.

There are four dots in this program and one period.

; is called a “semicolon”. It’s between the letter ‘L’ and the quote on the keyboard. Java uses a *lot* of semicolons although there are only two of them in this program: one on the end of line 5 and another at the end of line 6.

: is called a “colon”. You get it by holding SHIFT and typing a semicolon. Java does use colons, but they’re very rare.

Finally, < is a “less-than sign” and > is a “greater-than sign”, but sometimes they are used sort-of like braces or brackets. When they are used this way, they’re usually called “angle brackets”. Java uses angle brackets, but you won’t see them used in this book.

Okay, so back to the line-by-line. You have already typed the first two lines correctly.

You should start the third line by pressing TAB one time. Your cursor will move over several spaces (probably 4 or 8). Then type the word public again, one space, the word static, one space, the word void, one space, the word main followed by an open paren (no space between the word “main” and the paren).

After the paren there's one space, the word String with a capital 'S', an open and close square bracket right next to each other, one space, the word args, one space and finally a close parenthesis.

So line three starts with a tab, has a total of six spaces, and only the 'S' in "String" is capitalized. Whew.

On the fourth line, your text editor may have already started your cursor directly underneath the 'p' in "public". If it did not do that, then you'll have to start line 4 by pressing TAB yourself. Then just type another open curly brace and that's it.

The fifth line should start with *two* tabs. Then type the word System with a capital 'S', then a dot (period), then the word out, another dot, the word println (pronounced "PrintLine" even though there's no 'i' or 'e' at the end), an open paren, a space, a quotation mark (open quote), the sentence I am determined to learn how to code. (the sentence ends with a period), then a close quote, a space, a close paren and a semicolon.

So line 5 has two tabs, nine spaces, two dots (and a period), an open and close quote, an open and close paren, and only two capital letters.

Line 6 is nearly identical to line 5 except that the sentence says Today's date is instead of the determination sentence.

Line 7 starts with only one tab. If your text editor put two tabs in there for you, you should be able to get rid of the extra tab by pressing BACKSPACE one time. Then after the tab there's a close curly brace.

Finally, line 8 has no tabs and one more close curly brace. You can press ENTER after line 8 or not: Java doesn't care.

Notice that we have two open curly braces and two close curly braces in the file. Three open parens and three close parens. Two "open quotes" and two "close quotes". One open square bracket and one close square bracket. This will always be true.

Also notice that every time we did an open curly brace, the line(s) below it had more tabs at the beginning, and the lines below close curly braces had fewer tabs.

Okay, now save this (if you haven't already) as FirstProg.java and save it in the "code" folder you created in Exercise 0.

Make sure the file name matches mine exactly: the 'F' in "First" is capitalized, the 'P' in "Prog" is capitalized, and everything else is lowercase. And there should be no spaces in the file name. Java will refuse to run any program with a space in the file name. Also make sure the filename ends in .java and not .txt.

Compiling Your First Program

Now that the program has been written and hopefully contains no mistakes (we'll see soon enough), launch your Terminal (or PowerShell) and change into the directory where the code is saved.

Do a directory listing to make sure the Java file is there. On my computer, it

looks like this:

```
mitchell@graham-desktop:~$ cd javacode/
mitchell@graham-desktop:~/javacode$ ls
FirstProg.java  test.txt
mitchell@graham-desktop:~/javacode$
```

In the future, since your terminal probably doesn't look like mine, I am going to abbreviate the prompt like this:

```
$ ls
FirstProg.java  test.txt
$
```

That way it will be less confusing, since there is less "wrong" stuff to ignore and you only have to look at what you should type and what you should see.

Now, we have typed a list of commands in a programming language called Java. But the computer cannot execute our commands directly in this form. We have to give this file to a "compiler", which is a program that will translate our instructions into something more like ones and zeros that the computer can execute. In Java that ones and zeros file is called "bytecode". So we are going to run the Java compiler program to "compile" our Java source code into a bytecode file that we will be able to execute.

The Java compiler is named javac (the 'c' is for "compiler"), and we run it like so:

```
$ javac FirstProg.java
$
```

If you have extraordinary attention to detail and did everything that I told you, this command will take a second to run, and then the prompt will just pop back up with no message. If you made some sort of mistake, you will see an error like this:

```
$ javac FirstProg.java
FirstProg.java:8: error: reached end of file while parsing
}
 ^
1 error
$
```

Don't worry too much about the particular error message. When it gets confused, the compiler tries to guess about what you might have done wrong. Unfortunately, the guesses are designed for expert programmers, so it usually doesn't guess well for beginner-type mistakes.

Here is an example of a different error message you might get:

```
$ javac FirstProg.java
FirstProg.java:5: error: ';' expected
            System.out.println( "I am determined to learn how to code." ):
                                                                         ^
1 error
$
```

In this case, the compiler is actually right: the error is on line 5 and the specific error is that a semicolon was expected (';' expected). (The line ends with a colon (:) but it ought to be a semicolon (;).

Here's one more:

```
$ javac FirstProg.java
FirstProg.java:1: error: class Firstprog is public, should be declared in a file named Firstprog.java
public class Firstprog
^
1 error
$
```

This time is it a capitalization error. The code says public class Firstprog (note the lowercase 'p') but the filename is FirstProg.java. Because they don't match exactly, capitalization and all, the compiler gets confused and bails out.

So if you have any error messages, fix them, then *save* your code, go back to the terminal and compile again.

Warning!

> *If you make a change to the code in your text editor, you must* save *the file before attempting to re-compile it. If you don't save the changes, you will still be compiling the old version of the code that was saved previously, even if the code in your text editor is correct.*

Eventually you should get it right and it will compile with no errors and no message of any kind. Do a directory listing and you should see the bytecode file has appeared in the folder:

```
$ javac FirstProg.java
$ ls
FirstProg.class  FirstProg.java  test.txt
$
```

Now that we have a valid bytecode file we can run it (or "execute" it) by running it through the Java Virtual Machine (JVM) program called java:

What You Should See

```
$ java FirstProg
```

```
I am determined to learn how to code.
Today's date is
$
```

Note that the command you type is java FirstProg, not java FirstProg.java or even java FirstProg.class.

Are you stoked? You just wrote your first Java program and ran it! If you made it this far, then you almost certainly have what it takes to finish the book as long as you work on it every day and don't quit.

Study Drills

After most of the exercises, I will list some additional tasks you should try after typing up the code and getting it to compile and run. Some study drills will be fairly simple and some will be more challenging, but you should always give them a shot.

1. Change what is inside the quotes on line 6 to include today's date. Save the file once you have made your changes, compile the file and run it again.
40. Change what is inside the quotes on line 5 to have the computer display your name.

What You Should See After Completing the Study Drills

```
$ java FirstProg
I, Graham Mitchell, am determined to learn how to code.
Today's date is Friday, July 19, 2013.
$
```

Exercise 2: More Printing

Okay, now that we've gotten that first, hard assignment out of the way, we'll do another. The nice thing is that in this one, we still have a lot of the setup code (which will be nearly the same every time), but the ratio of set up to "useful" code is much better.

Type the following text into a single file named LetterToYourself.java. Make sure to match what I have written exactly, including spacing, punctuation, and capitalization.

```
1 public class LetterToYourself
 2 {
 3    public static void main( String[] args )
 4    {
 5       System.out.println( "+------------------------------------------------------------+" );
 6       System.out.println( "|                                          #### |" );
 7       System.out.println( "|                                          #### |" );
 8       System.out.println( "|                                          #### |" );
 9       System.out.println( "|                                             |" );
10        System.out.println( "|                                             |" );
11        System.out.println( "|                      Bill Gates             |" );
12        System.out.println( "|                      1 Microsoft Way        |" );
13        System.out.println( "|                      Redmond, WA 98104      |" );
14        System.out.println( "|                                             |" );
15        System.out.println( "+------------------------------------------------------------+" );
16    }
17 }
```

Notice that the first line is the same as the previous assignment, except that the name of the class is now LetterToYourself instead of FirstProg. Also note that the name of the file you're putting things into is LetterToYourself.java instead of FirstProg.java. This is not a coincidence.

In Java, each file can contain only one public class, and the name of the public class *must* match the file name (capitalization and all) except that the file name ends in .java and the public class name does not.

So, what does "public class" *mean* in Java? I'll tell you when you're older. Seriously, trying to go into that kind of detail up front is why most "beginner" programming books are bad for actual beginners. So don't worry about it. Just type it for now. (Unfortunately, there's going to be a lot of that.)

You will probably notice that the second, third and fourth lines of this program are *exactly* the same as the previous assignment. There are no differences whatsoever.

Then, after the second open brace, there are eleven printing statements. They are all identical except for what is between the quotation marks. The little vertical bar ("|") is called the "pipe" character, and you can type it using Shift + backslash ("\"). Assuming you are using a normal US keyboard, the backslash key is located between the Backspace and Enter keys.

Once everything is typed in and saved as LetterToYourself.java, you can compile and run it the same way you did the previous assignment. Switch to your terminal window, change the directory into the one where you are saving your code and type this to compile it:

```
$ javac LetterToYourself.java
```

If you are extremely good with annoying details and fortunate, you will have no errors and the javac command will complete without saying anything at all. Probably, you will have errors. If so, go back and compare what you type with what I wrote very carefully. Eventually you will discover your error(s). Fix them, save the file again, and try compiling again.

Once it compiles with no errors, you can run it like before:

```
$ javac LetterToYourself.java
$ java LetterToYourself
+-----------------------------------------------------+
|                                  #### |
|                                  #### |
|                                  #### |
|                                     |
|                                     |
|                     Bill Gates      |
|                     1 Microsoft Way     |
|                     Redmond, WA 98104       |
|                                     |
+-----------------------------------------------------+
```

Two programs done. Nice! What you have accomplished so far is *not* easy, and anyone that thinks it is has a *lot* of experience and has forgotten what it is like to try this stuff for the first time. Don't quit! Work a little every day and this *will* get easier.

Study Drills

1. This file is called LetterToYourself.java not LetterToBillGates.java! Go back to your text editor and change the name and address from Bill Gates' address at Microsoft to your *own* name and address. Then save it, compile it and run it again.

Common Student Questions

Do I have to use my real address?

Of course not. But make sure your fake address takes up three lines.

Why doesn't my letter line up when I run the program?!? Everything looks perfect in the code!

You probably used a mixture of tabs and spaces between the quotes in your println() statements. Many text editors will only move the cursor 4 spaces when you press TAB. But when your program runs, any tabs embedded inside

the quotes will take up 8 spaces, not 4. If you delete ALL the tabs between the quotes and replace them with spaces, things should look the same in your code and when you run the program.

Exercise 3: Printing Choices

Java has two common commands used to display things on the screen. So far we have only looked at println(), but print() is sometimes used, too. This exercise will demonstrate the difference.

Type the following code into a single file. By reading the code, could you guess that the file must be called PrintingChoices.java? In future assignments, I may not tell you what to name the Java file.

```
1 public class PrintingChoices
 2 {
 3     public static void main( String[] args )
 4     {
 5         System.out.println( "Alpha" );
 6         System.out.println( "Bravo" );
 7
 8         System.out.println( "Charlie" );
 9         System.out.println( "Delta" );
10          System.out.println();
11
 12          System.out.print( "Echo" );
13          System.out.print( "Foxtrot" );
14
 15          System.out.println( "Golf" );
16          System.out.print( "Hotel" );
17          System.out.println();
18          System.out.println( "India" );
19
 20          System.out.println();
21          System.out.println( "This" + " " + "is" + " " + "a" + " test." );
22
 23     }
24 }
```

When you run it, this is what you should see.

```
Alpha
Bravo
Charlie
Delta

EchoFoxtrotGolf
Hotel
India

This is a test.
```

Can you figure out the difference?

Both print() and println() display on the screen whatever is between the quotation marks. But println() moves to a new line after finishing printing, and print() does not: it displays and then leaves the cursor right at the end of the line so that the following printing statement picks up from that same position on the line.

You will also notice (on line 10) that we can have a println() statement with *nothing* between the parentheses. No quotation marks or anything. That statement instructs the computer to print *nothing*, and then move the cursor to the beginning of the next line.

You may also notice that this program has some lines with nothing on them (lines 7, 11, 14, 19 and 22). On the very first exercise, when I wrote that you must "match what I have written exactly, including spacing, punctuation, and capitalization", I wasn't *quite* being honest. Extra blank lines in your code are ignored by the Java compiler. You can put them in or remove them, and the program will work exactly the same.

My students often accuse me of being "full of lies." This is true. I have learned the hard way that when students are just learning something as hard as programming, telling them the truth will confuse them too much. So I often over-simplify what I say, even when that makes it technically inaccurate.

If you already know how to program, and my "lies" offend you, then this book will be difficult to read. But for those that are just learning, I assure you that you want me to simplify things at first. I promise I'll reveal the truth eventually.

Anyway, on line 21, I did one more new thing. So far you have only been printing a single thing inside quotation marks. But it is perfectly fine to print more than one thing, as long as you combine those things before printing.

So on line 21, I have six Strings[1] in quotation marks: the word "this", a space, the word "is", a space, the word "a", and finally a space followed by "test" followed by a period. There is a plus sign ("+") between each of those six Strings, so there are a total of five plus signs on line 21. When you put a plus sign between Strings, Java adds[2] them together to make one long thing-in-quotation marks, and then displays that all at once.

If you have an error in your code, it is probably on line 21. Remembering to start and stop all the quotes correctly and getting all those details right is tricky.

Today's lesson was hopefully *relatively* easy. Don't worry, I'll make up for it on the next one.

[11]What is a "String"? A bunch of characters (letters, numbers, symbols) between a pair of quotation marks. I'll explain more later.

[22]Technically combining smaller words to make a larger one is called "concatenation", not "adding". Java concatenates the Strings together.

Exercise 4: Escape Sequences and Comments

Have you thought about what might happen if we wanted to display a quotation mark on the screen? Since everything we want to display is contained between quotation marks in the println() statement, putting a quote *inside* the quotes would be a problem.

Most programming languages allow for "escape sequences", where you signal with some sort of escape character that the next character you see shouldn't be handled in the normal way.

The following (evil) code demonstrates a number of Java's escape sequences. Call it EscapeSequences.java.

```
1 public class EscapeSequences
 2 {
 3    public static void main( String[] args )
 4    {
 5       // Initial version created using FIGlet, font "Big Money", oriented southwest
 6
 7       System.out.print( "\t    ______\n\t   /     |\n\t   JJJJJ |" );
 8       System.out.println( "  _______  __    __ _______" );
 9       System.out.println( "\t      JJ | /      \\ /  \\   /  |/      \\" );
10       System.out.println( "\t __   JJ | aaaaaa  |\"\"  \\ /\"\"/ aaaaaa  |" );
11       System.out.println( "\t/  |  JJ | /    aa | \"\"  /\"\"/  /    aa |" );
12       System.out.println( "\tJJ \\__JJ |/aaaaaaa |  \"\" \"\"/  /aaaaaaa |" );
13       System.out.println( "\tJJ    JJ/ aa    aa |   \"\"\"/   aa    aa |" );
14       System.out.println( "\t JJJJJJ/   aaaaaaa/     \"/     aaaaaaa/" );
15    }
16 }
```

When you run it, this is what you should see.

```
______
       /     |
       JJJJJ |  _______   __     __  _______
         JJ | /      \ /  \   /  |/      \
      __   JJ | aaaaaa  |""  \ /""/ aaaaaa  |
     /  |  JJ | /    aa | ""  /""/  /    aa |
     JJ \__JJ |/aaaaaaa |  "" ""/  /aaaaaaa |
     JJ    JJ/ aa    aa |   """/   aa    aa |
      JJJJJJ/   aaaaaaa/     "/     aaaaaaa/
```

Java's escape character is a backslash ("\"), which is the same key you press to make a pipe ("|") show up but without holding Shift. All escape sequences in Java must be inside a set of quotes.

\" represents a quotation mark.

\t is a tab; it is the same as if you pressed the Tab key while typing your code. It probably seems more complicated now because you've never seen it before, but when you're reading someone else's code a \t inside the quotes is less ambiguous than a bunch of blank spaces that might be spaces or might be a tab.

\n is a newline. When printing it will cause the output to move down to the beginning of the next line before continuing printing.

\\ is how you display a backslash.

On line 5 you will notice that the line begins with two slashes (or "forward slashes", if you insist). This marks the line as a "comment", which is in the program for the human programmers' benefit. Comments are totally ignored by the computer.

In fact, the two slashes to mark a comment don't have to be at the beginning of the line; we could write something like this:

```
System.out.println( "A" ); // prints an 'A' on the screen
```

…and it would totally work. Everything from the two slashes to the end of that line is ignored by the compiler.

> (That's not a very good comment, though; any programmer who knows Java already knows what that line of code does. In general you should put comments explaining why the code is there, not what the code does. You'll get better at writing good comments as you get better at coding in general.)

Anyway, this one was a tough one, so no Study Drills this time. The next exercise will feature something new and return to normal difficulty.

Exercise 5: Saving Information in Variables

Programs would be pretty boring if the only thing you could do was print things on the screen. We would like our programs to be interactive.

Unfortunately, interactivity requires several different concepts working together, and explaining them all at once might be confusing. So I am going to cover them one at a time.

First up: variables! If you have ever taken an Algebra class, you are familiar with the concept of variables in mathematics. Programming languages have variables, too, and the basic concept is the same:

> "A variable is a name that refers to a location that holds a value."

Variables in Java have four major differences from math variables:

1. Variable names can be more than one letter long.
41. Variables can hold more than just numbers; they can hold words.
42. You have to choose what type of values the variable will hold when the variable is first created.
43. The value of a variable (but not its type) can change throughout the program. The variable score might start out with a value of 0, but by the end of the program, score might hold the value 413500 instead.

Okay, enough discussion. Let's get to the code! I'm not going to tell you what the name of the file is supposed to be. You'll have to figure it out for yourself.

```
1 public class CreatingVariables
 2 {
 3    public static void main( String[] args )
 4    {
 5       int x, y, age;
 6       double seconds, e, checking;
 7       String firstName, last_name, title;
 8
 9       x = 10;
10        y = 400;
11        age = 39;
12
 13        seconds = 4.71;
14        e = 2.71828182845904523536;
15        checking = 1.89;
16
 17        firstName = "Graham";
18        last_name = "Mitchell";
19        title = "Mr.";
20
 21        System.out.println( "The variable x contains " + x );
22        System.out.println( "The value " + y + " is stored in the variable y." );
23        System.out.println( "The experiment completed in " + seconds + " seconds." );
24        System.out.println( "My favorite irrational number is Euler's constant: " + e );
25        System.out.println( "Hopefully your balance is more than $" + checking + "!" );
26        System.out.println( "My full name is " + title + " " + firstName + last_name );
27    }
28 }
```

What You Should See

```
The variable x contains 10
The value 400 is stored in the variable y.
```

```
The experiment completed in 4.71 seconds.
My favorite irrational number is Euler's constant: 2.718281828459045
Hopefully your balance is more than $1.89!
My full name is Mr. GrahamMitchell
```

On lines 5 through 7 we declare[3] nine variables. The first three are named *x*, *y*, and *age*. All three of these variables are "integers", which is the type of variable that can hold a value between ± two billion.

A variable which is an integer could hold the value 10. It could hold the value -8192. An integer variable can hold 123456789. It could *not* hold the value 3.14 because that has a fractional part. An integer variable could *not* hold the value 10000000000 because ten billion is too big.

On line 6 we declare variables named *seconds*, *e*, and *checking*. These three variables are "doubles", which is the type of variable that can hold a number that might have a fractional part.

A variable which is a double could hold the value 4.71. It could hold the value -8192. (It *may* have a fractional part but doesn't have to.) It can pretty much hold *any* value between ± 1.79769×10^{308} and 4.94065×10^{-324}.

However, doubles have limited precision. Notice that on line 14 I store the value 2.71828182845904523536 into the variable named *e*, but when I print out that value on line 24, only 2.718281828459045 comes out. Doubles do not have enough significant figures to hold the value 2.71828182845904523536 precisely. Integers have perfect precision, but can only hold whole numbers and can't hold huge huge values.

The last type of variable we are going to look at in this exercise is the String. On line 7 we declare three String variables: *firstName*, *last_name* and *title*. String variables can hold words and phrases; the name is short for "string of characters".

On lines 9 through 11 we initialize[4] the three integer values. The value 10 is stored into *x*. Before this point, the variable *x* exists, but its value is undefined. 400 is stored into *y* and 39 is stored into the variable *age*.

Lines 13 through 15 give initial values to the three double variables, and lines 17 through 19 initialize the three String variables. Then lines 21 through 26 display the values of those variables on the screen. Notice that the variable names are not surrounded by quotes.

I know that it doesn't make sense to use variables for a program like this, but soon everything will become clear.

---

[3] [3]declare - to tell the program the name (or "identifier") and type of a variable.

[4] [4]initialize - to give a variable its first (or "initial") value.

Exercise 6: Mathematical Operations

Now that we know how to declare and initialize variables in Java, we can do some mathematics with those variables.

```
1 public class MathOperations
 2 {
 3    public static void main( String[] args )
 4    {
 5       int a, b, c, d, e, f, g;
 6       double x, y, z;
 7       String one, two, both;
 8
 9       a = 10;
10        b = 27;
11        System.out.println( "a is " + a + ", b is " + b );
12
 13        c = a + b;
14        System.out.println( "a+b is " + c );
15        d = a - b;
16        System.out.println( "a-b is " + d );
17        e = a+b*3;
18        System.out.println( "a+b*3 is " + e );
19        f = b / 2;
20        System.out.println( "b/2 is " + f );
21        g = b % 10;
22        System.out.println( "b%10 is " + g );
23
 24        x = 1.1;
25        System.out.println( "\nx is " + x );
26        y = x*x;
27        System.out.println( "x*x is " + y );
28        z = b / 2;
29        System.out.println( "b/2 is " + z );
30        System.out.println();
31
 32        one = "dog";
33        two = "house";
34        both = one + two;
35        System.out.println( both );
36    }
37 }
```

What You Should See

```
a is 10, b is 27
a+b is 37
a-b is -17
a+b*3 is 91
b/2 is 13
b%10 is 7

x is 1.1
x*x is 1.2100000000000002
b/2 is 13.0
```

```
doghouse
```

The plus sign (+) will add two integers or two doubles together, or one integer and one double (in either order). With two Strings (like on line 34) it will concatenate[5] the two Strings together.

The minus sign (-) will subtract one number from another. Just like addition, it works with two integers, two doubles, or one integer and one double (in either order).

An asterisk (*) is used to represent multiplication. You can also see on line 17 that Java knows about the correct order of operations. *b* is multiplied by 3 giving 81 and then *a* is added.

A slash (/) is used for division. Notice that when an integer is divided by another integer (like on line 19) the result is also an integer and not a double.

The percent sign (%) is used to mean 'modulus', which is essentially the remainder left over after dividing. On line 21, *b* is divided by 10 and the remainder (7) is stored into the variable *g*.

Common Student Questions

- Why is 1.1 times 1.1 equal to 1.2100000000000002 instead of just 1.21? Why is 0.333333 + 0.666666 equal to 0.999999 instead of 1.0? Sometimes with math we get repeating decimals, and most computers convert numbers into binary before working with them. It turns out that 1.1 is a repeating decimal in binary.

  Remember what I said in the last exercise: the problem with doubles is limited precision. You will mostly be able to ignore that fact in this book, but I would like you to keep in the back of your mind that double variables sometimes give you values that are *slightly* different than you'd expect.

[55]"concatenate" - to join character Strings end-to-end.

Exercise 7: Getting Input from a Human

Now that we have practiced creating variables for a bit, we are going to look at the other part of interactive programs: letting the human who is running our program have a chance to type something.

Go ahead and type this up, but notice that the first line in the program is *not* the public class line. This time we start with an "import" statement.

Not every program needs to get interactive input from a human using the keyboard, so this is not part of the core of the Java language. Just like a Formula 1 race car does not include an air conditioner, programming languages usually have a small core and then lots of optional libraries[6] that can be included if desired.

```
1 import java.util.Scanner;
 2
 3 public class ForgetfulMachine
 4 {
 5    public static void main( String[] args )
 6    {
 7       Scanner keyboard = new Scanner(System.in);
 8
 9       System.out.println( "What city is the capital of France?" );
10        keyboard.next();
11
 12        System.out.println( "What is 6 multiplied by 7?" );
13       keyboard.nextInt();
14
 15        System.out.println( "What is your favorite number between 0.0 and 1.0?" );
16       keyboard.nextDouble();
17
 18        System.out.println( "Is there anything else you would like to tell me?" );
19       keyboard.next();
20    }
21 }
```

When you first run this program, it will only print the first line:

```
What city is the capital of France?
```

…and then it will blink the cursor at you, waiting for you to type in a word. When I ran the program, I typed the word "Paris", but the program will work the same even if you type a different word.

Then after you type a word and press Enter, the program will continue, printing:

```
What is 6 multiplied by 7?
```

…and so on. Assuming you type in reasonable answers to each question, it will end up looking like this:

What You Should See

[6] library or "module" - a chunk of code that adds extra functionality to a program and which may or may not be included.

```
What city is the capital of France?
Paris
What is 6 multiplied by 7?
42
What is your favorite number between 0.0 and 1.0?
2.3
Is there anything else you would like to tell me?
No, there is not.
```

So let us talk about the code. On line 1 we have an import statement. The library we import is the scanner library java.util.Scanner ("java dot util dot Scanner"). This library contains functionality that allows us to read in information from the keyboard or other places like files or the Internet.

Lines 2 through 7 are hopefully boring. On line 8 we see something else new: we create a "Scanner object" named "keyboard". (It doesn't have to be named "keyboard"; you could use a different word there as long as you use it everywhere in your code.) This Scanner object named keyboard contains abilities we'll call functions or "methods". You must create and name a Scanner object before you can use one.

On line 10 we ask the Scanner object named keyboard to do something for us. We say "Keyboard, run your next() function." The Scanner object will pause the program and wait for the human to type something. Once the human types something and presses Enter, the Scanner object will package it into a String and allow your code to continue.

On line 13 we ask the Scanner object to execute its nextInt() function. This pauses the program, waits for the human to type something and press Enter, then packages it into an integer value (if possible) and continues.

What if the human doesn't type an integer here? Try running the program again and type 41.9 as the answer to the second question.

The program blows up and doesn't run any other statements because 41.9 can *not* be packaged into an integer value: 41.9 is a double. Eventually we will look at ways to handle error-checking for issues like this, but in the meantime, if the human types in something incorrectly which blows up our program, we will blame the human for not following directions and not worry about it.

Line 16 lets the human type in something which the Scanner object will attempt to convert into a double value, and line 19 lets the human type in a String. (Anything can be packaged as a String, including numbers, so this isn't likely to fail.)

Try running the program several more times, noticing when it blows up and when it doesn't.

Exercise 8: Storing the Human's Responses

In the last exercise, you learned how to pause the program and allow the human to type in something. But what happened to what was typed? When you typed in the answer "Paris" for the first question, where did that answer go? Well, it was thrown away right after it was typed because we didn't put any instructions to tell the Scanner object where to store it. So that is the topic of today's lesson.

```
1 import java.util.Scanner;
 2
 3 public class RudeQuestions
 4 {
 5     public static void main( String[] args )
 6     {
 7         String name;
 8         int age;
 9         double weight, income;
10
 11         Scanner keyboard = new Scanner(System.in);
12
 13         System.out.print( "Hello. What is your name? " );
14         name = keyboard.next();
15
 16         System.out.print( "Hi, " + name + "! How old are you? " );
17         age = keyboard.nextInt();
18
 19         System.out.println( "So you're " + age + ", eh? That's not old at all." );
20         System.out.print( "How much do you weigh, " + name + "? " );
21         weight = keyboard.nextDouble();
22
 23         System.out.print( weight + "! Better keep that quiet. Finally, what's your income, " + name + "? " );
24         income = keyboard.nextDouble();
25
 26         System.out.println( "Hopefully that is " + income + " per hour and not per year!" );
27         System.out.println( "Well, thanks for answering my rude questions, " + name + "." );
28     }
29 }
```

(Sorry that line 23 wraps like that. I didn't want to make the font tiny just on account of that one line. Just type it all on one line like usual.)

Just like the last exercise, when you first run this your program will only display the first question and then pause, waiting for a response:

```
Hello. What is your name?
```

Notice that because that first printing statement on line 13 is print() rather than println(), the cursor is left blinking at the end of the line the question is on. If you had used println(), the cursor would blink on the beginning of the next line instead.

What You Should See

```
Hello. What is your name? Brick
Hi, Brick! How old are you? 25
So you're 25, eh? That's not old at all.
How much do you weigh, Brick? 192
```

192.0! Better keep that quiet. Finally, what's your income, Brick? 8.75
Hopefully that is 8.75 per hour and not per year!
Well, thanks for answering my rude questions, Brick.

At the top of the program we declared four variables: one String variable named *name*, one integer variable named *age*, and two double variables named *weight* and *income*.

On line 14 we see the keyboard.next() that we know from the previous exercise will pause the program and let the human type in something it will package up in a String. So now where does the String they type go? In this case, we are storing that value into the String variable named "name". The String value gets stored into a String variable. Nice.

So, assuming you type Brick for your name, the String value "Brick" gets stored into the variable *name* on line 14. This means that on line 16, we can display that value on the screen! That's pretty cool, if you ask me.

On line 17 we ask the Scanner object to let the human type in something which it will try to format as an integer, and then that value will be stored into the integer variable named *age*. We display that value on the screen on line 19.

Line 21 reads in a double value and stores it into *weight*, and line 24 reads in another double value and stores it into *income*.

This is a really powerful thing. With some variables and with the help of the Scanner object, we can now let the human type in information, and we can remember it in a variable to use later in the program!

Before I wrap up, notice for example that the variable *income* is declared all the way up on line 9 (we choose its name and type), but it is undefined (it doesn't have a value) until line 24. On line 24 *income* is finally initialized (given its first value of the program). If you had attempted to print the value of *income* on the screen prior to line 24, the program would not have compiled.

Anyway, play with typing in different answers to the questions and see if you can get the program to blow up after each question.

Exercise 9: Calculations with User Input

Now that we know how to get input from the user and store it into variables and since we know how to do some basic math, we can now write our first *useful* program!

```
1 import java.util.Scanner;
 2
 3 public class BMICalculator
 4 {
 5     public static void main( String[] args )
 6     {
 7         Scanner keyboard = new Scanner(System.in);
 8         double m, kg, bmi;
 9
 10         System.out.print( "Your height in m: " );
11         m = keyboard.nextDouble();
12
 13         System.out.print( "Your weight in kg: " );
14         kg = keyboard.nextDouble();
15
 16         bmi = kg / (m*m);
17
 18         System.out.println( "Your BMI is " + bmi );
19     }
20 }
```

What You Should See

```
Your height in m: 1.75
Your weight in kg: 73
Your BMI is 23.836734693877553
```

This exercise is (hopefully) pretty straightforward. We have three variables (all doubles): *m* (meters), *kg* (kilograms) and *bmi* (body mass index). We read in values for *m* and *kg*, but *bmi*'s value comes not from the human but as the result of a calculation. On line 16 we compute the mass divided by the square of the height and store the result into *bmi*. And then we print it out.

The body mass index (BMI) is commonly used by health and nutrition professionals to estimate human body fat in populations. So this result would be informative for a health professional. For now that's all we can do with it.

Eventually we will learn how to display a different message on the screen *depending* on what the BMI equals, but for now this will have to do.

A pretty easy assignment for today, but I have some challenges for you in the Study Drills that should make things a bit tougher.

Study Drills

1. Add some variables and change the program so that the human can input their weight and height using pounds and inches, and then convert those values to kilograms and meters to figure the BMI.

Your height in inches: 69
Your weight in pounds: 160
Your BMI is 23.625289

1. ---

1. 2. Make it so the human can input their height in feet and inches separately.

Your height (feet only): 5
Your height (inches): 9
Your weight in pounds: 160
Your BMI is 23.625289

Exercise 10: Variables Only Hold Values

Okay, now that we can get input from the human and do calculations with it, I want to call attention to something that many of my students get confused about. The following code should compile, but it probably will not work the way you expect.

I have intentionally made a *logical* error in the code. It is not a problem with the syntax (the part of the code the compiler cares about), and it is not a runtime error like you get when the human types a double when the Scanner object is expecting an integer. This logical error is a flaw with how I have designed the flow of instructions, so that the output is not what I was trying to accomplish.

```
1 import java.util.Scanner;
 2
 3 public class Sequencing
 4 {
 5     public static void main( String[] args )
 6     {
 7         // BROKEN
 8
 9         Scanner keyboard = new Scanner(System.in);
10         double price = 0, salesTax, total;
11
12         salesTax = price * 0.0825;
13         total = price + salesTax;
14
15         System.out.print( "How much is the purchase price? " );
16         price = keyboard.nextDouble();
17
18         System.out.println( "Item price:\t" + price );
19         System.out.println( "Sales tax:\t" + salesTax );
20         System.out.println( "Total cost:\t" + total );
21     }
22 }
```

What You Should See

```
How much is the purchase price? 7.99
Item price:     7.99
Sales tax:      0.0
Total cost:     0.0
```

Are you surprised by the output? Did you expect the sales tax on $7.99 to show something like $0.66 instead of a big fat zero? And the total cost should have been something like $8.65, right? What happened?

What happened is that in Java (and most programming languages), *variables can not hold formulas*. Variables can only hold values.

Look at line 12. My students sometimes think that line stores the *formula* price * 0.0825 into the variable *salesTax* and then later the human stores the value 7.99 into the variable *price*. They think that on line 19 when we print out

salesTax that the computer then "runs" the formula somehow.

This is not what happens. In fact, this program shouldn't have even compiled. The variable *price* doesn't even have a proper value on line 12. The only reason it does have a value is because I did something sneaky on line 10.

Normally we have been declaring variables up at the top of our programs and then initializing them later. But on line 10 I declared *price* **and** initialized it with a value of 0. When you declare and initialize a variable at the same time, that is called "defining" the variable. *salesTax* and *total* are not defined on line 10, just declared.

So then on line 16 the value the human types in doesn't initialize *price*; *price* already has an initial value (0). But the value the human types in (7.99 or whatever) *does* get *stored* into the variable *price* here. The 0 is replaced with 7.99.

From line 10 until 15 the variable *price* contains the value 0. When line 16 begins executing and while we are waiting for the human to type something, *price* still contains 0. But by the time line 16 has completed, whatever the human typed has been stored into *price*, replacing the zero. Then from line 17 until the end of the program, the variable *price* contains the value 7.99 (or whatever).

So with this in mind, we can figure out what really happens on line 12. Line 12 does *not* store a formula into *salesTax* but it does store a value. What value? It takes the value of the variable *price* **at this point in the code** (which is 0), multiplies it by 0.0825 (which is still zero), and then stores that zero into *salesTax*.

As line 12 is beginning, the value of *salesTax* is undefined. (*salesTax* is declared but not defined.) By the end of line 12, *salesTax* holds the value 0. There is no line of code that changes *salesTax* (there is no line of code that begins with salesTax =), so that value never changes and *salesTax* is still zero when it is displayed on line 19.

Line 13 is similar. It takes the value of *price* **right then** (zero) and adds it to the value of *salesTax* **right then** (also zero) and stores the sum (zero) into the variable *total*. And *total*'s value is never changed, and *total* does not somehow "remember" that its value came from a formula involving some variables.

So there you have it. Variables hold values, not formulas. Computer programs are not a set of rules, they are a *sequence* of instructions that the computer executes *in order*, and things later in your code depend on what happened before.

Study Drills

1. Remove the " = 0" on line 10, so that *price* no longer gets defined on line 10, only declared. What happens when you try to compile the code? Does the error message make sense? (Now put the " = 0" back so that the program compiles again.)
44. Move the two lines of code that give values to *salesTax* and *total* so they occur after *price* has been given a proper value. Confirm that the program

now works as expected.

45. Now that these lines occur *after* the variable *price* has been properly given a real value, try removing the " = 0" on line 10 again. Does the program still give an error? Are you surprised?

Exercise 11: Variable Modification Shortcuts

The value of a variable can change over time as your program runs. (It won't change unless you write code to change it, but it *can* change is what I'm saying.)

In fact, this is pretty common. Something we do pretty often is take a variable and add something to it. For example, let's say the variable *x* contains the value 10. We want to add 2 to it so that *x* now contains 12.

We can do this:

```
int x = 10, temp_x;
temp_x = x + 2;
x = temp_x;
```

This will work, but it is annoying. If we want, we can take advantage of the fact that a variable can have one value at the beginning of a line of code and have a different value stored in it by the end. So we can write something like this:

```
int x = 10;
x = 2 + x;
```

This also works. That second line says "take the current value of *x* (10), add 2 to it, and store the sum (12) into the variable *x*. So when the second line of code begins executing *x* is 10, and when it is done executing, *x* is 12. The order of adding doesn't matter, so we can even do something like this:

```
int x = 10;
x = x + 2;
```

…which is identical to the previous example. Okay, now to the code!

```
1 public class VariableChangeShortcuts
 2 {
 3    public static void main( String[] args )
 4    {
 5       int i, j, k;
 6
 7       i = 5;
 8       j = 5;
 9       k = 5;
10
 11       System.out.println( "i: " + i + "\tj: " + j + "\tk: " + k );
12       i =  i + 3;
13       j =  j - 3;
14       k =  k * 3;
15       System.out.println( "i: " + i + "\tj: " + j + "\tk: " + k );
16
 17       i = 5;
18       j = 5;
19       k = 5;
20
 21       System.out.println( "\ni: " + i + "\tj: " + j + "\tk: " + k );
```

```
22        i += 3;
23        j -= 3;
24        k *= 3;
25        System.out.println( "i: " + i + "\tj: " + j + "\tk: " + k );
26
 27        i = j = k = 5;
28
 29        System.out.println( "\ni: " + i + "\tj: " + j + "\tk: " + k );
30        i += 1;
31        j -= 2;
32        k *= 3;
33        System.out.println( "i: " + i + "\tj: " + j + "\tk: " + k );
34
 35        i = j = k = 5;
36
 37        System.out.println( "\ni: " + i + "\tj: " + j + "\tk: " + k );
38        i =+ 1;
39        j =- 2;
40        System.out.println( "i: " + i + "\tj: " + j + "\tk: " + k );
41
 42        i = j = k = 5;
43
 44        System.out.println( "\ni: " + i + "\tj: " + j + "\tk: " + k );
45        i++;
46        j--;
47        System.out.println( "i: " + i + "\tj: " + j + "\tk: " + k );
48
 49    }
50 }
```

What You Should See

```
i: 5    j: 5    k: 5
i: 8    j: 2    k: 15

i: 5    j: 5    k: 5
i: 8    j: 2    k: 15

i: 5    j: 5    k: 5
i: 6    j: 3    k: 15

i: 5    j: 5    k: 5
i: 1    j: -2   k: 5

i: 5    j: 5    k: 5
i: 6    j: 4    k: 5
```

Hopefully lines 1-21 are nice and boring. We create three variables, give them values, display them, change their values and print them again. Then starting on line 17 we give the variables the same values they started with and print them.

On line 22 we see something new: a shortcut called a “compound assignment operator.” The i += 3 means the same as i = i + 3: “take the current value of *i*, add 3 to it, and store the result as the new value of *i*. When we say it out loud, we would say “i plus equals 3.”

On line 23 we see -= ("minus equals"), which subtracts 3 from *k*, and line 24 demonstrates *=, which multiplies. There is also /=, which divides whatever variable is on the left-hand side by whatever value the right-hand side ends up equaling. ("Modulus equals" (%=) also exists, which sets the variable on the left-hand side equal to whatever the remainder is when its previous value is divided by whatever is on the right. Whew.)

Then on line 27 I do something else weird. Instead of taking three lines of code to set *i*, *j* and *k* all to 5, I do it in one line. (Some people don't approve of this trick, but I think it's fine in cases like this.) Line 27 means "Put the value 5 into the variable *k*. Then take a copy of whatever value is now in *k* (5) and store it into *j*. Then take a copy of whatever is now in *j* and store it into *i*." So when this line of code is done, all three variables have been changed to equal 5.

Lines 30 through 32 are basically the same as lines 22 through 24 except that we are no longer using 3 as the number to add, subtract or multiply.

Line 38 might look like a typo, and if you wrote this in your own code it probably would be. Notice that instead of += I wrote =+. This will compile, but it is not interpreted the way you'd expect. The compiler sees i = +1;, that is, "Set *i* equal to positive 1." And line 39 is similar "Set *j* equal to negative 2." So watch for that.

On line 45 we see one more shortcut: the "post-increment operator." i++ just means "add 1 to whatever is in *i*." It's the same as writing i = i + 1 or i += 1. Adding 1 to a variable is *super* common. (You'll see.) That's why there's a special shortcut for it.

On line 46 we see the "post-decrement operator": j--. It subtracts 1 from the value in *j*.

Today's lesson is unusual because these shortcuts are optional. You could write code your whole life and never use them. But most programmers are lazy and don't want to type any more than they have to, so if you ever read other people's code you will see these pretty often.

Exercise 12: Boolean Expressions

So far we have only seen three types of variables:

int

integers, hold numbers (positive or negative) with no fractional parts

double

"double-precision floating-point" numbers (positive or negative) that could have a fractional part

String

a string a characters, hold words, phrases, symbols, sentences, whatever

But in the words of Yoda: "There is another." A "Boolean" variable (named after the mathematician George Boole) cannot hold numbers or words. It can only store one of two values: true or false. That's it. We can use them to perform logic. To the code!

```
1 import java.util.Scanner;
 2
 3 public class BooleanExpressions
 4 {
 5     public static void main( String[] args )
 6     {
 7         Scanner keyboard = new Scanner(System.in);
 8
 9         boolean a, b, c, d, e, f;
10         double x, y;
11
12         System.out.print( "Give me two numbers. First: " );
13         x = keyboard.nextDouble();
14         System.out.print( "Second: " );
15         y = keyboard.nextDouble();
16
17         a = (x <  y);
18         b = (x <= y);
19         c = (x == y);
20         d = (x != y);
21         e = (x >  y);
22         f = (x >= y);
23
24         System.out.println( x + " is LESS THAN " + y + ": " + a );
25         System.out.println( x + " is LESS THAN or EQUAL TO " + y + ": " + b );
26         System.out.println( x + " is EQUAL TO " + y + ": " + c );
27         System.out.println( x + " is NOT EQUAL TO " + y + ": " + d );
28         System.out.println( x + " is GREATER THAN " + y + ": " + e );
29         System.out.println( x + " is GREATER THAN or EQUAL TO " + y + ": " + f );
30         System.out.println();
31
32         System.out.println( !(x < y) + " " + (x >= y) );
33         System.out.println( !(x <= y) + " " + (x > y) );
34         System.out.println( !(x == y) + " " + (x != y) );
35         System.out.println( !(x != y) + " " + (x == y) );
36         System.out.println( !(x > y) + " " + (x <= y) );
37         System.out.println( !(x >= y) + " " + (x < y) );
38
```

```
 39     }
40 }
```

What You Should See

```
Give me two numbers. First: 3
Second: 4
3.0 is LESS THAN 4.0: true
3.0 is LESS THAN or EQUAL TO 4.0: true
3.0 is EQUAL TO 4.0: false
3.0 is NOT EQUAL TO 4.0: true
3.0 is GREATER THAN 4.0: false
3.0 is GREATER THAN or EQUAL TO 4.0: false

false false
false false
true true
false false
true true
true true
```

On line 17 the Boolean variable *a* is set equal to something strange: the result of a comparison. The current value in the variable *x* is compared to the value of the variable *y*. If *x* is less than *y*, then the comparison is true and the Boolean value true is stored into *a*. If *x* is not less than *y*, then the comparison is false and the boolean value false is stored into *a*. (I think that is easier to understand than it is to write.)

Line 18 is similar, except that the comparison is "less than or equal to", and the Boolean result is stored into *b*.

Line 19 is "equal to": *c* will be set to the value true if *x* holds the same value as *y*. The comparison in line 20 is "not equal to". Lines 21 and 22 are "greater than" and "greater than or equal to", respectively.

On lines 24 through 29, we display the values of all those Boolean variables on the screen.

Line 32 through line 37 introduce the "not" operator, which is an exclamation point (!). It takes the logical opposite. So on line 32 we display the logical negation of "x is less than y?", and we also print out the truth value of "x is greater than or equal to y?", which are equivalent. (The opposite of "less than" is "greater than or equal to".) Lines 33 through 37 show the opposites of the remaining relational operators.

Exercise 13: Comparing Strings

In this exercise we will see something that causes trouble for beginners trying to learn Java: the regular relational operators do not work with Strings, only numbers.

```
boolean a, b;
a = ("cat" < "dog");
b = ("horse" == "horse" );
```

The second line doesn't even compile! You can't use < to see if a word comes before another word in Java. And in the third line, *b* does get set to the value true here, but not if you read the value into a variable like so:

```
String animal;
animal = keyboard.next(); // the user types in "horse"
b = ( animal == "horse" );
```

b will always get set to the value false, no matter if the human types "horse" or not!

I don't want to try to explain why this is. The creators of Java do have a good reason for this apparently weird behavior, but it's not friendly for beginners and explaining it would probably only confuse you more at this point in your learning.

Do you remember when I warned you that Java is not a language for beginners?

So there *is* a way to compare Strings for equality, so let's look at it.

```
1 import java.util.Scanner;
 2
 3 public class WeaselOrNot
 4 {
 5     public static void main( String[] args )
 6     {
 7         Scanner keyboard = new Scanner(System.in);
 8
 9         String word;
10          boolean yep, nope;
11
 12          System.out.println( "Type the word \"weasel\", please." );
13         word = keyboard.next();
14
 15          yep  =   word.equals("weasel");
16         nope = ! word.equals("weasel");
17
 18          System.out.println( "You typed what was requested: " + yep );
19         System.out.println( "You ignored polite instructions: " + nope );
20     }
21 }
```

What You Should See

```
```

```
Type the word "weasel", please.
no
You typed what was requested: false
You ignored polite instructions: true
```

So Strings have a built-in method named .equals() (“dot equals”) that compares itself to another String, simplifying to the value true if they are equal and to the value false if they are not. And you must use the not operator (!) together with the .equals() method to find out if two Strings are different.

Study Drills

1. Try changing around the comparison on line 15 so that "weasel" is in front of the dot and the variable *word* is inside the parentheses. Make sure that "weasel" is still surrounded by quotes and that *word* is not. Does it work?

Exercise 14: Compound Boolean Expressions

Sometimes we want to use logic more complicated than just "less than" or "equal to". Imagine a grandmother who will only approve you dating her grandchild if you are older than 25 *and* younger than 40 *and* either rich or really good looking. If that grandmother was a programmer and could convince applicants to answer honestly, her program might look a bit like this:

```
1 import java.util.Scanner;
 2
 3 public class ShallowGrandmother
 4 {
 5     public static void main( String[] args )
 6     {
 7         Scanner keyboard = new Scanner(System.in);
 8
 9         int age;
10         double income, attractiveness;
11         boolean allowed;
12
 13         System.out.print( "Enter your age: " );
14         age = keyboard.nextInt();
15
 16         System.out.print( "Enter your yearly income: " );
17         income = keyboard.nextDouble();
18
 19         System.out.print( "How attractive are you, on a scale from 0.0 to 10.0? " );
20         attractiveness = keyboard.nextDouble();
21
 22         allowed = ( age > 25 && age < 40 && ( income > 50000 || attractiveness >= 8.5 ));
23
 24         System.out.println( "You are allowed to date my grandchild: " + allowed );
25     }
26 }
```

What You Should See

```
Enter your age: 39
Enter your yearly income: 49000
How attractive are you, on a scale from 0.0 to 10.0? 7.5
You are allowed to date my grandchild: false
```

So we can see that for complicated Boolean expressions you can use parentheses to group things, and you use the symbols && to mean "AND" and the symbols || to mean "OR".

I know what you are thinking: using & (an "ampersand" or "and sign") to mean "AND" makes a little sense, but why two of them? And whose idea was it to use || ("pipe pipe") to mean "OR"?!?

> Well, Ken Thompson's idea, probably. Java syntax is modeled after C++'s syntax, which was basically copied from C's syntax, and that was modified from B's syntax, which was invented by Dennis Ritchie and Ken Thompson.
>
> The programming language *B* used & to mean "AND" and | for "OR", but they were "bitwise": they only worked on two integers

and they would walk one bit at a time down the integers doing a bitwise AND or OR on each pair of bits, putting a 1 or 0 in the output for each comparison. (| was probably used because it was mathematical-looking and was a key on the keyboards of the PDP-7 computer that *B* was originally developed for.)

When Ken and Dennis started developing the programming language *C* to replace *B*, they decided there was a need for a *logical* "AND" and "OR", and the one-symbol-long things were already taken, so they used two ampersands to represent logical "AND" and two vertical bars or "pipes" to represent logical "OR". Whew.

Fortunately for you, you don't need to know any of that. You just need to remember what to type and get it right.

This next little bit is going to be a little bit weird, because I'm going to show you the "truth tables" for AND and OR, and you'll have to think of "AND" as an *operation* that is being performed on two values instead of a conjunction.

Here is the truth table for AND:

Inputs			***Output(s)***
A	***B***	***A && B***	
true	true	true	
true	false	false	
false	true	false	
false	false	false	

You read the tables this way: let's pretend that our shallow grandmother has decided that she will only go on a cruise if it is cheap AND the alcohol is included in the price. So we will pretend that statement A is "the cruise is cheap" and statement B is "the alcohol is included". Each row in the table is a possible cruise line.

Row 1 is a cruise where both statements are true. Will grandmother be excited about cruise #1? Yes! "Cruise is cheap" is true and "alcohol is included" is true, so "grandmother will go" (A && B) is also true.

Cruise #2 is cheap, but the alcohol is *not* included (statement B is false). So grandmother isn't interested: (A && B) is false when A is true and B is false.

Clear? Now here is the truth table for OR:

Inputs			***Output(s)***
A	***B***	***A || B***	
true	true	true	
true	false	true	

false	true	true
false	false	false

Let's say that grandmother will buy a certain used car if it is really cool-looking OR if it gets great gas mileage. Statement A is "car is cool looking", B is "good miles per gallon" and the result, A OR B, determines if it is a car grandmother would want.

Car #1 is awesome-looking and it also goes a long way on a tank of gas. Is grandmother interested? Heck, yes! We can say that the value true ORed with the value true gives a result of true.

In fact, the only car grandmother won't like is when both are false. An expression involving OR is only false if BOTH of its components are false.

Study Drills

1. Did you know that Java has bitwise operators, too? Investigate how numbers are represented in binary and see if you can figure out why the following code sets *x* to the value 7 and sets *y* to the value 1.

```
int x = 3 | 5;
int y = 3 & 5;
```

Exercise 15: Making Decisions with If Statements

Hey! I really like this exercise. You suffered through some pretty boring ones there, so it's time to learn something that is useful and not super difficult.

We are going to learn how to write code that has decisions in it, so that the output isn't always the same. The code that gets executed changes depending on what the human enters.

```
1 import java.util.Scanner;
 2
 3 public class AgeMessages
 4 {
 5     public static void main( String[] args )
 6     {
 7         Scanner keyboard = new Scanner(System.in);
 8
 9         int age;
10
 11         System.out.print( "How old are you? " );
12         age = keyboard.nextInt();
13
 14         if ( age < 13 )
15         {
16             System.out.println( "You are too young to create a Facebook account." );
17         }
18         if ( age < 16 )
19         {
20             System.out.println( "You are too young to get a driver's license." );
21         }
22         if ( age < 18 )
23         {
24             System.out.println( "You are too young to get a tattoo." );
25         }
26         if ( age < 21 )
27         {
28             System.out.println( "You are too young to drink alcohol." );
29         }
30         if ( age < 35 )
31         {
32             System.out.println( "You are too young to run for President of the United States." );
33             System.out.println( "How sad!" );
34         }
35     }
36 }
```

What You Should See

```
How old are you? 17
You are too young to get a tattoo.
You are too young to drink alcohol.
You are too young to run for President of the United States.
How sad!
```

Okay, this is called an "if statement". An if statement starts with the keyword if,

followed by a "condition" in parentheses. The condition must be a Boolean expression that evaluates to either true or false. Underneath that starts a block of code surrounded by curly braces, and the stuff inside the curly braces is indented one more level. That block of code is called the "body" of the if statement.

When the condition of the if statement is true, all the code in the body of the if statement is executed. When the condition of the if statement is false, all the code in the body is skipped. You can have as many lines of code as you want inside the body of an if statement; they will all be executed or skipped as a group.

Notice that when I ran the code, I put in 17 for my age. Because 17 is not less than 13, the condition on line 14 is false, and so the code in the body of the first if statement (lines 15 through 17) was skipped.

The second if statement was also false because 17 is not less than 16, so the code in its body (lines 19 through 21) was skipped, too.

The condition of the third if statement was true: 17 *is* less than 18, so the body of the third if statement was not skipped; it was executed and the phrase "You are too young to get a tattoo" *was* printed on the screen. The remaining if statements in the exercise are all true.

The final if statement contains two lines of code in its body, just to show you what it would look like.

Study Drills

1. If you type in an age greater than 35 what gets printed? Why?
46. Add one more if statement comparing their age to 65. If their age is greater than or equal to 65, say "You are old enough to retire!".
47. For each if statement, add another if statement that says the opposite. For example, if their age is greater than or equal to 13, say "You are old enough to create a Facebook account." When you are done, your program should show six messages every time no matter what age you enter.

Exercise 16: More If Statements

There is almost nothing new in this exercise. It is just more practice with if statements, because they're pretty important. It will also help you to remember the relational operators.

```
1 import java.util.Scanner;
 2
 3 public class ComparingNumbers
 4 {
 5     public static void main( String[] args )
 6     {
 7         Scanner keyboard = new Scanner(System.in);
 8         double first, second;
 9
 10         System.out.print( "Give me two numbers. First: " );
11         first = keyboard.nextDouble();
12         System.out.print( "Second: " );
13         second = keyboard.nextDouble();
14
 15         if ( first < second )
16         {
17             System.out.println( first + " is LESS THAN " + second );
18         }
19         if ( first <= second )
20         {
21             System.out.println( first + " is LESS THAN or EQUAL TO " + second );
22         }
23         if ( first == second )
24         {
25             System.out.println( first + " is EQUAL TO " + second );
26         }
27         if ( first >= second )
28         {
29             System.out.println( first + " is GREATER THAN or EQUAL TO "+second);
30         }
31         if ( first > second )
32         {
33             System.out.println( first + " is GREATER THAN " + second );
34         }
35
 36         if ( first != second )
37             System.out.println( first + " is NOT EQUAL TO " + second );
38
 39     }
40 }
```

What You Should See

```
Give me two numbers. First: 3
Second: 4
3.0 is LESS THAN 4.0
3.0 is LESS THAN or EQUAL TO 4.0
3.0 is NOT EQUAL TO 4.0
```

On line 37 you will see that I did something questionable: the body of the last if statement does not have any curly braces around it. Is this okay?

Actually, it is. When the body of an if statement does not have curly braces, then only the first line of code after the condition is included in the body. So, since all the if statements in this whole exercise have only one line of code in their bodies, all the if statement curly braces in this exercise are optional. You could remove and the program would work the same. It is never wrong to include them, though, and some programmers always put curly braces no matter what.

Study Drills

1. Add another line of code after line 37 that says System.out.println("Hey.");. Indent it so that it lines up with the println() statement above it, like so:

```
if ( first != second )
   System.out.println( first + " is NOT EQUAL TO " + second );
   System.out.println( "Hey." );
```

Run the program, and see what happens. Is the "Hey" part of the if statement body? That is, when the if statement is skipped, is the "Hey" skipped, too, or does it run no matter what? What do you think?

1. Add curly braces around the body of the final if statement so that the "Hey" line is part of the body. Then remove all the *other* if statement body curly braces so that only the last if statement in the program has them. Confirm that everything works as expected.

Exercise 17: Otherwise (If Statements with Else)

So, if statements are pretty great. Almost every programming language has them, and you use them ALL the time. In fact, if statements alone are functional enough that you could do a lot just using if statements.

But sometimes, having something else could make things a little more convenient. Like this example: quick! What is the logical opposite of the following expression?

```
if ( onGuestList || age >= 21 || ( gender.equals("F") && attractiveness >= 8 ) )
```

Eh? Got it yet? Well, if you said

```
if ( ! ( onGuestList || age >= 21 || ( gender.equals("F") && attractiveness >= 8 ) ) )
```

…then you're right and you're my kind of person. Clever and knows when to let the machine do the work for you. If you said

```
if ( ! onGuestList && age < 21 && ( ! gender.equals("F") || attractiveness < 8 ) )
```

…then you're right and… nice job. That's actually pretty tough to do correctly. But what about the logical opposite of this:

```
if ( expensiveDatabaseThatTakes45SecondsToLookup( userName, password ) == true )
```

Do we really want to write

```
if ( expensiveDatabaseThatTakes45SecondsToLookup( userName, password ) == false )
```

…because now we're having to wait 90 seconds to do two if statements instead of 45 seconds. So fortunately, programming languages give us something else. (Yeah, sorry. Couldn't resist.)

```
1 import java.util.Scanner;
 2
 3 public class ClubBouncer
 4 {
 5     public static void main( String[] args )
 6     {
 7         Scanner keyboard = new Scanner(System.in);
 8
 9         int age = 22;
10         boolean onGuestList = false;
11         double attractiveness = 7.5;
12         String gender = "F";
13
14         if ( onGuestList || age >= 21 || ( gender.equals("F") && attractiveness >= 8 ) )
15         {
16             System.out.println("You are allowed to enter the club.");
17         }
18         else
```

```
19      {
20         System.out.println("You are not allowed to enter the club.");
21      }
22   }
23 }
```

What You Should See

```
You are allowed to enter the club.
```

So what the keyword else means is this: look at the preceding if statement. Was the condition of that if statement true? If so, skip. If that previous if statement did *not* run, however, then the body of the else statement will be executed. “If blah blah blah is true, then run this block of code. Otherwise (else), run this different block of code instead.”

Else is *super* convenient because then we don’t *have* to figure out the logical opposite of some complex Boolean expression. We can just say else and let the computer deal with it.

An else is *only legal* immediately after an if statement ends. (Technically it is only allowed after the closing of the block of code that is the body of an if statement.)

Study Drills

1. Between line 17 and 18, add a println() statement to print something on the screen (it doesn’t matter what, but I put "C-C-C-COMBO BREAKER" because I’m weird.) Try to compile the program. Does it compile? Why not?

Exercise 18: If Statements with Strings

A few exercises back you learned how comparing Strings is not as easy as comparing numbers. So let's review with an example you can actually test out.

```
1 import java.util.Scanner;
 2
 3 public class SecretWord
 4 {
 5     public static void main( String[] args )
 6     {
 7         Scanner keyboard = new Scanner(System.in);
 8
 9         String secret = "please", guess;
10
 11         System.out.print( "What's the secret word? " );
12         guess = keyboard.next();
13
 14         if ( guess == secret )
15         {
16             System.out.println( "Impossible. (This will never be printed.)" );
17         }
18
 19         if ( guess.equals(secret) )
20         {
21             System.out.println( "That's correct!" );
22         }
23         else
24         {
25             System.out.println( "Nope, the secret word is not \"" + guess + "\"." );
26         }
27
 28     }
29 }
```

What You Should See

```
What's the secret word? abracadabra
Nope, the secret word is not "abracadabra".
```

Notice that as usual I'm sneaking something in. On line 9 instead of just declaring *secret* I also gave it a value. That is, I "defined" it (declared and initialized all at once).

Anyway, the if statement on line 14 will **never** be true. Never ever. No matter what you type in, it will never be the case that *guess == secret*.

> (I can't really explain why without going into way too much detail, but it has to do with the fact that == only compares the shallow values of the variables, and the shallow values of two Strings are only equal when they refer to the same memory location.)

What does work is using the .equals() method (which compares the deep values of the variables instead of their shallow values). This will be true if they type

the correct secret word.

Exercise 19: Mutual Exclusion with Chains of If and Else

In the previous exercise, we saw how using else can make it easier to include a chunk of alternative code that you want to run when an if statement did *not* happen.

But what if the alternative code is… another if statement?

```
1 import java.util.Scanner;
 2
 3 public class BMICategories
 4 {
 5    public static void main( String[] args )
 6    {
 7       Scanner keyboard = new Scanner(System.in);
 8
 9       double bmi;
10
 11       System.out.print( "Enter your BMI: " );
12       bmi = keyboard.nextDouble();
13
 14       System.out.print( "BMI category: " );
15       if ( bmi < 15.0 )
16       {
17          System.out.println( "very severely underweight" );
18       }
19       else if ( bmi <= 16.0 )
20       {
21          System.out.println( "severely underweight" );
22       }
23       else if ( bmi < 18.5 )
24       {
25          System.out.println( "underweight" );
26       }
27       else if ( bmi < 25.0 )
28       {
29          System.out.println( "normal weight" );
30       }
31       else if ( bmi < 30.0 )
32       {
33          System.out.println( "overweight" );
34       }
35       else if ( bmi < 35.0 )
36       {
37          System.out.println( "moderately obese" );
38       }
39       else if ( bmi < 40.0 )
40       {
41          System.out.println( "severely obese" );
42       }
43       else
44       {
45          System.out.println( "very severely/\"morbidly\" obese" );
46       }
47    }
48 }
```

What You Should See

```
Enter your BMI: 22.5
BMI category: normal weight
```

(*Note*: Although BMI is a very good estimate of human body fat, the formula doesn't work well for athletes with a lot of muscle, or people who are extremely short or very tall. If you are concerned about your BMI, check with your doctor.)

Notice that even though several of the if statements might have all been true, only the *first* true if statement printed its message on the screen. No other messages were printed: only one. That's the power of using else with if.

On line 15 there is an if statement that checks if your BMI is less than 15.0, and if so, displays the proper category for that body mass index.

Line 19 begins with an else. That else pays attention to the preceding if statement – the one on line 15 – to determine if it should run its body of code or skip it automatically. Assuming you typed in a BMI of 22.5, then the preceding if statement was not true and did not run. Because that if statement failed, the else will automatically execute its body of code.

However, that body of code starts *right after* the word else with a new if statement! This means that the statement if (bmi <= 16.0) will *only* be considered when the previous if statement was false.

Whenever my students are confused about this, I have an analogy I give them. (It's a little crude, but it seems to help.)

Imagine that you are single (romantically, I mean) and that you and some of your friends are out in a bar or the mall or whatever. Across the way, you see a really attractive single person and under your breath you tell the others: "Okay, dibs. I get the first shot."

Your group travels over to this person but no one else starts flirting until they see how you fare. If you seem to be making progress, your friends will back off and let you chat away. If however, you get rejected then one of your other companions feels cleared to try and make a play.

This is basically exactly what happens with else if. An else if statement (an if statement with an else in front of the if) contains a condition that might be true or might be false. But the else means that the if statement will *only* check to see if it is true or false assuming the preceding if statement (and only the immediately preceding one) was false.

The else on line 23 makes *that* if statement defer to the if statement that starts on line 19: if it's true, the line-23 if statement will skip *even if* it would have been true on its own. The else on line 27 makes its if statement defer to the previous if statement, and so on. And the else at the very end on line 43 is like the smallest dog in a pack: it only gets a shot if *all* the previous if statements in the chain were false.

We'll talk a little bit more about this in the next exercise, so that's enough for now.

Study Drills

1. Remove the else from in front of the if statement on line 27. Run the program, and enter 15.5 for the BMI. Do you see how that makes the if statement on line 27 "break rank" and no longer care about the if statements before it?
48. Instead of making the human enter their BMI directly, allow them to type in their height and weight and compute the BMI for them.

Exercise 20: More Chains of Ifs and Else

Okay, let's look a little more at making chains of conditions using else and if.

A confession: although I did attend UT Austin I don't think this is their real admissions criteria. Don't rely on the output of this program when deciding whether or not to apply to a back-up school.

```
1 import java.util.Scanner;
 2 import static java.lang.System.*;
 3
 4 public class CollegeAdmission
 5 {
 6     public static void main( String[] args )
 7     {
 8         Scanner keyboard = new Scanner(System.in);
 9         int math;
10
 11         out.println( "Welcome to the UT Austin College Admissions Interface!" );
12         out.print( "Please enter your SAT math score (200-800): " );
13         math = keyboard.nextInt();
14
 15         out.print( "Admittance status: " );
16
 17         if ( math >= 790 )
18             out.print( "CERTAIN " );
19         else if ( math >= 710 )
20             out.print( "SAFE " );
21         else if ( math >= 580 )
22             out.print( "PROBABLE " );
23         else if ( math >= 500 )
24             out.print( "UNCERTAIN " );
25         else if ( math >= 390 )
26             out.print( "UNLIKELY " );
27         else // below 390
28             out.print( "DENIED " );
29
 30         out.println();
31     }
32 }
```

What You Should See

```
Welcome to the UT Austin College Admissions Interface!
Please enter your SAT math score (200-800): 730
Admittance status: SAFE
```

Now, before I go into the new thing for this exercise, I should explain a shortcut I took in this program. Did you notice there was a second import statement at the top? If not, then your code didn't compile or you thought I made a mistake by putting out.println instead of System.out.println everywhere.

Well, I don't want to talk much about object-oriented code in this book, since that's severely *not* beginner friendly, but think of it like this. There is a built-in object in Java called System. Inside that object there is another object named

out. That object named out contains a method called print() and one called println().

So when you write System.out.println you are asking the computer to run the method called println inside the object named out which is inside the object named System (which is itself part of the built-in import library java.lang.System).

Because of this, I can create a variable named out, and it won't be a problem:

```
String out;
```

Even though there's an object named out in existence somewhere, it's inside the System object so the names don't conflict.

If I am lazy and don't have any desire to have my own variable named out, then I can ask the computer to "import all static items which are inside the class java.lang.System into the current namespace":

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

So now I can type just out.println instead of System.out.println. Woo!

In this exercise I *also* omitted all the curly braces that delimit the blocks of code that are in the bodies of each if statement. Because I only want there to be a single statement inside the body of each if statement, this is okay. If I wanted there to be more than one line of code then I would have to put the curly braces back.

Anyway, in the previous exercise I wrote about how putting else in front of an if statement makes it defer to the previous if statement. When the previous one is true and executes the code in its body, the current one skips automatically (and all the rest of the else if statements in the chain will skip, too). This has the effect of making it so that only the *first* true value triggers the if statement and all the rest don't run. We sometimes say that the if statements are "mutually exclusive": exactly one of them will execute. Never fewer than one, never more than one.

Today's exercise is another example of that. But this time I want to point out that the mutual exclusion only works *properly* because I put the if statements in the correct order.

Since the first one that is true will go and the others will not, you need to make sure that the very first if statement in the chain is the one which is *hardest* to achieve. Then the next hardest, and so on, with the easiest at the end. In the study drills I will have you change the order of the if statements, and you will see how this can mess things up.

Also, technically, else statements should have curly braces, just like if statements do, and by putting else if with nothing in between we are taking advantage of the fact that the braces are optional. This makes the code a lot more compact. Here is what the previous code would look like if it were arranged the way the computer is interpreting it. Maybe it will help you to understand the "defer" behavior of the else in front of an if; maybe it will just confuse you.

Hopefully it will help.

```
1 import java.util.Scanner;
 2 import static java.lang.System.*;
 3
 4 public class CollegeAdmissionExpanded
 5 {
 6     public static void main( String[] args )
 7     {
 8        Scanner keyboard = new Scanner(System.in);
 9        int math;
10
 11        out.println( "Welcome to the UT Austin College Admissions Interface!" );
12        out.print( "Please enter your SAT math score (200-800): " );
13        math = keyboard.nextInt();
14
 15        out.print( "Admittance status: " );
16
 17        if ( math >= 790 )
18        {
19           out.print( "CERTAIN " );
20        }
21        else
22        {
23           if ( math >= 710 )
24           {
25              out.print( "SAFE " );
26           }
27           else
28           {
29              if ( math >= 580 )
30              {
31                 out.print( "PROBABLE " );
32              }
33              else
34              {
35                 if ( math >= 500 )
36                 {
37                    out.print( "UNCERTAIN " );
38                 }
39                 else
40                 {
41                    if ( math >= 390 )
42                    {
43                       out.print( "UNLIKELY " );
44                    }
45                    else // below 390
46                    {
47                       out.print( "DENIED " );
48                    }
49                 }
50              }
51           }
52        }
53        out.println();
54     }
55 }
```

Yeah. So you can see why we usually just put else if.

Study Drills

1. In the original code file (CollegeAdmission.java), remove all the elses except for the last one. Run it and notice how it prints *all* the messages. Then put the elses back.
49. Move lines 25 and 26 so they appear between lines 18 and 19. Compile it and run it and notice how the program almost always just says "UNLIKELY" because most SAT scores are more than 390 and the if statement is so high in the list that it steals the show most of the time.
50. If you want, type in the code for CollegeAdmissionExpanded.java and confirm that it works the same as the non-expanded version.

Exercise 21: Nested If Statements

You saw a glimpse of this in the previous exercise, but you can put just about anything you like inside the body of an if statement including other if statements. This is called "nesting", and an if statement which is inside another is called a "nested if".

Here's an example of using that to do something useful.

```
1 import java.util.Scanner;
 2
 3 public class GenderTitles
 4 {
 5     public static void main( String[] args )
 6     {
 7         Scanner keyboard = new Scanner(System.in);
 8
 9         String title;
10
 11         System.out.print( "First name: " );
12         String first = keyboard.next();
13         System.out.print( "Last name: " );
14         String last = keyboard.next();
15         System.out.print( "Gender (M/F): " );
16         String gender = keyboard.next();
17         System.out.print( "Age: " );
18         int age = keyboard.nextInt();
19
 20         if ( age < 20 )
21         {
22             title = first;
23         }
24         else
25         {
26             if ( gender.equals("F") )
27             {
28                 System.out.print( "Are you married, "+first+"? (Y/N): " );
29                 String married = keyboard.next();
30                 if ( married.equals("Y") )
31                 {
32                     title = "Mrs.";
33                 }
34                 else
35                 {
36                     title = "Ms.";
37                 }
38             }
39             else
40             {
41                 title = "Mr.";
42             }
43         }
44
 45         System.out.println( "\n" + title + " " + last );
46
 47     }
48 }
```

What You Should See

```
First name: Graham
Last name: Mitchell
Gender (M/F): M
Age: 39

Mr. Mitchell
```

You have probably figured out that I like to mix things up a little bit to keep you on your toes. Did you notice what I did differently this time?

Normally I declare all my variables at the top of the program and give them values (or "initialize" them) later. But you don't actually have to declare a variable until you're ready to use it. So this time, I declared all my variables (except *title*) on the same line I put a value into them for the first time.

Why didn't I declare *title* on line 22, then? Because then it wouldn't be in "scope" later. *Scope* refers to the places in your program where a variable is visible. The general rule is that a variable is in scope once it is declared and from that point forward in the code until the block ends that it was declared in. Then the variable goes out of scope and can't be used any more.

Let us look at an example: on line 29 I defined (declared and initialized) a String variable called *married*. It is declared inside the body of the female-gender if statement. This variable exists from line 29 down through line 38 at the close curly brace of that if statement's body block. The variable *married* is not in scope anywhere else in the program; referring to it on lines 1 through 28 or on lines 39 through 48 would give a compiler error.

This is why I had to declare *title* towards the beginning of the program. If I had declared it on line 22, then the variable would have gone out of scope one line later, when the close curly brace of the younger-than-20 block occurred. Because I need *title* to be visible all the way down through line 45, I need to make sure I declare it inside the block of code that ends on line 47.

I could have waited until line 19 to declare it, though.

Anyway, there's not much else interesting to say about this exercise except that it demonstrates nesting if statements and elses inside of others. I do have a little surprise in the study drills, though.

Study Drills

1. Change the else on line 39 to a suitable if statement instead, like:

```
if ( gender.equals("M") )
```

Notice that the program doesn’t compile anymore. Can you figure out why?

It is because the variable *title* is declared on line 9 but is not given a value right away. Then on line 45, the value of *title* is printed on the screen. The variable *must* have a value at this point, or we will be trying to display the value of a variable that is undefined: it *has* no value. The compiler wants to prevent this.

When line 39 was an else, the compiler could guarantee that no matter what path through the nested if statements was taken, *title* would **always** get a value. Once we changed it to a regular if statement, there is now a way the human could type something to get it to sneak through all the nested if statements without giving *title* a value. Can you think of one?

(They could type an age 20 or greater and a letter different than "M" or "F" when prompted for gender. Then neither gender if statement would be true.)

We can fix this by changing the if statement back to else (probably a good idea) **or** by initializing *title* right when we declare it (probably a good idea anyway):

```
String title = "error";
```

…or something like that. Now *title* has a value no matter what, and the compiler is happy.

Exercise 22: Making Decisions with a Big Switch

if statements aren't the only way to compare variables with value in Java. There is also something called a switch. I don't use them very often, but you should be familiar with them anyway in case you read someone else's code that uses one.

```
1 import java.util.Scanner;
 2
 3 public class ThirtyDays
 4 {
 5     public static void main( String[] args )
 6     {
 7         Scanner keyboard = new Scanner(System.in);
 8
 9         int month, days;
10          String monthName;
11
 12          System.out.print( "Which month? (1-12) " );
13          month = keyboard.nextInt();
14
 15          switch(month)
16          {
17             case  1: monthName = "January";
18                      break;
19             case  2: monthName = "February";
20                      break;
21             case  3: monthName = "March";
22                      break;
23             case  4: monthName = "April";
24                      break;
25             case  5: monthName = "May";
26                      break;
27             case  6: monthName = "June";
28                      break;
29             case  7: monthName = "July";
30                      break;
31             case  8: monthName = "August";
32                      break;
33             case  9: monthName = "September";
34                      break;
35             case 10: monthName = "October";
36                      break;
37             case 11: monthName = "November";
38                      break;
39             case 12: monthName = "December";
40                      break;
41             default: monthName = "error";
42          }
43
 44          /* Thirty days hath September
45            April, June and November
46            All the rest have thirty-one
47            Except the second month alone....
48          */
49
 50          switch(month)
51          {
52             case  9:
```

```
53          case  4:
54          case  6:
55          case 11: days = 30;
56                 break;
57          case  2: days = 28;
58                 break;
59          default: days = 31;
60        }
61
 62        System.out.println( days + " days hath " + monthName );
63
 64    }
65 }
```

What You Should See

```
Which month? (1-12) 4
30 days hath April
```

A switch statement starts with the keyword switch and then some parentheses. Inside the parentheses is a single variable (or an expression that simplifies to a single value). Then there's an open curly brace.

Inside the body of the switch statement are several case statements that begin with the keyword case and then a value that the variable up in the parentheses might equal. Then there's a colon (:). You don't see colons very often in Java.

After the case, the value and then colon is some code. It can be as many lines of code as you like except that you're not allowed to declare any variables inside the switch statement. Then after all the code is the keyword break. The break marks the end of the case.

When a switch statement runs, the computer figures out the current value of the variable inside the parentheses. Then it looks through the list of cases, one at a time, looking for a match. When it finds a match it moves from the left side where the cases are to the right side and starts running code until it is stopped by a break.

If none of the cases match and there is a default case (it's optional), then the code in the default case will be run instead.

The second example starts on line 50 and demonstrates that once the switch statement finds a case that matches, it really does run code on the right side until it hits a break statement. It will even fall through from one case to another.

We can take advantage of this fall-through behavior to do clever things sometimes, like the code to figure out the number of days in a month. Since September, April, June and November all have 30 days, we can just put all their cases in a row and let it fall through for any of those to run the same thing.

Anyway, I won't use switch statements again in this book because I just virtually never find a good use for them, but it does exist and at least I can say that you saw it.

Study Drills

1. Remove some of the break statements in the first switch and add some println() statements to confirm that it will set *monthName* to one value then another then another until it finally gets stopped by a break.

Exercise 23: More String Comparisons

Well, you have learned that you can't compare Strings with ==; you have to use the .equals() method. But I think you're finally ready to see how we can compare Strings for alphabetical ordering.

```
1 import java.util.Scanner;
 2
 3 public class DictionaryOrder
 4 {
 5    public static void main( String[] args )
 6    {
 7       Scanner keyboard = new Scanner(System.in);
 8
 9       String name;
10
 11       System.out.print( "Give me the name of a made-up programming language: " );
12       name = keyboard.nextLine();
13
 14       if ( name.compareTo("c++") < 0 )
15          System.out.println( name + " comes BEFORE c++" );
16       if ( name.compareTo("c++") == 0 )
17          System.out.println( "c++ isn't a made-up language!" );
18       if ( name.compareTo("c++") > 0 )
19          System.out.println( name + " comes AFTER  c++" );
20
 21       if ( name.compareTo("go") < 0 )
22          System.out.println( name + " comes BEFORE go" );
23       if ( name.compareTo("go") == 0 )
24          System.out.println( "go isn't a made-up language!" );
25       if ( name.compareTo("go") > 0 )
26          System.out.println( name + " comes AFTER  go" );
27
 28       if ( name.compareTo("java") < 0 )
29          System.out.println( name + " comes BEFORE java" );
30       if ( name.compareTo("java") == 0 )
31          System.out.println( "java isn't a made-up language!" );
32       if ( name.compareTo("java") > 0 )
33          System.out.println( name + " comes AFTER  java" );
34
 35       if ( name.compareTo("lisp") < 0 )
36          System.out.println( name + " comes BEFORE lisp" );
37       if ( name.compareTo("lisp") == 0 )
38          System.out.println( "lisp isn't a made-up language!" );
39       if ( name.compareTo("lisp") > 0 )
40          System.out.println( name + " comes AFTER  lisp" );
41
 42       if ( name.compareTo("python") < 0 )
43          System.out.println( name + " comes BEFORE python" );
44       if ( name.compareTo("python") == 0 )
45          System.out.println( "python isn't a made-up language!" );
46       if ( name.compareTo("python") > 0 )
47          System.out.println( name + " comes AFTER  python" );
48
 49       if ( name.compareTo("ruby") < 0 )
50          System.out.println( name + " comes BEFORE ruby" );
51       if ( name.compareTo("ruby") == 0 )
52          System.out.println( "ruby isn't a made-up language!" );
```

```
53        if ( name.compareTo("ruby") > 0 )
54           System.out.println( name + " comes AFTER  ruby" );
55
 56        if ( name.compareTo("visualbasic") < 0 )
57           System.out.println( name + " comes BEFORE visualbasic" );
58        if ( name.compareTo("visualbasic") == 0 )
59           System.out.println( "visualbasic isn't a made-up language!" );
60        if ( name.compareTo("visualbasic") > 0 )
61           System.out.println( name + " comes AFTER  visualbasic" );
62     }
63 }
```

What You Should See

```
Give me the name of a made-up programming language: juniper
juniper comes AFTER  c++
juniper comes AFTER  go
juniper comes AFTER  java
juniper comes BEFORE lisp
juniper comes BEFORE python
juniper comes BEFORE ruby
juniper comes BEFORE visualbasic
```

(Of course I couldn't resist slipping in something. On line 12 instead of using the Scanner object's .next() method to read in a String, I used the Scanner object's .nextLine() method to read in a String. The difference is that .next() will stop reading if you type a space, so if you typed "visual basic" it would only read in "visual" and leave the rest behind. When you use .nextLine() it reads in *everything* you type including spaces and tabs – up until you press Enter – and puts it all into one long String and stores it into the variable *name*.)

You compare Strings to each other using the String object's .compareTo() method. The .compareTo() method doesn't work the way you probably expect, but there is genius in how it works.

The comparison involves two Strings. The first String is the one to the left of the .compareTo(). The second String is the one in the parentheses. And the comparison simplifies to an integer! If we call the first one *self* and the second one *other* it would look like this:

```
int n = self.compareTo(other);
```

So *self* compares itself to *other*. If *self* is identical to *other* (the same length and every character the same), then *n* would be set to 0. If *self* comes before *other* alphabetically, then *n* would be set to a negative number (a number less than 0). And if *self* comes after *other* alphabetically, then *n* would be set to a positive number (a number greater than 0).

The genius part is this: because .compareTo() gives us an integer instead of just a Boolean true or false, we only need this one method to do all the comparisons: less than, greater than, less than or equal to, whatever.

Since if *self* is equal to *other* we would get zero and since if *self* is less than

other we would get a number less than zero, we can write:

```
if ( self.compareTo(other) <= 0 )
```

If the result is less than zero the if statement would be true and if the result is equal to zero the if statement would be true. So this is kind-of like writing

```
if ( self <= other )
```

…except that what I just wrote won't actually compile and the .compareTo() trick will. Pretty cool, if you ask me.

```
if ( self.compareTo(other) <  0 )  // true when self <  other
if ( self.compareTo(other) <= 0 )  // true when self <= other
if ( self.compareTo(other) >  0 )  // true when self >  other
if ( self.compareTo(other) >= 0 )  // true when self >= other
if ( self.compareTo(other) == 0 )  // true when self == other
if ( self.compareTo(other) != 0 )  // true when self != other
```

So that's the idea. Pretty confusing for beginners, and slightly harder to use but not bad at all once you get used to it.

The other difficulty here (and this is not just a .compareTo() thing, it happens everywhere in code unless you write code to fix it) is that capitalization matters. "Bob" is not the same value as "bob". And what's worse is that because of the Unicode values[7] of the letters, "Bob" comes *before* "bob" alphabetically. If you want to avoid this issue, there are a lot of ways, but I like one of these two:

```
if ( self.toLowerCase().compareTo( other.toLowerCase() ) < 0 ) // or
if ( self.compareToIgnoreCase(other) < 0 )
```

Or you can let the human type in whatever they want and convert it to lower-case right away and then only compare it to lower-case things in your code.

Study Drills

1. Using the method of your choice, make this program work correctly even if the human types in words with the "wrong" capitalization.

[7] Computers only work with numbers internally. Letters are not numbers, but there is a big giant table that maps every character in every language ever to one of 1,112,063 numbers that uniquely identifies that character. The UTF-8 Unicode value of the letter "B" is 66; the value of the letter "b" is 98.

Exercise 24: Choosing Numbers Randomly

We're going to spend a few exercises on something you don't always see in programming books: how to have the computer choose a "random" number within a certain range. This is because you can write a *lot* of software without needing the computer to randomly pick a number. However, having random numbers will let us make some simple interactive games, and that is easily worth the pain of this slightly weird concept.

```
1 public class RandomNumbers
 2 {
 3     public static void main( String[] args )
 4     {
 5         int a, b, c;
 6         double x, y, z;
 7
 8         x = Math.random();
 9         y = Math.random();
10         z = Math.random();
11
 12         System.out.println( "x is " + x );
13         System.out.println( "y is " + y );
14         System.out.println( "z is " + z );
15
 16         x = Math.random() * 100;
17         y = Math.random() * 100;
18         z = Math.random() * 100;
19
 20         System.out.println( "\nx is " + x );
21         System.out.println( "y is " + y );
22         System.out.println( "z is " + z );
23
 24         a = (int)x;
25         b = (int)y;
26         c = (int)z;
27
 28         System.out.println( "\na is " + a );
29         System.out.println( "b is " + b );
30         System.out.println( "c is " + c );
31
 32         x = 0.9999999999999999;
33         a = (int)(x * 100);
34
 35         System.out.println( "\nx is " + x );
36         System.out.println( "a is " + a );
37
 38         x = Math.random();
39         a = 0 + (int)(x*10);
40         b = 1 + (int)(x*10);
41         c = 5 + (int)(x*10);
42
 43         System.out.println( "\na is " + a );
44         System.out.println( "b is " + b );
45         System.out.println( "c is " + c );
46     }
47 }
```

What You Should See

```
x is 0.5371428668784091
y is 0.4716636154720313
z is 0.9791002546275134

x is 57.33269918363617
y is 44.731436970719386
z is 75.79286027183542

a is 57
b is 44
c is 75

x is 0.9999999999999999
a is 99

a is 6
b is 7
c is 11
```

Note: Your output won't look the same as mine. The numbers are *random*, remember?

Java has a built-in function called Math.random(). Every time you call the function, it will produce a new random double in the range [0,1) (that is, it might be exactly 0 but will never be exactly 1 and will most likely be something in between. So if I write:

```
double x = Math.random();
```

…then *x* could have a value of 0, or 0.123544, or 0.3, or 0.999999999 but never 1.0 and never greater than 1. So on lines 8 through 10 the function Math.random() is called three times, and the result is stored into three different variables. These three values are printed so you can see what they are.

Unfortunately, *I* don't often want a *double* from [0,1). Imagine a number-guessing game where you say "I am thinking of a number with decimals between zero and one: try to guess it!" No fun. And we can't control the range of the value that Math.random() gives us, so we have to mush it into a range ourselves.

On lines 16 through 18 we pick a new random number, but we multiply it by 100 before storing it into a variable. (This has the effect of moving the decimal two place values to the right.) So we can tell that the original random number printed out on line 20 was 0.5733269918363617 because it is 57.33269918363617 after being multiplied by 100.

Note that multiplying by 100 still gives us a slight chance of getting exactly 0. If the original random number is 0 then multiplying it doesn't change that. The numbers we store into the variables could be something like 12.3544, or 30.0, or 99.9999999 but never 100.0 and never anything greater than 100.

On lines 24 through 26 we perform what is called a "typecast" or just "a cast".

The variable *x* is a double: it can hold numbers with decimals. The variable *a* is an integer: it can hold whole numbers only. Normally you're not allowed to store the value from a double into an int. A cast tells the compiler "I know that *x* is a double and that I'm trying to store its value into an int which can't hold decimals. But I don't care. Why don't you just pretend that the value in *x* is an integer? It is okay if you have to throw away everything after the decimal point. Just do it."

So on line 24 the computer makes a copy of the value from *x* but everything after the decimal point is chopped off and thrown away ("truncated") and that new integer value is stored into the variable *a*. (The value of *x* is unchanged.) The value is *not* rounded; it's truncated.

Theoretically what does this give us? If *x* was originally 0 or 12.3544 or 30.0, or 99.9999999, then *a* will be 0 or 12 or 30 or 99 but never 100 or anything bigger than 100. So *a*, *b* and *c* will **always** have integer values from 0 to 99.

On lines 32 and 33 I have attempted to show that casting from a double to an integer does *not* round; the numbers after the decimal point are truncated.

Finally, on lines 38 through 41 a single random number is chosen. In all three cases it is multiplied by 10 and then cast to an integer. This means that after the cast we always have a number from 0 to 9.

But on line 39 that random number from 0 to 9 gets 0 added to it before storing it into *a*. (Adding 0 doesn't change the number.) So *a* will always have a value from 0-9.

On line 40 the random number from 0 to 9 gets 1 added to it before storing it into *b*. This makes whatever it is bigger by 1. If it had been a 0, it will now be a 1. If it had been a 6, it will now be a 7. If it had been a 9 (the maximum), it will now be a 10. So *b* will always have a value from 1-10.

On line 41 the random number from 0-9 gets 5 added to it before storing it into *c*. So *c* will always have a value from 5 to 14. (This is still ten values.)

Okay, that's enough for today.

Study Drills

1. Remove the cast from line 24. Try to compile the program. What error message do you get? (Then put it back.)

2. Run the program several times and confirm that *a*, *b* and *c* as printed out on lines 28 through 30 always have values from 0 to 99.

1. Use your fingers to count and confirm that if I have a number from 0 to 9, there are ten possible numbers I could have. Multiplying a random number by ten and truncating gives you ten possible results (0-9). Multiplying a random number by five and truncating gives you five possible results (0-4).

51. Run the program several times and confirm that *a* as printed out on line 43 always has a value from 0-9, that *b* always has a value from 1-10, and *c* always has a value from 5-14.

Exercise 25: More Complex Random Numbers

The previous exercise had some tough thinking in it, so instead of teaching something new, this exercise will just spend more time with the same concepts.

```
1 public class RandomNumbers2
 2 {
 3    public static void main( String[] args )
 4    {
 5       int a, b, c, d, e, low, high;
 6
 7       a = 1 + (int)(Math.random()*10);
 8       b = 1 + (int)(Math.random()*10);
 9       c = 1 + (int)(Math.random()*10);
10       d = 1 + (int)(Math.random()*10);
11       e = 1 + (int)(Math.random()*10);
12
 13       System.out.println( a + "\t" + b + "\t" + c + "\t" + d + "\t" + e );
14
 15       a = 1 + (int)(Math.random()*100);
16       b = 1 + (int)(Math.random()*100);
17       c = 1 + (int)(Math.random()*100);
18       d = 1 + (int)(Math.random()*100);
19       e = 1 + (int)(Math.random()*100);
20
 21       System.out.println( a + "\t" + b + "\t" + c + "\t" + d + "\t" + e );
22
 23       a = 70 + (int)(Math.random()*31); // 31 is 100-70+1
24       b = 70 + (int)(Math.random()*31);
25       c = 70 + (int)(Math.random()*31);
26       d = 70 + (int)(Math.random()*31);
27       e = 70 + (int)(Math.random()*31);
28
 29       System.out.println( a + "\t" + b + "\t" + c + "\t" + d + "\t" + e );
30
 31       low  =  70;
32       high = 100;
33
 34       a = low + (int)(Math.random()*(high-low+1));
35       b = low + (int)(Math.random()*(high-low+1));
36       c = low + (int)(Math.random()*(high-low+1));
37       d = low + (int)(Math.random()*(high-low+1));
38       e = low + (int)(Math.random()*(high-low+1));
39
 40       System.out.println( a + "\t" + b + "\t" + c + "\t" + d + "\t" + e );
41    }
42 }
```

What You Should See

```
7     9     3     3     3
10    53    78    17    75
76    96    99    85    86
99    91    96    99    83
```

(Again, you won't see this. The numbers will be random.)

On lines 7 through 11 we choose five random numbers. Each number is multiplied by ten and cast to an integer to truncate it (so each random number is one of ten numbers: 0 through 9). Then 1 is added to each, so the variables *a* through *e* each get a random number from 1 to 10.

On lines 15 through 19 we choose five random numbers again. Each number is multiplied by one hundred and cast to an integer to truncate it (so each random number is one of 100 numbers: 0 through 99). Then 1 is added to each, so the variables *a* through *e* each get a random number from 1 to 100.

On lines 23 through 27 we choose five more random numbers. Each number is multiplied by 31 and cast to an integer to truncate it (so each random number is one of 31 numbers: 0 through 30). Then 70 is added to each. 0 plus 70 gives 70. 1 plus 70 gives 71. 23 plus 70 gives 93. 30 (the maximum) plus 70 gives 100. So the variables *a* through *e* each get a random number from 70 to 100.

So the general formula is this:

```
int a = low + (int)(Math.random() * range);
```

low is the smallest possible number we want. *range* is **how many** random numbers should be in the range. If you know your lowest possible number and your highest possible number but not how many numbers that is, the formula is:

```
int range = high - low + 1;
```

If the lowest random number I want is 1 and the highest random number is 5, then the *range* is five. 5 minus 1 is 4, then add one to account for the fact that subtracting gives you the distance between two numbers, not the count of stopping points along the way.[8]

You could even write the formula like this:

```
int a = low + (int)(Math.random()*(high-low+1));
```

This will have the computer pick a random number from *low* to *high*. And this is exactly what we do on lines 34 through 38.

Study Drills

1. Change the values of *low* and *high* on lines 31 and 32 to something else. Compile it and run the program many times to confirm that you always get random numbers in that range.

[8] There is a common logical error in programming that can occur if you accidentally count distance instead of stops; it is called the "fencepost problem." The name comes from the following brain-teaser: if I need to build a fence five meters long using a bunch of wooden boards that are slightly longer than one meter each, how many fence posts will I need? You need five boards but six fence *posts*.

Exercise 26: Repeating Yourself with the While Loop

This is one of my favorite exercises, because you are going to learn how to make chunks of code *repeat*. And if you can do that, you will be able to write all *sorts* of interesting things.

```
1 import java.util.Scanner;
 2
 3 public class EnterPIN
 4 {
 5     public static void main( String[] args )
 6     {
 7         Scanner keyboard = new Scanner(System.in);
 8         int pin, entry;
 9
 10         pin = 12345;
11
 12         System.out.println("WELCOME TO THE BANK OF JAVA.");
13         System.out.print("ENTER YOUR PIN: ");
14         entry = keyboard.nextInt();
15
 16         while ( entry != pin )
17         {
18             System.out.println("\nINCORRECT PIN. TRY AGAIN.");
19             System.out.print("ENTER YOUR PIN: ");
20             entry = keyboard.nextInt();
21         }
22
 23         System.out.println("\nPIN ACCEPTED. YOU NOW HAVE ACCESS TO YOUR ACCOUNT.");
24     }
25 }
```

What You Should See

```
WELCOME TO THE BANK OF JAVA.
ENTER YOUR PIN: 123

INCORRECT PIN. TRY AGAIN.
ENTER YOUR PIN: 1234

INCORRECT PIN. TRY AGAIN.
ENTER YOUR PIN: 12345

PIN ACCEPTED. YOU NOW HAVE ACCESS TO YOUR ACCOUNT.
```

On line 16 you get your first look at the while loop. A while loop is similar to an if statement. They both have a condition in parentheses that is checked to see if it true or false. If the condition is false, both while loops and if statements will skip all the code in the body. And when the condition is true, both while loops and if statement will execute all of the code inside their body one time.

The only difference is that if statements that are true will execute all of the code in the curly braces exactly once. while loops that are true will execute all of the code in the curly braces once and then *go back up and check the condition*

again. If the condition is *still* true, the code in the body will all be executed again. Then it checks the condition *again* and runs the body again if the condition is still true.

In fact, you could say that the while loop executes all of the code in its body over and over again *as long as* its condition is true when checked.

Eventually, the condition will be false when it is checked. Then the while loop will skip over all the code in its body and the rest of the program will continue. Once the condition of a while loop is false, it doesn't get checked again.

Looping is *so* great because we can finally do something more than once without having to type the code for it more than once! In fact, programmers sometimes say "Keep your code D.R.Y: Don't Repeat Yourself." Once you know how to program pretty well and finish all of the exercises in this book, you will start to get suspicious if you find yourself typing (or copying-and-pasting) the exact same code more than once in a program.

Exercise 27: A Number-Guessing Game

Now that you know how to repeat something using a while loop we are going to write a program that another human might actually *enjoy* running? Are you as excited as I am about this?

```
1 import java.util.Scanner;
 2
 3 public class HighLow
 4 {
 5     public static void main( String[] args )
 6     {
 7         Scanner keyboard = new Scanner(System.in);
 8         int secret, guess;
 9
 10         secret = 1 + (int)(Math.random()*100);
11
 12         System.out.println( "I'm thinking of a number between 1-100. Try to guess it." );
13         System.out.print( "> " );
14         guess = keyboard.nextInt();
15
 16         while ( secret != guess )
17         {
18             if ( guess < secret )
19             {
20                 System.out.println( "Sorry, your guess is too low. Try again." );
21             }
22             if ( guess > secret )
23             {
24                 System.out.println( "Sorry, your guess is too high. Try again." );
25             }
26             System.out.print( "> " );
27             guess = keyboard.nextInt();
28         }
29
 30         System.out.println( "You guessed it! What are the odds?!?" );
31     }
32 }
```

What You Should See

```
I'm thinking of a number between 1-100. Try to guess it.
> 50
Sorry, your guess is too high. Try again.
> 25
Sorry, your guess is too high. Try again.
> 13
Sorry, your guess is too low. Try again.
> 20
Sorry, your guess is too high. Try again.
> 16
Sorry, your guess is too high. Try again.
> 18
Sorry, your guess is too high. Try again.
> 14
Sorry, your guess is too low. Try again.
```

```
> 15
You guessed it! What are the odds?!?
```

So on line 10 the computer chooses a random number from 1 to 100 and stores it into the variable *secret*. We let the human make a guess.

Line 16 has a while loop. It says "As long as the value of the variable *secret* is not the same as the value of the variable *guess*… run the following chunk of code." Lines 17 through 28 are the body of the loop. Every time the condition is true, all twelve of these lines of code get executed.

Inside the body of the loop we have a couple of if statements. We already know that the human's guess is different from the secret number or we wouldn't be inside the while loop to begin with! But we don't know if the guess is wrong because it is too low or because it is too high, so these if statements figure that out and display the appropriate error message.

Then after the error message is displayed, on line 27 we allow them to guess again. The human (hopefully) types in a number which is then stored into the variable *guess*, overwriting their previous guess in that variable.

Then the program loops back up to line 16 and checks the condition again. If the condition is *still* true (their guess is still not equal to the secret number) then the whole loop body will execute again. If the condition is now false (they guessed it) then the whole loop body will be skipped and the program will skip down to line 29.

If the loop is over, we know the condition is false. So we don't need a new if statement down here; it is safe to print "you guessed it."

Exercise 28: Infinite Loops

One thing which sometimes surprises students is how easy it is to make a loop that repeats *forever*. These are called "infinite loops" and we sometimes make them on purpose[9] but usually they are the result of a logical error. Here's an example:

```
1 import java.util.Scanner;
 2
 3 public class KeepGuessing
 4 {
 5    public static void main( String[] args )
 6    {
 7       Scanner keyboard = new Scanner(System.in);
 8       int secret, guess;
 9
 10       secret = 1 + (int)(Math.random()*10);
11
 12       System.out.println( "I have chosen a number between 1 and 10. Try to guess it." );
13       System.out.print( "Your guess: " );
14       guess = keyboard.nextInt();
15
 16       while ( secret != guess )
17       {
18          System.out.println( "That is incorrect. Guess again." );
19          System.out.print( "Your guess: " );
20       }
21
 22       System.out.println( "That's right! You're a good guesser." );
23    }
24 }
```

What You Should See

```
I have chosen a number between 1 and 10. Try to guess it.
Your guess: 4
That is incorrect. Guess again.
Your guess: That is incorrect. Guess again.
Your guess: That is incorrect. Guess again.
Your guess: That is incorrect. Guess again.
Your guess: That is incorrect. Guess again.
Your guess: That is incorrect. Guess again.
Your guess: That is incorrect. Guess again.
Your guess: That is incorrect. Guess again.
Your guess: That is incorrect. Guess again.
Your guess: That is incorrect. Guess again.
Your guess: That is incorrect. Guess again.
Your guess: That is incorrect. Guess again.
Your guess: That is incorrect. Guess again.
Your guess: That is incorrect. Guess again.
```

[99] For example, in college one of my assignments in my Network Protocols class was to write a web server. Web servers listen to the network for a page request. Then they find the requested page and send it over the network to the requesting web browser. And then they wait for another request. I used an infinite loop on purpose in that assignment because the web server software was intended to start automatically when the machine booted, run the whole time, and only shut down when the machine did.

```
Your guess: That is incorrect. Guess again.
Your guess: That is incorrect. Guess again.
Your guess: That is incorrect. Guess again.
Your guess: That is incorrect. Guess again.
Your guess: That is incorrect. Guess again.
Your guess: That is incorrect. Guess again.
Your guess: That is incorrect. Guess again.
Your guess: That is incorrect. Guess again.
Your guess: That is incorrect. Guess again.
Your guess: That is incorrect. Guess again.
Your guess: That is incorrect. Guess again.
Your guess: That is incorrect. Guess again.
Your guess: That is incorrect. Guess again.
Your guess: That is incorrect. Guess again.
Your guess: That is incorrect. Guess again.
Your guess: That is incorrect. Guess again.
Your guess: That is incorrect. Guess again.
Your guess: That is incorrect. Guess again.
Your guess: ^C
```

The program actually didn't stop on it's own; I had to stop it by pressing CTRL-C while the program was repeating and repeating.

This code has an infinite loop in it. Line 16 checks to see if the value of the variable *secret* is different than the value of the variable *guess*. If so, it executes the body of the loop and if not it skips over the loop body down to line 21.

The problem is that once *secret* and *guess* are different the program can never reach another line of code that changes either variable, so the loop repeats lines 16 through 20 forever.

So when you are writing the condition of a while loop, try to keep in the back of your mind: "I need to make sure there's a way for this condition to eventually become false."

Study Drills

1. Fix the code so that it no longer makes an infinite loop.

Exercise 29: Using Loops for Error-Checking

So far in this book we have mostly been ignoring error-checking. We have assumed that the human will follow directions, and if their lack of direction-following breaks our program, we just blame the user and don't worry about it.

This is totally fine when you are just learning. Error-checking is hard, which is why most big programs have bugs and it takes a whole army of people working really hard to make sure that software has as few bugs as possible.

But you are finally to the point where you *can* code a little bit of error-checking.

```
1 import java.util.Scanner;
 2
 3 public class SafeSquareRoot
 4 {
 5     public static void main( String[] args )
 6     {
 7         Scanner keyboard = new Scanner(System.in);
 8         double x, y;
 9
 10         System.out.print("Give me a number, and I shall find the square root of it. ");
11         System.out.print("(No negatives, please.) ");
12         x = keyboard.nextDouble();
13
 14         while ( x < 0 )
15         {
16             System.out.print("Sorry, I won't take the square root of a negative.\nNew number:
");
17             x = keyboard.nextDouble();
18         }
19
 20         y = Math.sqrt(x);
21
 22         System.out.println("The square root of "+x+" is "+y);
23     }
24 }
```

What You Should See

```
Give me a number, and I shall find the square root of it. (No negatives, please.) -8
Sorry, I won't take the square root of a negative.
New number: -7
Sorry, I won't take the square root of a negative.
New number: -200
Sorry, I won't take the square root of a negative.
New number: 5
The square root of 5.0 is 2.23606797749979
```

Starting on line 14 is an example of what I call an "input protection loop." On line 20 we are going to take the square root of whatever value is in the variable *x* and we would like to make sure it contains a positive number before we do that. (Java does not have built-in support for imaginary numbers.) We could just

use the built-in absolute value function Math.abs(), but I'm trying to demonstrate error-checking, okay?

On line 12 we let the human type in a number. We have asked them nicely to only type in a positive number, but they can type whatever they like. (They could even type "Mister Mxyzptlk", but our error-checking skill aren't advanced enough to survive that, yet.)

So on line 14 we check to see if they followed directions. If the value in *x* is negative (less than zero) we print out an error message and let them try again. THEN, after they have typed their new number we *go back up* to line 14 and check if the condition is still true. Are they still failing to follow directions? If so, display the error message again and give them another chance.

Computers don't get impatient or bored, so the human is **trapped** in this loop until they comply. They could type negative numbers two billion times and each time the computer would politely complain and make them type something again.

Eventually, the human will wise up and type a non-negative number. Then the condition of the while loop will be false (finally) and the body of the loop will be skipped (finally) and execution will pick up on line 20 where we can safely take the square root of a number that we *know* is positive.

Real programs have stuff like this *all over*. You have to do it because humans are unreliable and often do unexpected things. What happens when your toddler pulls himself up to your laptop and starts mashing keys while a program is running? We would like the program to not crash.

Oh, and did you notice? I changed up something in this program. So far every time in this book I have printed something on the screen, I have put a blank space between the parentheses and the quotation marks, like so:

```
System.out.println( "This is a test." );
```

I did that because I wanted to make it clear that the thing inside the quotes (technically a "String literal") was one thing, and the parentheses were another. But Java doesn't actually care about those spaces. (Remember that I am full of lies.) Your program will still compile and work exactly the same if you leave the spaces out:

```
System.out.println("This is a test.");
```

You might have noticed that on line 22 I even left out the spaces between the String literals and the plus signs. Spacing like this doesn't affect the syntax of a Java program, although many companies and other software-writing groups have "style guidelines" that tell you what the Right Way to format your code is if you want to make other members of the group happy.

Some people get very worked up about this. They think you should always use spaces to indent your code, or always put the open brace of a code block at the end of the previous line:

```
if ( age < 16 ) {
  allowed = false;
}
```

…like that. I think for your own code, you should give other styles a try and do what makes you happy. And when you're working with others, you should format the code in a way that makes them happy. There are even tools that can change the format of your code automatically to fit a certain style! (Search for "source code beautifier" or "Java pretty printer" to see some examples.)

Study Drills

1. Instead of an input protection loop use an if statement and Math.abs() to handle taking the square root of negative numbers, too. Detect when the number is negative, take the square root of the positive version, and print a little "i" next to the answer.

Exercise 30: Do-While loops

In this exercise I am going to do something I normally don't do. I am going to show you *another* way to make loops in Java. Since you have only been looking at while loop for four exercises, showing you a different type of loop can be confusing. Usually I like to wait until students have been doing something a *long* time before I show them a new way to do the same thing.

So if you think you are going to be confused, feel free to skip this exercise. It won't hurt you hardly at all, and you can come back to it when you're feeling more confident.

Anyway, there are several ways to make a loop in Java. In addition to the while loop, there is also a do-while loop. They are virtually identical because they both check a condition in parentheses. If the condition is true, the body of the loop is executed. If the condition is false, the body of the loop is skipped (or the looping stops).

So what's the difference? Type in the code and then we'll talk about it.

```
1 import java.util.Scanner;
 2
 3 public class CoinFlip
 4 {
 5     public static void main( String[] args )
 6     {
 7         Scanner keyboard = new Scanner(System.in);
 8
 9         String coin, again;
10         int flip, streak = 0;
11
12         do
13         {
14             flip = 1 + (int)(Math.random()*2);
15
16             if ( flip == 1 )
17                 coin = "HEADS";
18             else
19                 coin = "TAILS";
20
21             System.out.println( "You flip a coin and it is... " + coin );
22
23             if ( flip == 1 )
24             {
25                 streak++;
26                 System.out.println( "\tThat's " + streak + " in a row...." );
27                 System.out.print( "\tWould you like to flip again (y/n)? " );
28                 again = keyboard.next();
29             }
30             else
31             {
32                 streak = 0;
33                 again = "n";
34             }
35         } while ( again.equals("y") );
36
37         System.out.println( "Final score: " + streak );
38     }
```

```
39 }
```

What You Should See

```
You flip a coin and it is... HEADS
      That's 1 in a row....
      Would you like to flip again (y/n)? y
You flip a coin and it is... HEADS
      That's 2 in a row....
      Would you like to flip again (y/n)? y
You flip a coin and it is... HEADS
      That's 3 in a row....
      Would you like to flip again (y/n)? n
Final score: 3
```

There are only two differences between while loops and do-while loops.

1. The condition of a while loop is *before* the body, but do-while loops just have the keyword do before the body and the condition is at the end, just after the close curly brace. (And there's a semicolon after the close paren of the loop condition, which while loops don't have.)
52. while loops check their condition *before* going into the loop body, but do-while loops run the body of the loop once *no matter what* and only check the condition *after* the first time through.

In computer-science circles, the while loop is called a "pre-test" loop (because it checks the condition first) and the do-while is called a "post-test" loop (because it checks the condition afterward).

If the condition of the while loop is true the very first time it is checked, then code using a while loop and equivalent code using a do-while loops will behave exactly the same. Anything you can do with a while loop you could do with a do-while loop (and slightly different code) and vice-versa.

So why would the developers of Java bother to make a do-while loop? Because sometimes what you're checking in the condition is something you don't really know until you have gone through the body of the loop at least once.

In this case, we are flipping a coin by picking a random number between 1-2 and using an if statement. Then we ask them if they want to flip again or stop. The condition of our loop repeats if they say they want to flip again.

If we had done this with a while loop, the condition would look like this:

```
while ( again.equals("y") )
{
```

This is fine, and would work, but the variable *again* doesn't get a value until line 28. And so our code wouldn't compile because *again* (in the words of the Java compiler) "might not have been initialized." And so we would have to give it a value up before the loop that doesn't mean anything and is only there to please the compiler.

That is annoying, so the do-while loop allows us to leave our condition the same but wait until the end to check it. This is handy.

Study Drills

1. Change the code so that it uses a while loop instead of a do-while loop. Make sure it compiles and works the same.
53. Change it back to a do-while loop. (You might look back at this code later when you forget how to write a do-while loop and we don't want your only example to have been changed to a while loop.)

Exercise 31: Adding Values One at a Time

This exercise will demonstrate something that you have to do a *lot*: dealing with values that you only get one at a time.

If I asked you to let the human type in three numbers and add them up, and if I promised they would only need to type in exactly three numbers (never more, never fewer), you would probably write something like this:

```
int a, b, c, total;
a = keyboard.nextInt();
b = keyboard.nextInt();
c = keyboard.nextInt();
total = a + b + c;
```

If I told you the human was going to type in *five* numbers, your code might look like this:

```
double num1, num2, num3, num4, num5, total;
num1 = keyboard.nextDouble();
num2 = keyboard.nextDouble();
num3 = keyboard.nextDouble();
num4 = keyboard.nextDouble();
num5 = keyboard.nextDouble();
total = num1+num2+num3+num4+num5;
```

But what if I told you they wanted to type in one hundred numbers? Or ten thousand? Or *maybe* three and *maybe* five, I'm not sure? Then you need a different approach. You'll need a loop (that's how we repeat things, after all). And you need a variable that will add the values one at a time as they come. A variable that starts with "nothing" in it and adds values one at a time is called an "accumulator" variable, although that's a pretty old word and so your friends who code may never have heard it if they're under the age of forty.

Anyway, the basic idea looks like this:

```
1 import java.util.Scanner;
 2
 3 public class RunningTotal
 4 {
 5     public static void main( String[] args )
 6     {
 7         Scanner keyboard = new Scanner(System.in);
 8
 9         int current, total = 0;
10
 11          System.out.print("Type in a bunch of values and I'll add them up. ");
12          System.out.println("I'll stop when you type a zero.");
13
 14          do
15          {
16              System.out.print("Value: ");
17              current = keyboard.nextInt();
18              int newtotal = current + total;
19              total = newtotal;
```

```
20            System.out.println("The total so far is: " + total);
21        } while ( current != 0 );
22
 23        System.out.println("The final total is: " + total);
24    }
25 }
```

What You Should See

```
Type in a bunch of values and I'll add them up. I'll stop when you type a zero.
Value: 3
The total so far is: 3
Value: 4
The total so far is: 7
Value: 5
The total so far is: 12
Value: 6
The total so far is: 18
Value: 0
The total so far is: 18
The final total is: 18
```

We need two variables: one to hold the value they just typed in (*current*) and one to hold the running total (um… *total*). On line 9, we make sure to start by putting a zero into *total*. You'll see why soon.

On line 17 the human gets to type in a number. This is inside the body of a do-while loop, which runs at least once no matter what, so this code always happens. Let's pretend they type 3 at first.

On line 18 the first half of the magic happens. We declare a new variable called *newtotal* and set its value equal to the number the human just typed plus *whatever value is already in the variable* total. There's a zero in *total* at first, so this line of code adds zero to *current* and stores that number into *newtotal*.

Then on line 19 the second half of the magic happens: we *replace* the value in *total* (the zero) with the current value of *newtotal*. So now *total* no longer has a zero in it; it has the same value *current* did. So *total* was 0, now it is 3.

Then we print the subtotal and on line 21 check to see if *current* was zero. If not, the loop repeats back up line 14.

The human gets to type in a second number. Let's say it is a 4. The variable *newtotal* gets initialized to *current* (4) plus *total* (3), so *newtotal* is 7. Then on line 19 we change *total*'s value to 7.

The condition is checked again, and the process continues. Eventually the human types a 0, that 0 gets added to the total (which doesn't hurt it) and the condition is false so the do-while loop stops looping.

I should mention two things before the exercise ends:

1. Because the variable *newtotal* is declared (and defined) on line 18, the scope of that variable is limited to the body of the do-while loop. That means on line 21, *newtotal* is no longer in scope, so any attempt to reference *newtotal* in the condition of the do-while loop would give an error. The variable keeps getting

created and destroyed each time through the loop. This is sort-of inefficient.

2. We could have coded this without even using a *newtotal* variable. Since Java figures out the final value of the right-hand side before storing it into the variable named on the left-hand side, we could have combined lines 18 and 19 into a single line:

```
total = current + total;
```

This works totally fine. (In fact, you have seen it before.)

Study Drills

1. Rewrite the code to use a while loop instead of a do-while loop. Get it to compile and make sure it still works. Then change it back.
54. Change the condition of the do-while loop so that the loop stops when *newtotal* is exactly 20. Oh? It doesn't compile because *newtotal* is out of scope? Change where *newtotal* is declared so that this works.

Exercise 32: Adding Values for a Dice Game

Pig is a simple dice game for two or more players. (You can read the Wikipedia entry for Pig if you want a lot more information.) The basic idea is to be the first one to "bank" a score of 100 points. When you roll a 1, your turn ends and you gain no points that turn. Any other roll adds to your score for that turn, but *you only keep those points* if you decide to "hold". If you roll a 1 before you hold, all your points for that turn are lost.

You know enough to handle the code for the entire game of Pig, but it is a *lot* at once compared to the smaller programs you have been seeing, so I am going to break it into two lessons. Today we will write only the artificial intelligence (A.I.) code for a computer player. This computer player will utilize the "hold at 20" strategy, which means the computer keeps rolling until their score for the turn adds up to 20 or more, and then holds no matter what. This is actually not a terrible strategy, and it is easy enough to code.

```
1 import java.util.Scanner;
 2
 3 public class PigDiceComputer
 4 {
 5     public static void main( String[] args )
 6     {
 7         Scanner keyboard = new Scanner(System.in);
 8
 9         int roll, total;
10
 11         total = 0;
12
 13         do
14         {
15             roll = 1 + (int)(Math.random()*6);
16             System.out.println( "Computer rolled a " + roll + "." );
17             if ( roll == 1 )
18             {
19                 System.out.println( "\tThat ends its turn." );
20                 total = 0;
21             }
22             else
23             {
24                 total += roll;
25                 System.out.println( "\tComputer has " + total + " points so far this round." );
26                 if ( total < 20 )
27                 {
28                     System.out.println( "\tComputer chooses to roll again." );
29                 }
30             }
31         } while ( roll != 1 && total < 20 );
32
 33         System.out.println( "Computer ends the round with " + total + " points." );
34
 35     }
36 }
```

What You Should See

```
Computer rolled a 2.
        Computer has 2 points so far this round.
        Computer chooses to roll again.
```

```
Computer rolled a 3.
    Computer has 5 points so far this round.
    Computer chooses to roll again.
Computer rolled a 1.
    That ends its turn.
Computer ends the round with 0 points.
```

Basically the whole program is in the body one big do-while loop that tells the computer when to stop: either it rolls a 1 or it gets a total of 20 or more. As long as the roll is not one *and* the total is less than 20, the condition will be true and the loop will start over from the beginning (on line 13). And we choose a do-while loop because we want the computer to roll at least once no matter what.

The roll is made on line 15: a random number from 1-6 is a good substitute for rolling a dice.

On line 17 we check for rolling a 1. If so, all points are lost. If not (else), we add this roll to the running total. Notice we used "plus equals", which we have seen before.

The if statement on line 26 is just so we can get a nice message that the computer is going to roll again.

Not terrible, right? So come back next lesson for the full game!

Study Drills

1. Find a dice (technically it should be "die", since "dice" is plural and you only need one) or find an app or website to simulate rolling a die. Get out a sheet of paper and something to write with. Draw a line down the middle of the paper and make two columns. Label the left column "roll" and the right column "total". Put a 0 in the *total* column and leave the other column blank at first.
   Then roll the die and write down the number you rolled at the top of the *roll* column. Put the number (15) in parentheses next to the roll value, since the die roll occurs on line 15 in the code.
   Then step through the code line by line just like the computer would. Compare the current value of *roll* with 1. If they are equal, cross out the current value in the *total* column and put 0 (20) there, since *total* would become zero on line 20 of the code.
   Keep going until the program would end. Here's an example of what my table would look like for the sample run of the program shown in the "What You Should See" section.

roll	***total***
	0 (11)
2 (15)	
	2 (24)
3 (15)	

	5 (24)
1 (15)	
	0 (20)

Exercise 33: The Dice Game Called 'Pig'

In the previous lesson we wrote the computer A.I. for the dice game *Pig*. (Remember that you can read the Wikipedia entry for Pig if you want a lot more information about this game.) In this lesson we will have the code for the entire game, with one human player and one computer player that take turns.

The entire program you wrote last time corresponds roughly to lines 43 through 67 in this program. The only major difference is that instead of a single *total* variable we will have a *turnTotal* variable to hold only the points for one turn and a *total2* variable that holds the computer's overall points from round to round.

```
1 import java.util.Scanner;
 2
 3 public class PigDice
 4 {
 5     public static void main( String[] args )
 6     {
 7         Scanner keyboard = new Scanner(System.in);
 8
 9         int roll, total1, total2, turnTotal;
10         String choice = "";
11
12         total1 = 0;
13         total2 = 0;
14
15         do
16         {
17             turnTotal = 0;
18             System.out.println( "You have " + total1 + " points." );
19
20             do
21             {
22                 roll = 1 + (int)(Math.random()*6);
23                 System.out.println( "\tYou rolled a " + roll + "." );
24                 if ( roll == 1 )
25                 {
26                     System.out.println( "\tThat ends your turn." );
27                     turnTotal = 0;
28                 }
29                 else
30                 {
31                     turnTotal += roll;
32                     System.out.println( "\tYou have " + turnTotal + " points so far this round." );
33                     System.out.print( "\tWould you like to \"roll\" again or \"hold\"? " );
34                     choice = keyboard.next();
35                 }
36             } while ( roll != 1 && choice.equals("roll") );
37
38             total1 += turnTotal;
39             System.out.println( "\tYou end the round with " + total1 + " points." );
40
41             if ( total1 < 100 )
42             {
43                 turnTotal = 0;
44                 System.out.println( "Computer has " + total2 + " points." );
45
46                 do
47                 {
48                     roll = 1 + (int)(Math.random()*6);
49                     System.out.println( "\tComputer rolled a " + roll + "." );
50                     if ( roll == 1 )
51                     {
52                         System.out.println( "\tThat ends its turn." );
53                         turnTotal = 0;
54                     }
55                     else
```

```
56                {
57                   turnTotal += roll;
58                   System.out.println( "\tComputer has " + turnTotal + " points so far this round." );
59                   if ( turnTotal < 20 )
60                   {
61                      System.out.println( "\tComputer chooses to roll again." );
62                   }
63                }
64             } while ( roll != 1 && turnTotal < 20 );
65
 66             total2 += turnTotal;
67             System.out.println( "\tComputer ends the round with " + total2 + " points." );
68          }
69
 70       } while ( total1 < 100 && total2 < 100 );
71
 72       if ( total1 > total2 )
73       {
74          System.out.println( "Humanity wins!" );
75       }
76       else
77       {
78          System.out.println( "The computer wins." );
79       }
80
 81    }
82 }
```

What You Should See

```
You have 0 points.
        You rolled a 2.
        You have 2 points so far this round.
        Would you like to "roll" again or "hold"? roll
        You rolled a 1.
        That ends your turn.
        You end the round with 0 points.
Computer has 0 points.
        Computer rolled a 6.
        Computer has 6 points so far this round.
        Computer chooses to roll again.
        Computer rolled a 6.
        Computer has 12 points so far this round.
        Computer chooses to roll again.
        Computer rolled a 1.
        That ends its turn.
        Computer ends the round with 0 points.
You have 0 points.
        You rolled a 3.
        You have 3 points so far this round.
        Would you like to "roll" again or "hold"? roll
        You rolled a 5.
        You have 8 points so far this round.
        Would you like to "roll" again or "hold"? roll
        You rolled a 2.
        You have 10 points so far this round.
        Would you like to "roll" again or "hold"? roll
        You rolled a 2.
        You have 12 points so far this round.
        Would you like to "roll" again or "hold"? hold
        You end the round with 12 points.
Computer has 0 points.
```

```
        Computer rolled a 5.
        Computer has 5 points so far this round.
        Computer chooses to roll again.
        Computer rolled a 4.
        Computer has 9 points so far this round.
        Computer chooses to roll again.
        Computer rolled a 4.
        Computer has 13 points so far this round.
        Computer chooses to roll again.
        Computer rolled a 3.
        Computer has 16 points so far this round.
        Computer chooses to roll again.
        Computer rolled a 5.
        Computer has 21 points so far this round.
        Computer ends the round with 21 points.
You have 12 points.
        You rolled a 2.
        You have 2 points so far this round.
        Would you like to "roll" again or "hold"? roll

...etc

        Computer has 20 points so far this round.
        Computer ends the round with 102 points.
The computer wins.
```

We begin the program with two variables: *total1* holds the human's total and *total2* holds the computer's total. Both start at 0.

Then on line 15 begins a really huge do-while loop that basically contains the whole game and doesn't end until line 70. Scroll down and you can see that this loop repeats as long as both *total1* and *total2* are less than 100. When either player reaches 100 or more, the condition is no longer true and the do-while won't repeat back up again.

Then after that do-while loop ends (starting on line 72) there is an if statement and an else to determine the winner.

Let us scroll back up and look at the human's turn, which begins on line 17. The *turnTotal* is the number of points the human has earned this round so far. And since it's the beginning of the round, we should start it out at 0.

Line 20 is the beginning of a do-while loop that contains the human's turn. It ends on line 36, and all the code between lines 20 and 36 will repeat as long as the human does not roll a 1 and as long as the human keeps choosing to roll again.

Each roll for the human begins just like the computer did: by choosing a random number from 1 to 6. We print this out on line 22.

Now two things could happen: either the roll is 1 – and the human loses all points earned this round – or the roll is 2-6 and the roll is added to their *turnTotal*. We display the appropriate messages, and on lines 33 and 34 we give the human the choice to chance it by rolling again or play it safe by holding. Then on line 36 the condition of the do-while loop will check and repeat back up to line 20 if appropriate.

Once the player's turn ends, we add the *turnTotal* (which might be 0) to the

player's overall total and display their current number of points.

On line 41 the computer's turn begins. However, the computer doesn't get a turn if the human has already reached 100 points: the game is over in that case. So to prevent the computer from playing we must wrap the whole computer's turn in a big if statement so that it is skipped if the human's total (*total1*) is greater than or equal to 100. This if statement begins here on line 41 and ends on line 68.

So on line 43 the computer's turn begins for real. This is basically the same as the previous exercise, so I won't bother to explain it again. Notice that the computer is deciding whether or not to roll again based on its turn total.

Line 70 ends the do-while loop containing the whole game, and lines 72 through 79 determine and display the winner.

Hopefully you were able to follow the flow of the game well enough. It's pretty complicated.

I would also point out how *important* it is for a program like this that every time you put an open brace, you indent everything inside the following block one more level. It will save you a lot of grief if you can just scan your eyes visually down from the open curly brace on line 47 to the matching close curly brace on line 64 to see what is inside that do-while loop and what isn't.

Exercise 34: Calling a Function

The previous exercise was pretty complicated. So we will relax a bit with today's exercise. We are going to learn how to write a "function"[10] in Java and how to make it execute by "calling" it.

```
1 public class ThereAndBackAgain
 2 {
 3     public static void main( String[] args )
 4     {
 5         System.out.println( "Here." );
 6         erebor();
 7         System.out.println( "Back first time." );
 8         erebor();
 9         System.out.println( "Back second time." );
10     }
11
 12     public static void erebor()
13     {
14         System.out.println( "There." );
15     }
16 }
```

What You Should See

```
Here.
There.
Back first time.
There.
Back second time.
```

So lines 5 through 9 are pretty boring, except that on lines 6 and 8 we are referring to some thing called "erebor" that you haven't seen before in Java. Do you know why you haven't seen it? Because it doesn't exist!

Skipping down to lines 12 through 15 you will notice that I added something to our program that is *not* inside the body of main(). Normally the close curly brace of public static void main is almost at the end of the code, and the only thing after it is the close curly brace of public class Whatevs. But not this time!

Lines 12 through 15 define a *function* named erebor(). (The word erebor() doesn't mean anything in particular to Java. We could have named it bilbo() or

[10][10] This is one of the things where I'm full of lies. Technically, Java doesn't even *have* functions. It only has "methods" and this is a method, not a function.

But this is *only* a method because that is all Java has. In any other programming language, what we have written here would be called a function and not a method. This is because methods are an object-oriented thing and this program is not even remotely object-oriented.

So even though it's technically incorrect, I am going to refer to this sort of thing as a *function* and only use the word *method* when I make something that actually behaves like a method.

My intentionally wrong vocabulary will only cause problems if you are talking to a pedantic Java programmer because they might make fun of you. If that happens, show them this footnote and ask them how many years they have been teaching beginners to code. I promise that doing it this way is better than trying to show you real methods from the beginning or doing this now and trying to distinguish between "methods that act like functions" and "methods that behave like real methods."

smaug() or anything we like.)

This function has an open curly brace on line 13 just like main() always has an open curly brace. And on line 15 is the end of the function's body, and there's a close curly brace. So the function definition starts on line 12 and ends on line 15.

What does the function do? It prints the String "There." on the screen.

So now let's go back up to main() and look at the function calls inside the body of main().

On line 5 we print the String "Here." on the screen. Then on line 6 you will see a "function call." This line of code tells the computer to jump down to the function erebor(), run through all the code in the body of that function, and to return to line 6 once that has been accomplished.

So you see that when we call the erebor() function, the String "There." gets printed on the screen right after the String "Here.". When the computer runs line 6, execution of the program pauses in main(), skips over all the rest of the code in main(), jumps down to line 12, runs all the code in the body of the function erebor() (all 1 line of it) and then once execution hits the close curly brace on line 15, it returns back up to the end of line 6 and unpauses the execution in main(). It runs line 7 next.

On line 7 is displays another message on the screen, then on line 8 there is another function call. The function *erebor* is called a second time. It pauses main() on line 8, jumps down and runs through the body of *erebor* (which prints the String "There." again), then returns back up to line 8 where execution of main() resumes.

Finally line 9 prints one last String on the screen. Execution then proceeds to the close curly brace of main() which is on line 10. When main() ends, the program ends.

That's pretty important, so I will say it again: when main() ends, the program ends. Even if you have a bunch of different functions inside the class, program execution begins with the first line of main(). And once the last line of main() has been executed, the program stops running even if there are functions that never got called. (We will see an example of this in the next exercise.)

Study Drills

1. Remove the parentheses at the end of the first function call on line 6 so that it looks like so:

```
erebor;
```

What happens when you compile? (Then put the parentheses back.)

1. Remove the second function call (the one on line 8). You can either just delete the line entirely or put slashes in front of it so the compiler thinks it's a comment like so:

```
    // erebor();
```

Compile it, but before you run it, how do you think the output will be different? Run it and see if you were right.

Exercise 35: Calling Functions to Draw a Flag

Now that you understand the absolute basics about how to define a function and how to call that function, let us get some practice by defining *eleven* functions!

There are no zeros (0) in this program. Everything that looks like an O is a capital letter O. Also notice that lines 45 and 50 feature print() instead of println().

```
1 import static java.lang.System.*;
 2
 3 public class OverlyComplexFlag
 4 {
 5    public static void main( String[] args )
 6    {
 7       printTopHalf();
 8
 9       print48Colons();
10        print48Ohs();
11        print48Colons();
12        print48Ohs();
13        print48Colons();
14        print48Ohs();
15    }
16
 17    public static void print48Colons()
18    {
19       out.println( "|::::::::::::::::::::::::::::::::::::::::::::::::|" );
20    }
21
 22    public static void print48Ohs()
23    {
24       out.println( "|OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO|"
);
25    }
26
 27    public static void print29Colons()
28    {
29       out.println( "|:::::::::::::::::::::::::::::|" );
30    }
31
 32    public static void printPledge()
33    {
34       out.println( "I pledge allegiance to the flag." );
35    }
36
 37    public static void print29Ohs()
38    {
39       out.println( "|OOOOOOOOOOOOOOOOOOOOOOOOOOOOO|" );
40    }
41
 42    public static void print6Stars()
43    {
44       out.print( "| *  *  *  *  *  * " );
45    }
46
 47    public static void print5Stars()
48    {
```

```
49        out.print( "|   *  *  *  *  *  " );
50     }
51
 52     public static void printSixStarLine()
53     {
54        print6Stars();
55        print29Ohs();
56     }
57
 58     public static void printFiveStarLine()
59     {
60        print5Stars();
61        print29Colons();
62     }
63
 64     public static void printTopHalf()
65     {
66        out.println( " _____________________________________________" ); // 1 space
then 48 underscores
67        printSixStarLine();
68        printFiveStarLine();
69        printSixStarLine();
70        printFiveStarLine();
71        printSixStarLine();
72        printFiveStarLine();
73        printSixStarLine();
74     }
75
 76 }
```

What You Should See

```
 _____________________________________________
| *  *  *  *  *  * |OOOOOOOOOOOOOOOOOOOOOOOOOOOO|
|   *  *  *  *  *  |::::::::::::::::::::::::::::|
| *  *  *  *  *  * |OOOOOOOOOOOOOOOOOOOOOOOOOOOO|
|   *  *  *  *  *  |::::::::::::::::::::::::::::|
| *  *  *  *  *  * |OOOOOOOOOOOOOOOOOOOOOOOOOOOO|
|   *  *  *  *  *  |::::::::::::::::::::::::::::|
| *  *  *  *  *  * |OOOOOOOOOOOOOOOOOOOOOOOOOOOO|
|:::::::::::::::::::::::::::::::::::::::::::::::|
|OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO|
|:::::::::::::::::::::::::::::::::::::::::::::::|
|OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO|
|:::::::::::::::::::::::::::::::::::::::::::::::|
|OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO|
```

This exercise is ridiculous. There is no good reason that any self-respecting programmer would ever write the code to draw a flag on the screen like this. It is okay if writing this program felt a little silly. But functions are important and I prefer to start with silly examples that you can actually understand instead of realistic examples that are too hard to follow.

So how do we even trace through the execution of a program like this? We start at the beginning: the first line of main(). On line 7 the first thing *main* does is call the function printTopHalf(). So *main* gets put on pause and execution jumps

down to the first line of the printTopHalf() function, which is on line 66.

The first thing that *printTopHalf* does is print a bunch of underscores on the screen, which we will be the top of our flag. After that execution moves on to line 67, which is *another* function call! So main() is still on pause from before, waiting for printTopHalf() to finish, and now printTopHalf() itself is on pause, waiting for printSixStarLine() to finish and return control to here.

printSixStarLine() begins on line 54, where it calls the print6Stars() function. That function (thankfully) only displays something on the screen so when the close curly brace of print6Stars() comes on line 45, it returns control back to the second line (line 55) of printSixStarLine(), which then has another function call. This runs through the body of the function print29Ohs() and comes back to line 56. So then printSixStarLine() ends, which returns control to the end of line 67.

At this point, I think explaining all the function calls will be more confusing than just following the execution path on your own, so here I will just print all the line numbers that execute, in order. Calling a function will increase the indentation level and returning from that function will reduce the indentation level.

```
5 - begin main
6
7 - call printTopHalf
   64 - begin printTopHalf
      65
      66
      67 - call printSixStarLine
         52 - begin printSixStarLine
            53
            54 - call print6Stars
               42 - begin print6Stars
                  43
                  44
                  45 - end print6Stars
            54 - resume printSixStarLine
            55 - call print29Ohs
               37 - begin print29Ohs
                  38
                  39
                  40 - end print29Ohs
            55 - resume printSixStarLine
            56 - end printSixStarLine
       67 - resume printTopHalf
          68 - call printFiveStarLine
             58 - begin printFiveStarLine
                59
                60 - call print5Stars
                   47 - begin print5Stars
                      48
                      49
                      50 - end print5Stars
                60 - resume printFiveStarLine
                61 - call print29Colons
                   27 - begin print29Colons
                      28
                      29
```

```
                30 - end print29Colons
          61 - resume printFiveStarLine
          62 - end printFiveStarLine
    68 - resume printTopHalf
    69 - call printSixStarLine
   52 - begin printSixStarLine
      53
      54 - call print6Stars
         42 - begin print6Stars
            43
            44
            45 - end print6Stars
      54 - resume printSixStarLine
      55 - call print29Ohs
         37 - begin print29Ohs
            38
            39
            40 - end print29Ohs
      55 - resume printSixStarLine
      56 - end printSixStarLine
69 - resume printTopHalf
    70 - call printFiveStarLine
       58 - begin printFiveStarLine
          59
          60 - call print5Stars
             47 - begin print5Stars
                48
                49
                50 - end print5Stars
          60 - resume printFiveStarLine
          61 - call print29Colons
             27 - begin print29Colons
                28
                29
                30 - end print29Colons
          61 - resume printFiveStarLine
          62 - end printFiveStarLine
70 - resume printTopHalf
    71 - call printSixStarLine
   52 - begin printSixStarLine
      53
      54 - call print6Stars
         42 - begin print6Stars
            43
            44
            45 - end print6Stars
      54 - resume printSixStarLine
      55 - call print29Ohs
         37 - begin print29Ohs
            38
            39
            40 - end print29Ohs
      55 - resume printSixStarLine
      56 - end printSixStarLine
71 - resume printTopHalf
    72 - call printFiveStarLine
       58 - begin printFiveStarLine
          59
          60 - call print5Stars
             47 - begin print5Stars
```

```
                48
                49
                50 - end print5Stars
            60 - resume printFiveStarLine
            61 - call print29Colons
              27 - begin print29Colons
                28
                29
                30 - end print29Colons
            61 - resume printFiveStarLine
            62 - end printFiveStarLine
        72 - resume printTopHalf
          73 - call printSixStarLine
          52 - begin printSixStarLine
            53
            54 - call print6Stars
              42 - begin print6Stars
                43
                44
                45 - end print6Stars
            54 - resume printSixStarLine
            55 - call print29Ohs
              37 - begin print29Ohs
                38
                39
                40 - end print29Ohs
            55 - resume printSixStarLine
            56 - end printSixStarLine
        73 - resume printTopHalf
          74 - end printTopHalf
7 - resume main
8
9 - call print48Colons
    17 - begin print48Colons
      18
      19
      20 - end print48Colons
9 - resume main
10 - call print48Ohs
    22 - begin print48Ohs
      23
      24
      25 - end print48Ohs
10 - resume main
11 - call print48Colons
    17 - begin print48Colons
      18
      19
      20 - end print48Colons
11 - resume main
12 - call print48Ohs
    22 - begin print48Ohs
      23
      24
      25 - end print48Ohs
12 - resume main
13 - call print48Colons
    17 - begin print48Colons
      18
      19
```

```
        20 - end print48Colons
    13 - resume main
    14 - call print48Ohs
       22 - begin print48Ohs
          23
          24
          25 - end print48Ohs
    14 - resume main
```

Holy cow! If you can successfully trace through that, then you are well on your way to becoming a competent programmer.

Study Drills

1. You didn't actually trace all the way through the program, did you? Well, go back and do it. This book isn't called "Learn Java the Halfway" is it? Print out the code, grab a pencil, and draw lines whenever a function calls somewhere else and draw a line back when the function returns. When you are done it should look a bit like a plate of graphite spaghetti!

55. On lines 32 through 35 you find a definition for a function named printPledge(). But yet the output of this function never shows up. Why not? At the end of main() add a function call to run this function so that it shows up underneath the flag.

(Despite the evilness of this program, I am pretty proud of that flag. If you use a ruler to measure the dimensions of everything you will find that my flag is about as close as I think you can make it to the dimensions of a real United States flag. I actually spent quite a while measuring and adjusting everything.)

Exercise 36: Displaying Dice with Functions

The last exercise used functions in a program where functions actually made things *worse*. So today we are ready to look at a situation where using a function actually makes the program better.

Yacht is an old dice game that was modified for the commercial game Yahtzee. It involves rolling five dice at once and earning points for various combinations. The rarest combination is “The Yacht”, when all five dice show the same number.

This program doesn’t do any other scoring, it just rolls five dice until they are all the same. (Computers go fast, so even if this takes a lot of tries it doesn’t take very long.)

```
1 public class YachtDice
 2 {
 3    public static void main( String[] args )
 4    {
 5       int roll1, roll2, roll3, roll4, roll5;
 6       boolean allTheSame;
 7
 8       do
 9       {
10          roll1 = 1 + (int)(Math.random()*6);
11          roll2 = 1 + (int)(Math.random()*6);
12          roll3 = 1 + (int)(Math.random()*6);
13          roll4 = 1 + (int)(Math.random()*6);
14          roll5 = 1 + (int)(Math.random()*6);
15          System.out.println("\nYou rolled: " + roll1 + " " + roll2 + " " + roll3 + " " + roll4 + " " +
roll5);
16          showDice(roll1);
17          showDice(roll2);
18          showDice(roll3);
19          showDice(roll4);
20          showDice(roll5);
21          allTheSame = ( roll1 == roll2 && roll2 == roll3 && roll3 == roll4 && roll4 == roll5 );
22
 23       } while ( ! allTheSame );
24       System.out.println("The Yacht!!");
25    }
26
 27    public static void showDice( int roll )
28    {
29       System.out.println("+---+");
30       if ( roll == 1 )
31       {
32          System.out.println("|   |");
33          System.out.println("| o |");
34          System.out.println("|   |");
35       }
36       else if ( roll == 2 )
37       {
38          System.out.println("|o  |");
39          System.out.println("|   |");
40          System.out.println("|  o|");
41       }
42       else if ( roll == 3 )
43       {
```

```
44          System.out.println("|o  |");
45          System.out.println("| o |");
46          System.out.println("|  o|");
47       }
48       else if ( roll == 4 )
49       {
50          System.out.println("|o o|");
51          System.out.println("|   |");
52          System.out.println("|o o|");
53       }
54       else if ( roll == 5 )
55       {
56          System.out.println("|o o|");
57          System.out.println("| o |");
58          System.out.println("|o o|");
59       }
60       else if ( roll == 6 )
61       {
62          System.out.println("|o o|");
63          System.out.println("|o o|");
64          System.out.println("|o o|");
65       }
66       System.out.println("+---+");
67    }
68 }
```

What You Should See

```
You rolled: 4 6 5 5 6
+---+
|o o|
|   |
|o o|
+---+
+---+
|o o|
|o o|
|o o|
+---+
+---+
|o o|
| o |
|o o|
+---+
+---+
|o o|
| o |
|o o|
+---+
+---+
|o o|
|o o|
|o o|
+---+

You rolled: 2 2 5 6 6
+---+
```

```
|o  |
|   |
|  o|
+---+
+---+
|o  |
|   |
|  o|
+---+
+---+
|o o|
| o |
|o o|
+---+
+---+
|o o|
|o o|
|o o|
+---+
+---+
|o o|
|o o|
|o o|
+---+

...etc

You rolled: 5 5 5 5 5
+---+
|o o|
| o |
|o o|
+---+
+---+
|o o|
| o |
|o o|
+---+
+---+
|o o|
| o |
|o o|
+---+
+---+
|o o|
| o |
|o o|
+---+
+---+
|o o|
| o |
|o o|
+---+
The Yacht!!
```

Other than the fancy Boolean expression on line 21, the interesting thing in this exercise is a single function called showDice.

On lines 10 through 14 we choose five random numbers (each from 1 to 6) and

store the results into the five integer variables *roll1* through *roll5*.

We want to use some if statements to display a picture of the die's value on the screen, but we don't want to have to write the same if statements five times (which we would have to do because the variables are different). The solution is to create a function that takes a parameter.

On line 27 you see the beginning of the definition of the showDice function. After the name (or "identifier") showDice there is a set of parentheses and between them a variable is declared! This variable is called a "parameter". The showDice function has one parameter. That parameter is an integer. It is named *roll*.

This means that whenever you write a function call for showDice you can *not* just write the name of the function with parentheses like showDice(). It won't compile. You must include an integer value in the parentheses (this is called an "argument"), either a variable or an expression that simplifies to an integer value.

Here are some examples.

```
showDice;                    // NO  (without parens this refers to a variable not a function call)
showDice();                  // NO  (function call must have one argument, not zero)
showDice(1);                 // YES (one argument is just right)
showDice(4);                 // YES
showDice(1+2);               // YES
showDice(roll2);             // YES
showDice(roll5);             // YES
showDice( (roll3+roll4) / 2 );  // YES (strange but legal)
showDice(17);                // YES (although it won't show a proper dice picture)
showDice(3, 4);              // NO  (function call must have one argument, not two)
showDice(2.0);               // NO  (argument must be an integer, not a double)
showDice("two");             // NO  (argument must be an integer, not a String)
showDice(false);             // NO  (argument must be an integer, not a Boolean)
```

In all cases, a copy of the argument's value is stored into the parameter. So if you call the function like so showDice(3); then the function is called and the value 3 is stored into the parameter *roll*. So by line 29 the parameter variable *roll* has already been declared *and* initialized with the value 3.

If we call the function using a variable like showDice(roll2); then the function is called and a copy of whatever value is currently in *roll2* will have been stored into the parameter variable *roll* before the body of the function is executed.

So on line 16 the showDice function is executed, and *roll* will have been set equal to whatever value is in *roll1*.

Then on line 17 showDice is called again, but this time *roll* will be set equal to whatever value is in *roll2*. Line 18 calls showDice while setting its parameter equal to the value of *roll3*. And so on.

In this way we basically run the same chunk of code five times, but substituting a different variable for *roll* each time. This saves us a lot of code.

For comparison, I also wrote a simplified two-dice version of this exercise *without* using functions. Notice how I had to repeat the exact same sequence of if statements twice: once for each variable.

Also notice that although defining a function is a little bit more work than just copying-and-pasting the if statements and changing the variable, the two-dice version is longer than the five-dice version.

```
1 public class YachtDiceNoFunctions
 2 {
 3    public static void main( String[] args )
 4    {
 5       int roll1, roll2;
 6
 7       do
 8       {
 9          roll1 = 1 + (int)(Math.random()*6);
10           roll2 = 1 + (int)(Math.random()*6);
11           int total = roll1 + roll2;
12           System.out.println("\nYou rolled a " + roll1 + " and a " + roll2);
13           System.out.println("+---+");
14           if ( roll1 == 1 )
15           {
16              System.out.println("|   |");
17              System.out.println("| o |");
18              System.out.println("|   |");
19           }
20           else if ( roll1 == 2 )
21           {
22              System.out.println("|o  |");
23              System.out.println("|   |");
24              System.out.println("|  o|");
25           }
26           else if ( roll1 == 3 )
27           {
28              System.out.println("|o  |");
29              System.out.println("| o |");
30              System.out.println("|  o|");
31           }
32           else if ( roll1 == 4 )
33           {
34              System.out.println("|o o|");
35              System.out.println("|   |");
36              System.out.println("|o o|");
37           }
38           else if ( roll1 == 5 )
39           {
40              System.out.println("|o o|");
41              System.out.println("| o |");
42              System.out.println("|o o|");
43           }
44           else if ( roll1 == 6 )
45           {
46              System.out.println("|o o|");
47              System.out.println("|o o|");
48              System.out.println("|o o|");
49           }
50           System.out.println("+---+");
51
 52
 53            System.out.println("+---+");
54           if ( roll2 == 1 )
```

```
55          {
56             System.out.println("|   |");
57             System.out.println("| o |");
58             System.out.println("|   |");
59          }
60          else if ( roll2 == 2 )
61          {
62             System.out.println("|o  |");
63             System.out.println("|   |");
64             System.out.println("|  o|");
65          }
66          else if ( roll2 == 3 )
67          {
68             System.out.println("|o  |");
69             System.out.println("| o |");
70             System.out.println("|  o|");
71          }
72          else if ( roll2 == 4 )
73          {
74             System.out.println("|o o|");
75             System.out.println("|   |");
76             System.out.println("|o o|");
77          }
78          else if ( roll2 == 5 )
79          {
80             System.out.println("|o o|");
81             System.out.println("| o |");
82             System.out.println("|o o|");
83          }
84          else if ( roll2 == 6 )
85          {
86             System.out.println("|o o|");
87             System.out.println("|o o|");
88             System.out.println("|o o|");
89          }
90          System.out.println("+---+");
91
 92           System.out.println("The total is " + total + "\n");
93       } while ( roll1 != roll2 );
94
 95       System.out.println("Doubles!  Nice job.");
96    }
97 }
```

What You Should See

```
You rolled a 5 and a 4
+---+
|o o|
| o |
|o o|
+---+
+---+
|o o|
|   |
|o o|
+---+
```

```
The total is 9


You rolled a 6 and a 2
+---+
|o o|
|o o|
|o o|
+---+
+---+
|o  |
|   |
|  o|
+---+
The total is 8


You rolled a 2 and a 2
+---+
|o  |
|   |
|  o|
+---+
+---+
|o  |
|   |
|  o|
+---+
The total is 4

Doubles!  Nice job.
```

Study Drills

1. Add a sixth dice. Notice how easy it is to display *roll6* by just adding a single function call.

Exercise 37: Returning a Value from a Function

Some functions have parameters and some do not. Parameters are the only way to send values *into* a function. There is also only one way to get a value *out* of a function: the return value.

This exercise gives an example of a function that has three parameters (the side lengths of a triangle) and one output: the area of that triangle using Heron's Formula.

```
1 public class HeronsFormula
 2 {
 3     public static void main( String[] args )
 4     {
 5        double a;
 6
 7        a = triangleArea(3, 3, 3);
 8        System.out.println("A triangle with sides 3,3,3 has an area of " + a );
 9
 10        a = triangleArea(3, 4, 5);
11        System.out.println("A triangle with sides 3,4,5 has an area of " + a );
12
 13        a = triangleArea(7, 8, 9);
14        System.out.println("A triangle with sides 7,8,9 has an area of " + a );
15
 16        System.out.println("A triangle with sides 5,12,13 has an area of " + triangleArea(5, 12, 13) );
17        System.out.println("A triangle with sides 10,9,11 has an area of " + triangleArea(10, 9, 11) );
18        System.out.println("A triangle with sides 8,15,17 has an area of " + triangleArea(8, 15, 17) );
19     }
20
 21     public static double triangleArea( int a, int b, int c )
22     {
23        // the code in this function computes the area of a triangle whose sides have lengths a, b, and
c
24        double s, A;
25
 26        s = (a+b+c) / 2;
27        A = Math.sqrt( s*(s-a)*(s-b)*(s-c) );
28
 29        return A;
30        // ^ after computing the area, "return" it
31     }
32 }
```

What You Should See

```
A triangle with sides 3,3,3 has an area of 2.0
A triangle with sides 3,4,5 has an area of 6.0
A triangle with sides 7,8,9 has an area of 26.832815729997478
A triangle with sides 5,12,13 has an area of 30.0
A triangle with sides 10,9,11 has an area of 42.42640687119285
A triangle with sides 8,15,17 has an area of 60.0
```

You can see that the function triangleArea has three parameters. They are all integers, and they are named *a*, *b* and *c*. As you already know, this means that we cannot call the function without providing three integer values as arguments.

In addition to this, the triangleArea function *returns* a value. Notice that on line 21 that it doesn't say void between public static and triangleArea. It says

double. That means “this function returns a value, and the type of value it returns is a double.”

If instead it had the keyword void in this position, it means “this function does not return any value.” If we wanted triangleArea to return a different type of value:

```
public static int     triangleArea( int a, int b, int c ) // this would return an int

public static String  triangleArea( int a, int b, int c ) // this would return a String

public static boolean triangleArea( int a, int b, int c ) // this would return either true or false

public static void    triangleArea( int a, int b, int c ) // this cannot return any value of any type
```

Sometimes my students get confused about functions that return values versus functions that do not return values. An analogy is helpful.

> Let us say that we are sitting in my school classroom. We hear the sound of thunder and I remember that I left my car windows down. I don’t want rain to make the inside of my car wet, so I send you out into the parking lot.
>
> “Student, please go out into the parking lot and roll up the windows of my car.”
>
> “Yes, sir,” you say.
>
> If you need information from me about what my car looks like, then those are parameters. If you already know which one is mine, you need no parameters.
>
> Eventually you return and say “I completed the task.” This sort of function does not return a value.

```
rollUpWindows(); // if you don't need parameters

rollUpWindows("Toyota", "Corolla", 2008, "blue"); // if you do need parameters
```

In either case, the function is executed and goes off and does its thing, but returns no value. Now, example #2:

> Again we are in my classroom. I am online trying to update my car insurance and the web page is asking me for my car’s license plate number. I don’t remember it, so I ask you to go to the parking lot and get it for me.
>
> Eventually you return and *tell me the license plate number*. Maybe you wrote it down on a scrap of paper or maybe you memorized it. When you give it to me, I copy it down myself. This sort of function returns a value.

```
String plate;

plate = retrieveLicensePlate(); // if you don't need parameters
```

```
plate = retrieveLicensePlate("Toyota", "Corolla", 2008, "blue"); // if you do need them
```

If I am rude, you could return to my classroom and give me the value and I could put my fingers in my ears so I don't hear you or refuse to write it down myself so that I quickly forget it. If you call a function that returns a value, you can choose to *not* store the return value into a variable and just allow the value to vanish:

```
retrieveLicensePlate("Toyota", "Corolla", 2008, "blue"); // returns a value which is lost

triangleArea(3, 3, 3); // returns the area but we refuse to store it into a variable
```

This is usually a bad idea, but maybe you have your reasons.

In any case, on line 10 we call the triangleArea function. We pass in 3, 4 and 5 as the three arguments. The 3 gets stored as the value of *a* (down on line 21). The 4 is stored into *b*, and 5 is put into *c*. It runs all the code on lines 23 through 28 with those values for the parameters. By the end, the variable *A* has a value stored in it.

On line 29 we *return* the value that is in the variable *A*.[11] This value travels back up to line 10, where it is stored into the variable *a*.

And just to make sure you can see why functions are worth the trouble, here is an example of writing this same program without using a function.

```
1 public class HeronsFormulaNoFunction
 2 {
 3    public static void main( String[] args )
 4    {
 5       int a, b, c;
 6       double s, A;
 7
 8       a = 3;
 9       b = 3;
10       c = 3;
11       s = (a+b+c) / 2;
12       A = Math.sqrt( s*(s-a)*(s-b)*(s-c) );
13       System.out.println("A triangle with sides 3,3,3 has an area of " + A );
14
 15       a = 3;
16       b = 4;
17       c = 5;
18       s = (a+b+c) / 2;
19       A = Math.sqrt( s*(s-a)*(s-b)*(s-c) );
20       System.out.println("A triangle with sides 3,4,5 has an area of " + A );
21
 22       a = 7;
23       b = 8;
24       c = 9;
25       s = (a+b+c) / 2;
```

[11][11](The variable *A* itself does **not** get returned, only its value. In fact, remember that the "scope" of a variable is limited to the block of code inside which it is defined? (You learned that in Exercise 21.) The variable *a* is only in scope inside the function main, and the variables *s*, *A*, and the parameter variables *a*, *b* and *c* are only in scope inside the function triangleArea.)

```
26        A = Math.sqrt( s*(s-a)*(s-b)*(s-c) );
27        System.out.println("A triangle with sides 7,8,9 has an area of " + A );
28
 29        a = 5;
30        b = 12;
31        c = 13;
32        s = (a+b+c) / 2;
33        A = Math.sqrt( s*(s-a)*(s-b)*(s-c) );
34        System.out.println("A triangle with sides 5,12,13 has an area of " + A );
35
 36        a = 10;
37        b = 9;
38        c = 11;
39        s = (a+b+c) / 2;
40        A = Math.sqrt( s*(s-a)*(s-b)*(s-c) );
41        System.out.println("A triangle with sides 10,9,11 has an area of " + A );
42
 43        a = 8;
44        b = 15;
45        c = 17;
46        s = (a+b+c) / 2;
47        A = Math.sqrt( s*(s-a)*(s-b)*(s-c) );
48        System.out.println("A triangle with sides 8,15,17 has an area of " + A );
49    }
50 }
```

What You Should See

```
A triangle with sides 3,3,3 has an area of 2.0
A triangle with sides 3,4,5 has an area of 6.0
A triangle with sides 7,8,9 has an area of 26.832815729997478
A triangle with sides 5,12,13 has an area of 30.0
A triangle with sides 10,9,11 has an area of 42.42640687119285
A triangle with sides 8,15,17 has an area of 60.0
```

Study Drills

1. Which one is longer, the one with the function or the one without?
56. There is a bug in the formula for both files. When (a+b+c) is an odd number, dividing by 2 throws away the .5. Fix it so that instead of (a+b+c)/2 you have (a+b+c)/2.0. How much harder would it have been to fix the version that didn't use a function?
57. Add one more test: find the area of a triangle with sides 9, 9, and 9. Was it difficult to add? How much harder would it have been to add the test to the version that didn't use a function?

What You Should See After Doing the Study Drills

```
A triangle with sides 3,3,3 has an area of 3.897114317029974
A triangle with sides 3,4,5 has an area of 6.0
A triangle with sides 7,8,9 has an area of 26.832815729997478
A triangle with sides 5,12,13 has an area of 30.0
```

```
A triangle with sides 10,9,11 has an area of 42.42640687119285
A triangle with sides 8,15,17 has an area of 60.0
A triangle with sides 9,9,9 has an area of 35.074028853269766
```

That’s better.

Exercise 38: Areas of Shapes

Today's exercise has nothing new. It is merely additional practice with functions. This program has three functions (four if you count main) and they all have parameters and all three return values.

```
1 import java.util.Scanner;
 2
 3 public class ShapeArea
 4 {
 5     public static void main( String[] args )
 6     {
 7         Scanner keyboard = new Scanner(System.in);
 8
 9         int choice;
10          double area = 0;
11
 12          System.out.println("Shape Area Calculator version 0.1 (c) 2013 Mitchell Sample Output,
Inc.");
13
 14          do
15          {
16             System.out.println("\n-=-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
17             System.out.println("1) Triangle");
18             System.out.println("2) Circle");
19             System.out.println("3) Rectangle");
20             System.out.println("4) Quit");
21             System.out.print("> ");
22             choice = keyboard.nextInt();
23
 24              if ( choice == 1 )
25             {
26                System.out.print("\nBase: ");
27                int b = keyboard.nextInt();
28                System.out.print("Height: ");
29                int h = keyboard.nextInt();
30                area = computeTriangleArea(b, h);
31                System.out.println("The area is " + area);
32             }
33             else if ( choice == 2 )
34             {
35                System.out.print("\nRadius: ");
36                int r = keyboard.nextInt();
37                area = computeCircleArea(r);
38                System.out.println("The area is " + area);
39             }
40             else if ( choice == 3 )
41             {
42                System.out.print("\nLength: ");
43                int length = keyboard.nextInt();
44                System.out.print("Width: ");
45                int width = keyboard.nextInt();
46                System.out.println("The area is " + computeRectangleArea(length, width) );
47             }
48             else if ( choice != 4 )
49             {
50                System.out.println("ERROR.");
51             }
```

```
52
 53        } while ( choice != 4 );
54
 55     }
56
 57     public static double computeTriangleArea( int base, int height )
58     {
59        double A;
60        A = 0.5 * base * height;
61        return A;
62     }
63
 64     public static double computeCircleArea( int radius )
65     {
66        double A;
67        A = Math.PI * radius * radius;
68        return A;
69     }
70
 71     public static int computeRectangleArea( int length, int width )
72     {
73        return (length * width);
74     }
75 }
```

What You Should See

```
Shape Area Calculator version 0.1 (c) 2013 Mitchell Sample Output, Inc.

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-

1) Triangle
2) Circle
3) Rectangle
4) Quit
> 1

Base: 3
Height: 5
The area is 7.5

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-

1) Triangle
2) Circle
3) Rectangle
4) Quit
> 2

Radius: 3
The area is 28.274333882308138

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-

1) Triangle
2) Circle
3) Rectangle
```

```
4) Quit
> 4
```

On line 57 we have defined a function to compute the area of a triangle (using just the base and height this time). It needs two arguments and will return a double value. On line 59 we declare a variable named *A*. This variable is "local" to the function. Even though there is a variable named *A* declared on line 66, they are not the same variable. (It's like having two friends named "Michael". Just because they have the same name doesn't make them the same person.)

The value of the variable *b* (defined on line 27) is passed in as the initial value of the parameter *base* in the function call on line 30. *b* is stored into *base* because *b* is first, not because *base* starts with a *b*. The computer doesn't care anything about that. Only the order matters.

On line 61 the value of *A* is returned to main and ends up getting stored in the variable called *area*.

I did three strange things in the rectangle area function whose definition begins on line 71.

First, the formal parameters have the same names as the actual arguments. (Remember, the parameters are the variables declared in the function definition on line 71 and the arguments are the variables in the parentheses in the function call on line 46.) This is a neat coincidence, but it doesn't mean anything. It is like having an actor named "Steven" playing a character named "Steven". The value from main's version of *length* gets stored into computeRectangleArea's *length* variable because they are both listed first in the parentheses and for no other reason.

Secondly, I did not bother to create a variable for the value the function is going to return on line 73. I simply returned the value of the expression length*width. The function will figure out what the value is and return it right away without ever storing it into a variable.

Thirdly, the rectangle area value is returned back to main on line 46, but I didn't bother to store the return value into a variable: I just printed it on the screen directly. (I also did this in HeronsFormula but I didn't call attention to it.) This is totally fine and actually pretty common. We call functions all the time and we almost always *use* the return value of the function but we don't always need to store the return value into its own variable.

Finally, before we move on to another topic I should mention that in Java, functions can only return a single value. In some other programming languages functions can return more than one value. But in Java functions can return a single value or no value (if the function is void) but never more than one.

P.S. These functions are a bit silly. If I were *really* needing a shape area calculator, I am not sure if it would be worth it to create a whole function for an equation that is only one line of code. But this example is good for explaining, anyway.

Study Drills

1. Add a function to compute the area of a square. Add it to the menu as well.

Exercise 39: Thirty Days Revisited with Javadoc

In the previous exercise we wrote some functions that might have been better off omitted. In today's exercise we are going to re-do a previous exercise, making it better with functions.

And, because I always have to keep pushing, I have added special comments above the class and above each function called "Javadoc comments".

```
1 import java.util.Scanner;
 2
 3 /**
 4  * Contains functions that make it easier to work with months.
 5  */
 6 public class ThirtyDaysFunctions
 7 {
 8     public static void main( String[] args )
 9     {
10         Scanner kb = new Scanner(System.in);
11
 12         System.out.print( "Which month? (1-12) " );
13         int month = kb.nextInt();
14
 15         System.out.println( monthDays(month) + " days hath " + monthName(month) );
16
 17     }
18
 19     /**
20      * Returns the name for the given month number (1-12).
21      *
22      * @author Graham Mitchell
23      * @param  month  the month number (1-12)
24      * @return        a String containing the English name for the given month, or "error" if
month out of range
25      */
26     public static String monthName( int month )
27     {
28         String monthName = "error";
29
 30         if ( month == 1 )
31             monthName = "January";
32         else if ( month == 2 )
33             monthName = "February";
34         else if ( month == 3 )
35             monthName = "March";
36         else if ( month == 4 )
37             monthName = "April";
38         else if ( month == 5 )
39             monthName = "May";
40         else if ( month == 6 )
41             monthName = "June";
42         else if ( month == 7 )
43             monthName = "July";
44         else if ( month == 8 )
45             monthName = "August";
46         else if ( month == 9 )
47             monthName = "September";
48         else if ( month == 10 )
```

```
49          monthName = "October";
50       else if ( month == 11 )
51          monthName = "November";
52       else if ( month == 12 )
53          monthName = "December";
54
 55       return monthName;
56    }
57
 58    /**
59     * Returns the number of days in a given month.
60     *
61     * @author Graham Mitchell
62     * @param  month  the month number (1-12)
63     * @return        the number of days in a non-leap year for that month, or
```

```
31 if month out of range
64     */
65    public static int monthDays( int month )
66    {
67       int days;
68
 69       /* Thirty days hath September
70          April, June and November
71          All the rest have thirty-one
72          Except the second month alone.... */
73
 74       switch(month)
75       {
76          case  9:
77          case  4:
78          case  6:
79          case 11: days = 30;
80                   break;
81          case  2: days = 28;
82                   break;
83          default: days = 31;
84       }
85
 86       return days;
87    }
88 }
```

What You Should See

```
Which month? (1-12) 9
30 days hath September
```

If you ignore the Javadoc comments for now, hopefully you should see that using functions here actually improves the code. main() is very short, because most of the interesting work is being done in the functions.

All the code and variables pertaining to the name of the month is isolated in the monthName() function. And all the code to find the number of days in a month is

contained inside the monthDays() function.

Collecting variables and code into functions like this is called "procedural programming" and it is considered a major advance over just having all your code in main(). It makes your code easier to debug because if you have a problem with the name of the month, you *know* it has to be inside the monthName() function.

Okay, now let's talk about the Javadoc comments.

javadoc is an automatic documentation-generating tool that is included with the Java compiler. You write document right in your code by doing a special sort of block comment above classes, functions or variables.

The comment begins with /** and ends with */ and every line in between starts with an asterisk (*) which is lined up like you see in the exercise.

The first line of the javadoc comment is a one-sentence summary of the thing (class or function). And then there are tags like @author or @return that give more detail about who wrote the code, what parameters the function expects or what value it is going to return.

Okay, so now for the magic part. Go to the terminal window just like you were going to compile your code, and type the following command:

```
$ javadoc ThirtyDaysFunctions.java
Loading source file ThirtyDaysFunctions.java...
Constructing Javadoc information...
Standard Doclet version 1.7.0_21
Building tree for all the packages and classes...
Generating /ThirtyDaysFunctions.html...
Generating /package-frame.html...
Generating /package-summary.html...
Generating /package-tree.html...
Generating /constant-values.html...
Building index for all the packages and classes...
Generating /overview-tree.html...
Generating /index-all.html...
Generating /deprecated-list.html...
Building index for all classes...
Generating /allclasses-frame.html...
Generating /allclasses-noframe.html...
Generating /index.html...
Generating /help-doc.html...
$
```

Then if you look in the folder where ThirtyDaysFunctions.java is located, you will see a *lot* of new files. (Maybe I should have warned you first.) Open the file called index.html in the web browser of your choice.

This is javadoc documentation, and there is a *lot* of information there. You can find the comment you put for the class near the top, and the comments for the functions are in the section called "Method Summary".

The details about the parameters and return types are down below in the section called "Method Detail".

Study Drills

1. Look at the javadoc documentation for one of the built-in Java classes: java.util.Scanner. Notice how similar it looks to what the javadoc tool generated? All the official Java documentation is created using the javadoc tool, so learning how to read it will be an important part of becoming an expert Java programmer. Don't worry too much about the details right now, though. Just try to get a feel for how it looks.
   In the upper left is a list of all the packages of code that are included as part of Java and below that on the left is a list of all the classes/ libraries you could import to save you from having to write code. A big part of what professional Java programmers do is write code to glue together existing Java libraries.
   This is probably overwhelming right now. That's fine because you have just started. Hopefully no one expects you to understand much about this yet. In fact, most programmers only know about a fraction of the built-in Java libraries, and they search on the Internet and read the documentation when they need to do something new, just like you do!

Exercise 40: Importing Standard Libraries

In the last exercise you got a terrifying look at all of the built-in modules that are available in Java. Today we will look at a "simple" program that took me about half an hour to write because I spent a lot of time searching the Internet and importing things and trying things that didn't work.

This code works, though. It allows the human to enter a password (or anything, really) and then prints out the SHA-256 message digest of that password.

When you are typing in this code, don't forget to put the throws Exception at the end of line 7.

```
1 import java.util.Scanner;
 2 import java.security.MessageDigest;
 3 import javax.xml.bind.DatatypeConverter;
 4
 5 public class PasswordDigest
 6 {
 7     public static void main( String[] args ) throws Exception
 8     {
 9         Scanner keyboard = new Scanner(System.in);
10
 11         String pw, hash;
12
 13         MessageDigest digest = MessageDigest.getInstance("SHA-256");
14
 15         System.out.print("Password: ");
16         pw = keyboard.nextLine();
17
 18         digest.update( pw.getBytes("UTF-8") );
19         hash = DatatypeConverter.printHexBinary( digest.digest() );
20
 21         System.out.println( hash );
22     }
23 }
```

What You Should See

```
Password: password
5E884898DA28047151D0E56F8DC6292773603D0D6AABBDD62A11EF721D1542D8
```

That 64-character long string is the SHA-256 digest of the String password. That message digest will always be the same for that input.

If you type in a different password, you'll get a different digest, of course:

```
Password: This is a really long password and no one will ever guess it.
A113B65D8BA8DB72D631D97B7A3698E82CDB9D1F52456C8957312CB91EC02B10
```

Back in the early days of programming, when machines first started having usernames and password, it was pretty obvious that you wouldn't want to store the passwords themselves in a database. Instead, they would store some sort

of cryptographic hash of the password.

Cryptographic hashes have two useful properties:

1. They are consistent. A given input will always produce exactly the same output.
58. They are one-way. You can easily compute the output for a given input, but figuring out the input that gave you a certain output is very hard or impossible.

SHA-256 is a very good cryptographic hash function, and it produces a "digest" for a given input (or "message") that is always exactly 256 bits long. Here instead of trying to deal with bits we have printed out the base-64 representation of those bits, which ends up being 64 characters long where each character is a hexadecimal digit.

Back in the 1970s, to change your password on a certain machine you would type your password and the machine would store your username and the hash of your new password in a file.

Then when you wanted to log in to the machine later, it would make you type in your username and password. It would find the username in the password database file and find the stored hash of your password. Then it would find the hash of whatever password you just typed. If the stored hash and the computed hash match, then you must have typed in the correct password and you would be allowed access to the machine.

This is a clever scheme. It is also *much* better than ever storing passwords directly in a database. However, nowadays computers are way too fast and have way too much storage space for this to be enough security. Since machines can compute the SHA-256 of a password *very* quickly, it doesn't take long for a determined hacker to figure out what your password was.

> (If you really want to securely store passwords in a database, you should be using bcrypt, which is made for such things. Unfortunately bcrypt isn't built-in to Java, so you will need to download a bcrypt library made by someone else.)

Okay, enough about secure passwords, let us walk through this code. You might want to have the javadoc documentation for these two libraries open.

- java.security.MessageDigest
- javax.xml.bind.DatatypeConverter

On lines 2 and 3 we import the two libraries we will be using to do the hard parts of this exercise.

On line 13 we create a variable of type MessageDigest (which now exists because we imported java.security.MessageDigest. Our variable is named *digest*, although I could have called it something else. And the value of the variable comes from the return value of the method MessageDigest.getInstance(). We pass in a String as an argument to this method, which is which digest we want. In this case we are using "SHA-256", but "SHA-1" and "MD5" would have also worked. You can read about this stuff in the javadoc documentation.

Lines 15 and 16 are hopefully boring. Notice that I used nextLine() instead of just next() to read in the password, which allows the human to type in more

than one word.

On line 18 we call the getBytes() method of the String class, with an argument of "UTF-8". This converts the String value to a raw list of bytes in UTF-8 format which we pass directly as an argument to the update() method of the MessageDigest object named *digest*. I learned about the getBytes() method by reading the javadoc documentation for the String class!

- java.lang.String

On line 19 we call the digest() method of the MessageDigest object named *digest*. This gives us a raw list of bytes and isn't suitable for printing on the screen, so we pass that raw list of bytes directly as a parameter to the printHexBinary() method of the DatatypeConverter class. This returns a String, which we store into the variable *hash*.

We then display the hash value on the screen. Whew!

If this exercise freaked you out a little bit, don't worry. If you can make it through the first 39 exercises in the book, then you could learn to do this sort of thing, too. You have to learn how to read javadoc documentation to learn what sort of tools other people have already written for you and how to connect them together to get what you want. It just takes a lot of practice! Remember that writing this exercise the first time took me more than half an hour, and I've been programming since the 1980s and started coding in Java in 1996!

Study Drills

1. Look at the javadoc documentation for all the methods used in this exercise: getInstance, getBytes, update, digest, and printHexBinary. Look at what arguments they expect and look at the types of values they will return.
59. Remove the throws Exception from the end of line 7. Try to compile it. (Then put it back.) You will learn a tiny bit about exceptions in the next exercise.

Exercise 41: Programs that Write to Files

We are going to take a break from focusing on functions now for a bit and learn something easy. We are going to create a program that can put information into a text file instead of only being able to print things on the screen.

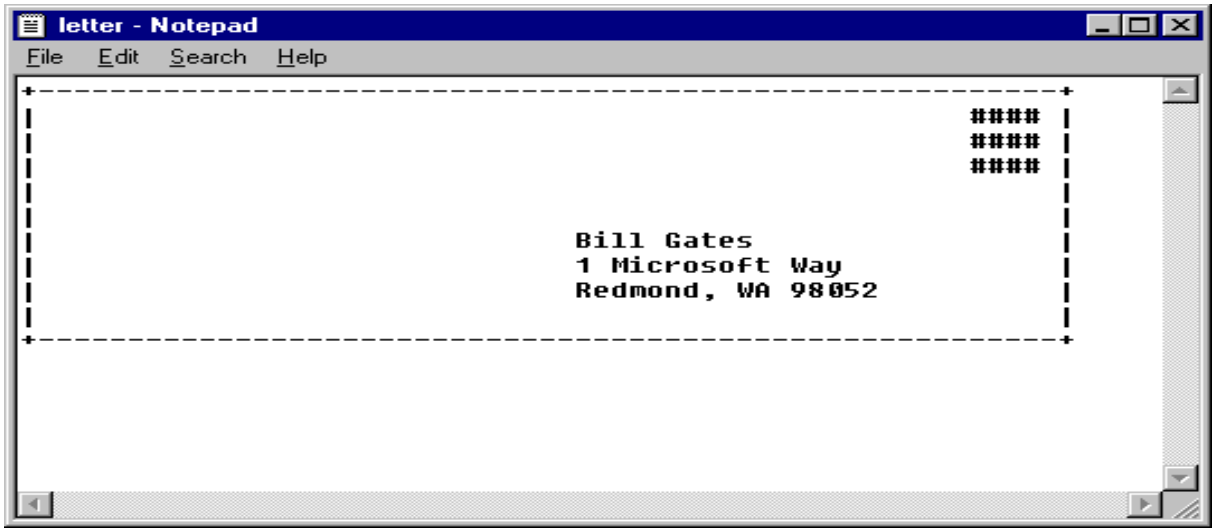
When you are typing in this code, don't miss the throws Exception at the end of line 6. (And in this exercise, I'll actually explain what that means.)

```
1 import java.io.FileWriter;
 2 import java.io.PrintWriter;
 3
 4 public class LetterRevisited
 5 {
 6     public static void main( String[] args ) throws Exception
 7     {
 8         PrintWriter fileout;
 9
 10         fileout = new PrintWriter( new FileWriter("letter.txt") );
11
 12         fileout.println( "+------------------------------------------------------+" );
13         fileout.println( "|                                      #### |" );
14         fileout.println( "|                                      #### |" );
15         fileout.println( "|                                      #### |" );
16         fileout.println( "|                                         |" );
17         fileout.println( "|                                         |" );
18         fileout.println( "|                      Bill Gates         |" );
19         fileout.println( "|                      1 Microsoft Way       |" );
20         fileout.println( "|                      Redmond, WA 98104        |" );
21         fileout.println( "|                                         |" );
22         fileout.println( "+------------------------------------------------------+" );
23         fileout.close();
24     }
25 }
```

What You Should See

That's right. When you run your program, it will appear to do nothing. But if you wrote it correctly, it should have created a file called letter.txt in the same folder your code is in. You can view this file using the same text editor you are using to write your code.

If for some reason you are using the version of Notepad that came with Windows 95, it will look a little something like this:



(I made that

screenshot a long time ago, okay? I've been doing this a long time, remember. When I first gave this assignment to students, Windows 95 was the newest version of Windows…. In fact, I guess the zip code changed at some point.)

On lines 1 and 2 there are two new import statements, one for each of the Java classes that will make this easy.

On line 8 we declare a variable. The variable is of type PrintWriter and I have chosen to name it *fileout* (although the variable's name doesn't matter).

On line 10 we give the PrintWriter variable a value: the reference to a new PrintWriter object. Creating the PrintWriter object requires an argument, though. The argument we give it is a new FileWriter object, which itself is created with the filename as an argument.

It is possible to write to a text file using *only* a FileWriter object and without using any PrintWriter at all. However, PrintWriters are much easier to work with, as you can tell by looking at the rest of the code. So instead of working with the FileWriter object directly, we "wrap" the FileWriter object with a PrintWriter object and just work through the PrintWriter object.

> (It is okay if you didn't understand the last two paragraphs. You don't need to understand them to write to files.)

The good news is that once the PrintWriter object is set up, everything else is easy. Because you have secretly been working with PrintWriters since the very beginning! This is because System.out is a PrintWriter!

So on line 12 you can see that writing to the file looks very similar to printing on the screen. But the String (+-----) will *not* be printed on the screen. It will be stored as the first line of the file letter.txt!

If a file named letter.txt already exists in that folder, its contents will be overwritten without warning. If the file does not exist, it will be created.

The only other important line in the exercise is line 23. This actually saves the contents of the file and closes it so your program can't write to it anymore. If you remove this line, your program will most likely create a file called letter.txt, but the file will be empty.

Okay, before I end the exercise, I want to briefly discuss throws Exception. This is not something we do much in Real Programming, and explaining it properly is beyond the scope of this book, but I do want to touch on it.

In the original version of the exercise, when you put throws Exception after the first line of a function, it means "I have written code in this function that might not work, and if it fails it will blow up (by throwing an exception)."

In this case the thing that might not work is the line new FileWriter("letter.txt") because it tries to open a file for writing in the current folder. This could fail if there is already a file called "letter.txt" and the file is read-only. Or maybe the whole folder is read-only. Or there's some other reason the program can't get write permission to the file.

So instead of just blowing up the program we are supposed to detect the exception and handle it. Like so:

```
1 import java.io.FileWriter;
 2 import java.io.PrintWriter;
 3 import java.io.IOException;
 4
 5 public class LetterRevisitedException
 6 {
 7    public static void main( String[] args )
 8    {
 9       PrintWriter fileout;
10
 11       try
12       {
13          fileout = new PrintWriter( new FileWriter("letter.txt") );
14       }
15       catch ( IOException err )
16       {
17          System.out.println("Sorry, I can't write to the file 'letter.txt'.");
18          System.out.println("Maybe the file exists and is read-only?");
19          fileout = null;
20          System.exit(1);
21       }
22
 23       fileout.println( "+-------------------------------------------------------+" );
24       fileout.println( "|                                        #### |" );
25       fileout.println( "|                                        #### |" );
26       fileout.println( "|                                        #### |" );
27       fileout.println( "|                                           |" );
28       fileout.println( "|                                           |" );
29       fileout.println( "|                       Bill Gates          |" );
30       fileout.println( "|                       1 Microsoft Way     |" );
31       fileout.println( "|                       Redmond, WA 98104   |" );
32       fileout.println( "|                                           |" );
33       fileout.println( "+-------------------------------------------------------+" );
34       fileout.close();
35    }
36 }
```

The try block means “this code may throw an exception, but attempt it.” If everything goes well (if there is no exception thrown) then the catch block is skipped. If there is an exception thrown, the catch block gets executed, and the exception that was thrown gets passed in as a parameter. (I have named the exception parameter *err*, though it could be named anything.)

Inside the catch block I print out a suitable error message and then end the program by calling the built-in function System.exit(). If you pass an argument of 0 to System.exit(), the program will end, but the zero means “everything is fine”. An argument of 1 means “this program is ending, and it is because something went wrong.”

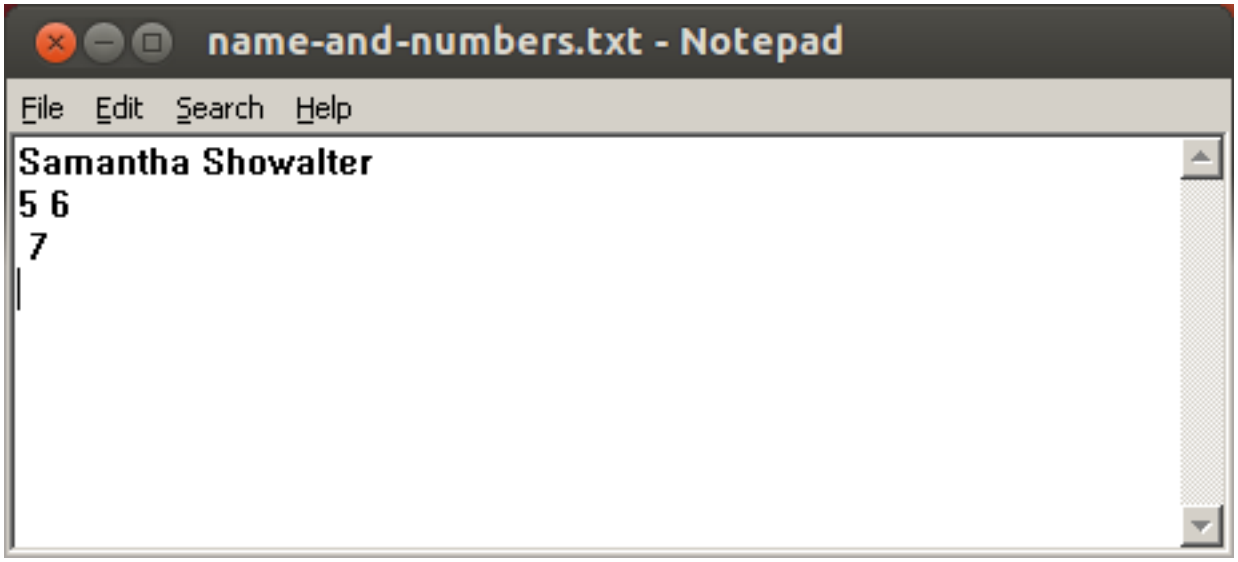
So I won’t use try and catch anymore in this book, but at least now you know what you are avoiding by putting throws Exception.

Exercise 42: Getting Data from a File

A program that can put information into a file is only part of the story. So in this exercise you will learn how to read information that is already in a text file.

If you type up this code and compile it and run, it will blow up. This is because it is trying to read from a text file called name-and-numbers.txt, which must be in the same folder as your code. You probably don't have a file like this!

So before you even write the code, let us make a text file containing a String and three integers. My file looks like this:



(This is a slightly newer version of Notepad. Happy now?)

Okay, to the code!

```
1 import java.util.Scanner;
 2 import java.io.File;
 3
 4 public class GettingFromFile
 5 {
 6     public static void main( String[] args ) throws Exception
 7     {
 8         Scanner fileIn = new Scanner(new File("name-and-numbers.txt"));
 9
 10         int a, b, c, sum;
11         String name;
12
 13         System.out.print("Getting name and three numbers from file...");
14         name = fileIn.nextLine();
15         a = fileIn.nextInt();
16         b = fileIn.nextInt();
17         c = fileIn.nextInt();
18         fileIn.close();
19
 20         System.out.println("done.");
21         System.out.println("Your name is " + name);
22         sum = a + b + c;
23         System.out.println( a + "+" + b + "+" + c + " = " + sum );
24     }
25 }
```

What You Should See

```
Getting name and three numbers from file...done.
Your name is Samantha Showalter
5+6+7 = 18
```

Did you know that the Scanner object doesn't have to get input from the human at the keyboard? It can read data from text files, too!

We just create the Scanner object slightly differently: instead of System.in as the argument, we use new File("blah.txt"). This will open the text file read-only. The Scanner object (which I have chosen to call *fileIn*) will be attached to the file like a straw stuck into a juice box. (The juice box is the text file, and the Scanner object is the straw.)

Line 14 looks pretty uninteresting. It "pauses" the program and reads in a String from the Scanner object, which gets it from the file. This String from the file is stored into the variable.

Lines 15 through 17 are simple, too. Except what is read from the file is converted to an integer before putting it in the variables.

What if the next thing in the file isn't an integer? Then your program will blow up. And now you can't blame the human anymore: you created this file. It is your job to make sure you know what values are in it, and in what order.

On line 18 the file is closed, which means your Scanner object isn't connected to it anymore.

Was this easier than you expected it to be? Hopefully so.

Study Drills

1. Open the text file and change the name or numbers. Save it. Then run your program again (you don't have to compile it again; the code hasn't changed and it doesn't open the file until it is run.)

Exercise 43: Saving a High Score

Now that you know how to get information from files *and* how to put information in files, we can create a game that saves the high score!

This is the coin flipping game from a few exercises ago, but now the high score is saved from run to run.

```
1 import java.util.Scanner;
 2 import java.io.File;
 3 import java.io.FileWriter;
 4 import java.io.PrintWriter;
 5
 6 public class CoinFlipSaved
 7 {
 8     public static void main( String[] args ) throws Exception
 9     {
10         Scanner keyboard = new Scanner(System.in);
11
 12         String coin, again, bestName, saveFile = "coin-flip-score.txt";
13         int flip, streak = 0, best;
14
 15         File in = new File(saveFile);
16         if ( in.createNewFile() )
17         {
18            System.out.println("Save game file doesn't exist. Created.");
19            best = -1;
20            bestName = "";
21         }
22         else
23         {
24            Scanner input = new Scanner(in);
25            bestName = input.next();
26            best = input.nextInt();
27            input.close();
28            System.out.println("High score is " + best + " flips in a row by " + bestName );
29         }
30
 31
 32         do
33         {
34            flip = 1 + (int)(Math.random()*2);
35
 36            if ( flip == 1 )
37               coin = "HEADS";
38            else
39               coin = "TAILS";
40
 41            System.out.println( "You flip a coin and it is... " + coin );
42
 43            if ( flip == 1 )
44            {
45               streak++;
46               System.out.println( "\tThat's " + streak + " in a row...." );
47               System.out.print( "\tWould you like to flip again (y/n)? " );
48               again = keyboard.next();
49            }
50            else
```

```
51          {
52             streak = 0;
53             again = "n";
54          }
55       } while ( again.equals("y") );
56
 57       System.out.println( "Final score: " + streak );
58
 59       if ( streak > best )
60       {
61          System.out.println("That's a new high score!");
62          System.out.print("Your name: ");
63          bestName = keyboard.next();
64          best = streak;
65       }
66       else if ( streak == best )
67       {
68          System.out.println("That ties the high score. Cool.");
69       }
70       else
71       {
72          System.out.println("You'll have to do better than " + streak + " if you want a high
score.");
73       }
74
 75       // Save this name and high score to the file.
76       PrintWriter out = new PrintWriter( new FileWriter(saveFile) );
77       out.println(bestName);
78       out.println(best);
79       out.close();
80    }
81 }
```

What You Should See

```
Save game file doesn't exist. Created.
You flip a coin and it is... HEADS
      That's 1 in a row....
      Would you like to flip again (y/n)? y
You flip a coin and it is... HEADS
      That's 2 in a row....
      Would you like to flip again (y/n)? y
You flip a coin and it is... HEADS
      That's 3 in a row....
      Would you like to flip again (y/n)? n
Final score: 3
That's a new high score!
Your name: Mitchell
```

(Okay, so I cheated. It took me quite a few tries to get a streak of three in a row.)

On line 15 we create a File object using the filename coin-flip-score.txt. We can do this even if the file doesn't exist.

On line 16 there is an if statement, and in the condition I call the

createNewFile() method of the File object. This will check to see if the file exists. If so, it will do nothing and return the Boolean value false. If the file does not exist, it will create the file empty and return the value true.

When the if statement is true, then, it means the save game file didn't exist. We say so and put suitable initial values into the variables *best* and *bestName*. If not, then there's already a file there, so we use a Scanner object to get the existing name and high score out of the file. Cool, eh?

Lines 32 through 57 are the existing coin flip game. I didn't change any of this code at all.

On line 59 we need to figure out if they beat the high score. If so, we print out a message to that effect and let them enter their name.

If they tied the high score, we say so, but they don't get any fame for that.

And on line 70 the else will run if they didn't beat or tie the high score. So we taunt them, of course.

On lines 75 through 79 we save the current high score along with the name of the high scorer to the file. This might be a new score, or it might be the previous value we read at the beginning of the program.

Study Drills

1. Change the program so that it only saves to the high score file if it has changed.
60. "Hack" the high score file by opening it in a text editor and manually changing it. Impress your friends with your amazing lucky streak!

Exercise 44: Counting with a For Loop

As you have seen in previous exercises, while loops and do-while loops can be used to to make something happen more than once.

But both kinds of loops are designed to keep going *as long as* something is true. If we know in advance how many times we want to do something, Java has a special kind of loop designed just for making a variable change values: the for loop.

```
1 import java.util.Scanner;
 2
 3 public class CountingFor
 4 {
 5     public static void main( String[] args )
 6     {
 7         Scanner keyboard = new Scanner(System.in);
 8
 9         int n;
10          String message;
11
 12          System.out.println( "Type in a message, and I'll display it five times." );
13          System.out.print( "Message: " );
14          message = keyboard.nextLine();
15
 16          for ( n = 1 ; n <= 5 ; n++ )
17          {
18              System.out.println( n + ". " + message );
19          }
20
 21          System.out.println( "\nNow I'll display it ten times and count by 5s." );
22          for ( n = 5 ; n <= 50 ; n += 5 )
23          {
24              System.out.println( n + ". " + message );
25          }
26
 27          System.out.println( "\nFinally, three times counting backward." );
28          for ( n = 3 ; n > 0 ; n -= 1 )
29          {
30              System.out.println( n + ". " + message );
31          }
32
 33      }
34 }
```

What You Should See

```
Type in a message, and I'll display it five times.
Message: Howdy, y'all!
1. Howdy, y'all!
2. Howdy, y'all!
3. Howdy, y'all!
4. Howdy, y'all!
5. Howdy, y'all!
```

```
Now I'll display it ten times and count by 5s.
5. Howdy, y'all!
10. Howdy, y'all!
15. Howdy, y'all!
20. Howdy, y'all!
25. Howdy, y'all!
30. Howdy, y'all!
35. Howdy, y'all!
40. Howdy, y'all!
45. Howdy, y'all!
50. Howdy, y'all!

Finally, three times counting backward.
3. Howdy, y'all!
2. Howdy, y'all!
1. Howdy, y'all!
```

Line 16 demonstrates a very basic for loop. Every for loop has three parts with semicolons between.

The first part (n=1) only happens once no matter how many times the loop repeats. It happens at the very beginning of the loop and usually sets a starting value for some variable that is going to be used to control the loop. In this case, our "loop control variable" is *n* and it will start with a value of 1.

The second part (n <= 5) is a condition, just like the condition of a while or do-while loop. The for loop is a pre-test loop just like a while loop, which means that this condition is tested before the loop starts looping. If the condition is true, the loop body will be executed one time. If the condition is false, the loop body will be skipped and the loop is over.

The third part (n++) runs *after* each iteration of the loop, just before it checks the condition again. Remember that ++ adds one to a variable.

So if we *unroll* this loop, these are the statements that will happen and their order:

```
n = 1;
// check if ( n <= 5 ), which is true
System.out.println( 1 + "." + message );
n++; // so now n is 2
// check if ( n <= 5 ), which is true
System.out.println( 2 + "." + message );
n++; // so now n is 3
// check if ( n <= 5 ), which is true
System.out.println( 3 + "." + message );
n++; // so now n is 4
// check if ( n <= 5 ), which is true
System.out.println( 4 + "." + message );
n++; // so now n is 5
// check if ( n <= 5 ), which is true
System.out.println( 5 + "." + message );
n++; // so now n is 6
// check if ( n <= 6 ), which is false. The loop stops
```

Notice that the first part only happened once, and that the third part happened

exactly as many times as the loop body did.

On line 22 there is another for loop. The loop control variable is still *n*. (Notice that the loop control variable appears in all three parts of the loop. This is almost always the case.)

The first part (the "initialization expression") sets the loop control variable to start at 5. Then the second part checks to see if *n* is less than or equal to 50. If so, the body is executed one time and then the third part is executed. The third part adds 5 to the loop control variable, and then the condition is checked again. If it is still true, the loop repeats. Once it is false, the loop stops.

On line 28 there is one final for loop. This time the loop control variable starts at 3 and the loop repeats as long as *n* is greater than zero. And after each iteration of the loop body the third part (the "update expression") *subtracts* 1 from the loop control variable.

So when should you use a for loop versus a while loop?

for loops are best when we know in advance how many times we want to do something.

- Do this ten times.
- Do this five times.
- Pick a random number, and do it that many times.
- Take this list of items, and do it one time for each item in the list.

On the other hand, while and do-while loops are best for repeating *as long as* something is true:

- Keep going as long as they haven't guessed it.
- Keep going as long as you haven't got doubles.
- Keep going as long as they keep typing in a negative number.
- Keep going as long as they haven't typed in a zero.

Study Drills

1. Delete the first part (the "initialization expression") from the third loop. If you remove it correctly, it will still compile. What happens when you run it?

Exercise 45: Caesar Cipher (Looping Through a String)

The Caesar cipher is a very simple form of cryptography named after Julius Caesar, who used it to protect his private letters. In the cipher, each letter is shifted up or down in the alphabet by a certain amount. For example, if the shift is 2, then all As in the message are replaced with C, B is replaced with D, and so on.

```
1 import java.util.Scanner;
 2
 3 public class CaesarCipher
 4 {
 5    /**
 6     * Returns the character shifted by the given number of letters.
 7     */
 8    public static char shiftLetter( char c, int n )
 9    {
10        int ch = c;
11
 12       if ( ! Character.isLetter(c) )
13           return c;
14
 15       ch = ch + n;
16        if ( Character.isUpperCase(c) && ch > 'Z' || Character.isLowerCase(c) && ch > 'z' )
17           ch -= 26;
18        if ( Character.isUpperCase(c) && ch < 'A' || Character.isLowerCase(c) && ch < 'a' )
19           ch += 26;
20
 21       return (char)ch;
22    }
23
 24    public static void main( String[] args )
25    {
26        Scanner keyboard = new Scanner(System.in);
27        String plaintext, cipher = "";
28        int shift;
29
 30       System.out.print("Message: ");
31        plaintext = keyboard.nextLine();
32        System.out.print("Shift (0-26): ");
33        shift = keyboard.nextInt();
34
 35       for ( int i=0; i<plaintext.length(); i++ )
36        {
37           cipher += shiftLetter( plaintext.charAt(i), shift );
38        }
39        System.out.println( cipher );
40
 41    }
42 }
```

What You Should See

```
Message: This is a test. XyZaBcDeF
Shift (0-26): 2
Vjku ku c vguv. ZaBcDeFgH
```

Did you know that main() doesn't have to be the first function in the class? Well, it doesn't. Functions can appear in any order.

Also in addition to int, double, String and boolean there is a basic variable type I haven't mentioned: char. A char variable can hold characters like Strings do,

but it can only hold *one* character at a time. String literals in the code are enclosed in double quotes like "Axe", while char literals in the code are in single quotes like 'A'.

Starting on line 8 there is a function called shiftLetter(). It has two parameters: *c* is the character to shift and *n* is the number of spaces to shift it. This function returns a char. So shiftLetter('A', 2) would return the character 'C'.

We don't want to try to shift anything that isn't a letter, so on line 10 we use the built-in Character class to tell us.

Then we are going to be doing a little math with the character, so we store the character's Unicode value into an int on line 13 to make this easier. Then on line 15 we add the desired offset to the character.

This would be it, except that we want the offset to "wrap around", so lines 16 through 19 make sure that the final value is still a letter. Then finally on line 21 we take the value of *ch*, cast it to a char, and return it.

In main(), lines 27 through 34 are pretty boring. Before I can explain the for loop, though, I need to explain two String class methods: charAt() and length().

If you have a String you can get a single char out of it using the charAt() method. Like so:

```
String s = "Howdy";
char c = s.charAt(2);
// Now c == 'w'
// s.length() is 5
```

charAt() is zero-based, so s.charAt(0) gets the first character out of a String *s*. And if s.length() tells you how many characters there are in *s*, then s.charAt(s.length()-1) gets the final character.

Now we can understand the for loop on line 36. The initialization expression *declares* and a loop control variable *i* and sets it equal to 0. The condition goes as long as *i* is less than the number of characters in the message. And the update expression will add 1 to *i* each time.

On line 38, a lot of things are happening. We use the charAt method to pull out only the *i*-th character of the message. That character and the shift value are passed as arguments to the shiftLetter() function, which returns the shifted letter. And finally that shifted letter is tacked on to the end of the String *cipher*.

By the time the loop ends, it has gone through each letter of the message one at a time and built up a new message from the shifted versions of the letters.

Maybe that was too much at once. Let me know.

Study Drills

1. Make a new version of this exercise that gets the message from a text file instead and creates an "encrypted" file with the ciphertext instead of just printing it on the screen.

Exercise 46: Nested For Loops

In programming, the term “nested” usually means to put something inside the same thing. “Nested loops” would be two loops with one inside the other one. If you do it right, then means the inner loop will repeat all its iterations every time the outer loop does one more iteration.

```
1 public class NestingLoops
 2 {
 3     public static void main( String[] args )
 4     {
 5         // this is #1 - I'll call it "CN"
 6         for ( char c='A'; c <= 'E'; c++ )
 7         {
 8             for ( int n=1; n <= 3; n++ )
 9             {
10                 System.out.println( c + " " + n );
11             }
12         }
13
 14         System.out.println("\n");
15
 16         // this is #2 - I'll call it "AB"
17         for ( int a=1; a <= 3; a++ )
18         {
19             for ( int b=1; b <= 3; b++ )
20             {
21                 System.out.print( "(" + a + "," + b + ") " );
22             }
23             // * You will add a line of code here.
24         }
25
 26         System.out.println("\n");
27
 28     }
29 }
```

What You Should See

```
A 1
A 2
A 3
B 1
B 2
B 3
C 1
C 2
C 3
D 1
D 2
D 3
E 1
E 2
E 3
```

(1,1) (1,2) (1,3) (2,1) (2,2) (2,3) (3,1) (3,2) (3,3)

Study Drills

1. Look at the first set of nested loops (“CN”). Which variable changes faster? Is it the variable controlled by the outer loop (c) or the variable controlled by the inner loop (n)?

61. Change the order of the loops so that the “c” loop is on the inside and the “n” loop is on the outside. How does the output change?

62. Look at the second set of nested loops (“AB”). Change the print() statement to println(). How does the output change? (Then change it back to print().)

63. Add a System.out.println() statement after the close brace of the inner loop (the “b” loop), but still inside the outer loop. How does the output change?

Exercise 47: Generating and Filtering Values

Nested for loops are sometimes handy because they are very compact and can make some variables change through a lot of different combinations of values.

Many years ago, a student posed the following math problem to me:

> "Farmer Brown wants to spend exactly $100.00 and wants to purchase exactly 100 animals. If sheep cost $10 each, goats cost $3.50 each and chickens are $0.50 apiece, then how many of each animal should he buy?"

After he left, I thought about it for a few seconds and then wrote the following program.

```
1 public class FarmerBrown
 2 {
 3    public static void main( String[] args )
 4    {
 5       for ( int s = 1 ; s <= 100 ; s++ )
 6       {
 7          for ( int g = 1 ; g <= 100 ; g++ )
 8          {
 9             for ( int c = 1 ; c <= 100 ; c++ )
10             {
11                if ( s+g+c == 100 && 10.00*s + 3.50*g + 0.50*c == 100.00 )
12                {
13                   System.out.print( s + " sheep, " );
14                   System.out.print( g + " goats, and " );
15                   System.out.println( c + " chickens." );
16                }
17             }
18          }
19       }
20    }
21 }
```

What You Should See

```
4 sheep, 4 goats, and 92 chickens.
```

This program is neat because it is very short. But an observer sitting inside the innermost loop (just in front of the if statement on line 11 will see one million different combinations of *s*, *g* and *c* flow by. The first combination attempted will be 1 sheep, 1 goat, 1 chicken. That will be plugged into the math equations in the if statement. They won't be true, and nothing will be printed.

Then the next combination will be 1 sheep, 1 goat and 2 chickens. Which will also fail. Then 1 sheep, 1 goat, 3 chickens. And so on up to 1 sheep, 1 goat and 100 chickens when the inner loop runs its last iteration.

Then the g++ on line 7 will execute, the condition on line 7 will check to make sure *g* is still less than or equal to 100 (which it is) and the body of the middle for loop will execute again.

This will cause the initialization expression of the innermost loop to run again, which resets *c* to 1. So the next combination of variable that will be tested in the if statement is 1 sheep, 2 goats and 1 chicken. Then 1 sheep, 2 goats, 2 chickens, then 1 sheep, 2 goats, 3 chickens. Et cetera.

By the end all 100 * 100 * 100 combinations have been tested and 999,999 of them failed. But because computers are very fast, the answer appears instantaneously.

Since curly braces are optional in Java when there is only a single line of code in the body of an if statement or in the body of a for loop, I could have made the code even more compact:

```
1 public class FarmerBrownCompact
 2 {
 3    public static void main( String[] args )
 4    {
 5       for ( int s = 1 ; s <= 100 ; s++ )
 6          for ( int g = 1 ; g <= 100 ; g++ )
 7             for ( int c = 1 ; c <= 100 ; c++ )
 8                if ( s+g+c == 100 && 10.00*s + 3.50*g + 0.50*c == 100.00 )
 9                   System.out.println( s + " sheep, " + g + " goats, and " + c + " chickens." );
10    }
11 }
```

This is perfectly legal and behaves identically to the previous version. Compare that to how much code we would have to write if we had solved this program with while loops instead of for loops:

```
1 public class FarmerBrownWhile
 2 {
 3    public static void main( String[] args )
 4    {
 5       int s = 1;
 6       while ( s <= 100 )
 7       {
 8          int g = 1;
 9          while ( g <= 100 )
10          {
11             int c = 1;
12             while ( c <= 100 )
13             {
14                if ( s+g+c == 100 && 10.00*s + 3.50*g + 0.50*c == 100.00 )
15                {
16                   System.out.print( s + " sheep, " );
17                   System.out.print( g + " goats, and " );
18                   System.out.println( c + " chickens." );
19                }
20                c++;
21             }
22             g++;
23          }
24          s++;
25       }
```

```
26     }
27 }
```

The while loop version is also more *fragile* because it would be easy to accidentally forget to reset a variable to 1 or to increment it at the end of the loop body. Doing this with while loops might be easier to get to compile but it is more likely to have subtle logical errors that compile but don't work as intended.

Study Drills

1. Our code works, but it is not as efficient as it could be. (For example, there is no reason to make the "sheep" loop try 11 or 12 or more sheep because we can't afford them. See if you can change the loop bounds to make the combinations less wasteful.

Exercise 48: Arrays - Many Values in a Single Variable

In this exercise you will learn two new things. The first one is *super* important and the second one is just kind-of neat.

In Java, an "array" is a type of variable with one name ("identifier") but containing more than one variable. In my opinion, you're not a Real Programmer until you can work with arrays. So, that's good news. You're almost there!

```
1 public class ArrayIntro
 2 {
 3     public static void main( String[] args )
 4     {
 5         String[] planets = { "Mercury", "Venus", "Earth", "Mars", "Jupiter", "Saturn",  "Uranus",
"Neptune" };
 6
 7         for ( String p : planets )
 8         {
 9             System.out.println( p + "\t" + p.toUpperCase() );
10         }
11     }
12 }
```

What You Should See

```
Mercury MERCURY
Venus   VENUS
Earth   EARTH
Mars    MARS
Jupiter JUPITER
Saturn  SATURN
Uranus  URANUS
Neptune NEPTUNE
```

On line 5 we declare and define a variable named *planets*. It is not just a String: notice the square brackets. This variable is an *array* of Strings. That means that this one variable holds all eight of those Strings and they are separated into distinct slots so we can access them one at a time.

The curly braces on this line are used for a different purpose than usual. All these values are in quotes because they are Strings. There are commas between each value, then the whole initializer list is in curly braces. And there's a semicolon at the end.

The second new thing in this exercise is a new kind of for loop. (This is sometimes called a "foreach" loop, since it works a bit like a loop in another programming language where the keyword actually is foreach instead of for.)

On line 7 you will see this foreach loop in action. You read it out loud like this: "for each String 'p' in the array 'planets'…."

So inside the body of this foreach loop the String variable *p* will take on a copy of the value of each value in the String array *planets*. That is, the first time through the loop, *p* will contain a copy of the first value in the array ("Mercury").

Then the second time through the loop, *p* will contain a copy of the second value in the array ("Venus"). And so on, until all the values in the array have been seen. Then the loop will automatically stop.

Inside the body of the loop (on line 9) we are just printing out the current value of *p* and an uppercase version of *p*. Just for fun, I guess.

This new kind of for loop only works with compound variables like this: variables that have one name but contain multiple values. Arrays aren't the only sort of compound variable in Java, but we won't be looking at any of the others in this book.

Arrays are a big deal, so that's enough for this exercise. I want to make *absolutely sure* you understand what is happening in this assignment before throwing more on your plate.

Exercise 49: Finding Things in an Array

More with arrays! In this exercise we will examine how to find a particular value. The technique we are using here is sometimes called a "linear search" because it starts with the first slot of the array and looks there, then moves to the second slot, then the third and so on down the line.

```
1 import java.util.Scanner;
 2
 3 public class ArrayLinearSearch
 4 {
 5     public static void main( String[] args )
 6     {
 7         Scanner keyboard = new Scanner(System.in);
 8
 9         int[] orderNumbers = { 12345, 54321, 78753, 101010, 8675309, 31415, 271828 };
10         int toFind;
11
 12         System.out.println("There are " + orderNumbers.length + " orders in the database.");
13
 14         System.out.print("Orders: ");
15         for ( int num : orderNumbers )
16         {
17             System.out.print( num + "  " );
18         }
19         System.out.println();
20
 21         System.out.print( "Which order to find? " );
22         toFind = keyboard.nextInt();
23
 24         for ( int num : orderNumbers )
25         {
26             if ( num == toFind )
27             {
28                 System.out.println( num + " found.");
29             }
30         }
31     }
32 }
```

What You Should See

```
There are 7 orders in the database.
Orders: 12345  54321  78753  101010  8675309  31415  271828
Which order to find? 78753
78753 found.
```

This time the array is named *orderNumbers* and it is an array of integers. It has seven slots. 12345 is the first slot, and 271828 is in the last slot of the array. Each of the seven slots can hold an integer.

When we create an array Java gives us a built-in variable which tells us the capacity of the array. This variable is read-only (you can retrieve its value but not change it) and is called .length. In this case, since the array *orderNumbers*

has seven slots, the variable orderNumbers.length is equal to 7. This is used on line 12.

On line 15 we have a foreach loop to display all the order numbers on the screen. "For each integer 'num' in the array 'orderNumbers'…." So inside the body of this loop, *num* will take on each value in the array one at a time and display them all.

On line 22 we let the human type in an order number. Then we use the loop to let *num* take on each order number and compare them to *toFind* one at a time. When we have a match, we say so.

> (You have to imagine that we have hundreds or thousands of orders in the database instead of just seven and that we print out more than just the order number when we find a match. We'll be getting there soon enough.)

Study Drills

1. We created an int called *num* inside both foreach loops. Could we have just declared the variable once up on line 10 and then removed the int from both loops? Try it and see.
64. Try to change the code so that if the order number is *not* found, it prints out a single message saying so. This is tricky. Even if you aren't successful, give it a good effort before moving on to the next exercise.

Exercise 50: Saying Something Is NOT in an Array

In life, there is a general lack of symmetry between certain types of statements.

> A white crow exists.

This statement is easy enough to prove. Start looking at crows. Once you find a white one, stop. Done.

> No white crows exist.

This statement is *much* harder to prove because to prove it we have to gather up everything in the world that qualifies as a crow. If we have looked at them *all* and not found any white crows, only then can we safely say that *none* exist.

Hopefully you tried the study drill in yesterday's exercise.

```
1 import java.util.Scanner;
 2
 3 public class ItemNotFound
 4 {
 5     public static void main( String[] args )
 6     {
 7         Scanner keyboard = new Scanner(System.in);
 8
 9         String[] heroes = {
10             "Abderus", "Achilles", "Aeneas", "Ajax", "Amphitryon",
11             "Bellerophon", "Castor", "Chrysippus", "Daedalus", "Diomedes",
12             "Eleusis", "Eunostus", "Ganymede", "Hector", "Iolaus", "Jason",
13             "Meleager", "Odysseus", "Orpheus", "Perseus", "Theseus" };
14         String guess;
15         boolean found;
16
17         System.out.print( "Pop Quiz! Name any mortal hero from Greek mythology: " );
18         guess = keyboard.next();
19
20         found = false;
21         for ( String hero : heroes )
22         {
23             if ( guess.equals(hero) )
24             {
25                 System.out.println( "That's correct!" );
26                 found = true;
27             }
28         }
29
30         if ( found == false )
31         {
32             System.out.println( "No, " + guess + " wasn't a Greek mortal hero." );
33         }
34     }
35 }
```

What You Should See

```
Pop Quiz! Name any mortal hero from Greek mythology: Hercules
No, Hercules wasn't a Greek mortal hero.
```

Most students want to solve this problem by putting another if statement (or an else) inside the loop to say "not found". But this can not work. If I want to know

if something is found, it is okay to say so as soon as I find it. But if I want to know if something was *never* found, you have to wait until the loop is over before you know for sure.

So in this case I use a technique called a “flag”. A flag is a variable that starts with one value. The value is changed if something happens. Then later in the program you can use the value of the flag to see if the thing happened or not.

My flag variable is a Boolean called *found*, which is set to false on line 20. If a match is found, we say so *and* change the flag to true on line 26. Notice that inside the loop there is no code that can change the flag to false, so once it has been flipped to true it will stay that way.

Then on line 30, after the loop is done, you can examine the flag. If it is still false, then we know the if statement inside the loop was never true and therefore we never found what we were looking for.

Exercise 51: Arrays Without Foreach Loops

As you might noticed by now, arrays and foreach loops are designed to work together well. But there are situations where what we have been doing won't work.

- A foreach loop can't iterate through an array *backward*; it can only go forward.
- A foreach loop can't be used to change the values in the slots of the array. The foreach loop variable is a read-only copy of what's in the array and changing it doesn't change the array.

In addition, we have only been putting values into an array using an initializer list (the curly braces thing), which has its own limitations:

- An initializer list only works when the array is being declared; you can't use it elsewhere in the code.
- An initializer list is best suited for relatively small arrays, if you have 1000 values in the array, an initializer list will be no fun.
- Initializer lists don't help us if we want the values in the array to come from a file or some other place we don't have when we are typing the code.

So there is another way to store values in an array and access them. In fact, it is more common than what you have been doing. Using square brackets and a slot number, we can access the slots of an array individually.

```
1 public class ArraySlotAccess
 2 {
 3    public static void main( String[] args )
 4    {
 5       int[] arr = new int[3];
 6       int i;
 7
 8       arr[0] = 0;
 9       arr[1] = 0;
10        arr[2] = 0;
11
 12        System.out.println("Array contains: " + arr[0] + " " + arr[1] + " " + arr[2] );
13
 14        // Fill each slot of this array with a random number 1-100
15        arr[0] = 1 + (int)(Math.random()*100);
16        arr[1] = 1 + (int)(Math.random()*100);
17        arr[2] = 1 + (int)(Math.random()*100);
18
 19        // Display them again.
20        System.out.println("Array contains: " + arr[0] + " " + arr[1] + " " + arr[2] );
21
 22        // This is a bit silly, but try to understand it.
23        i = 0;
24        arr[i] = 1 + (int)(Math.random()*100);
25        i = 1;
26        arr[i] = 1 + (int)(Math.random()*100);
27        i = 2;
28        arr[i] = 1 + (int)(Math.random()*100);
29
 30        // Display them again.
31        System.out.print("Array contains: ");
```

```
32        i = 0;
33        System.out.print(arr[i] + " ");
34        i = 1;
35        System.out.print(arr[i] + " ");
36        i = 2;
37        System.out.print(arr[i] + " ");
38        System.out.println();
39
 40        // This is even more silly but it works. Can you guess where this is headed?
41        i = 0;
42        arr[i] = 1 + (int)(Math.random()*100);
43        i++;
44        arr[i] = 1 + (int)(Math.random()*100);
45        i++;
46        arr[i] = 1 + (int)(Math.random()*100);
47        i++;
48
 49        // Display them again.
50        System.out.print("Array contains: ");
51        i = 0;
52        System.out.print(arr[i] + " ");
53        i++;
54        System.out.print(arr[i] + " ");
55        i++;
56        System.out.print(arr[i] + " ");
57        i++;
58        System.out.println();
59
 60        // Ah! Let's just use a regular 'for' loop!
61        for ( i=0 ; i < arr.length ; i++ )
62        {
63           arr[i] = 1 + (int)(Math.random()*100);
64        }
65
 66        // Display them again.
67        System.out.print("Array contains: ");
68        for ( i=0 ; i < arr.length ; i++ )
69        {
70           System.out.print(arr[i] + " ");
71        }
72        System.out.println();
73     }
74 }
```

What You Should See

```
Array contains: 0 0 0
Array contains: 98 49 18
Array contains: 83 77 1
 Array contains: 62 74 32
 Array contains: 40 17 54
```

On line 5 we are creating an array of integers *without* using an initializer list. The [3] means that the array has a capacity of 3. Since we didn’t provide values, every slot in the array starts out with a value of 0 stored in it. Once an array has

been created, its capacity can't be changed.

On lines 8 through 10 there is a surprise. The array has 3 slots, but the slot numbers are 0-based. (The number that refers to an array slot is called an "index". Collectively these ought to be called "indices" (INN-duh-SEEZ) but most people just say "indexes".)

So the first slot in an array is index 0. This array can hold three values, so the last index is 2. There is nothing you can do about this except get used to it. So arr.length is 3, but there is not a slot with index 3. This will probably be a source of bugs for you at first, but eventually you will learn.

Anyway, lines 8 through 10 store the value 0 into all three slots in the array. (Which is the value that was already in them, so this code doesn't do anything useful.)

On line 12 we print out all three current values in the array so you can see that they're all zero.

On lines 15 through 17 we put random numbers into each slot of the array. And print them out again on line 20.

Starting on line 22 I have done something silly. Try to withhold judgment until the end of the exercise.

Forgetting about why you might *want* to do it, do you see that line 24 is essentially identical to line 15? Line 24 stores a random number into a spot in the array. Which spot? The index depends on the current value of *i*. And *i* is currently 0. So we are storing the random number into the slot with index 0. Okay?

So on line 25 we *change* the value of *i* from 0 to 1. Then on line 26 we store a random value in the slot indexed by the value of *i*, so index 1. Clear? Weird, but legal.

I have used similar shenanigans on lines 31 through 38 to display all the values on the screen again. Now, this is clearly objectively worse than what I was doing on line 20. I mean, it took me 8 lines of code to do what I had been doing in one line. (Stay with me.)

On lines 40 through 47 we do something that might even be worse than lines 22 through 28. Lines 41 and 42 are the same, but instead of putting a 1 directly into *i* on line 43, I just say "increase the value of *i* by 1." So *i* had contained a 0; it contains a 1 after line 43 is done.

Pretty much the only advantage to this approach is that at least copy-and-paste is easier. Lines 42 and 43 are *literally identical* to lines 44 and 45. And the same for lines 46 and 47. I mean, like byte-for-byte the same.

We display them in a similar silly fashion on lines 50 through 58.

But then maybe it occurs to you. "Why would I bother to type the exact same lines three times in a row when I could just…." You know a thing that allows you to repeat a chunk of code while making a single variable increase by one each time, right?

That's right: a for loop is just the thing. Not so silly after all, am I?

Lines 61 through 64 are the same as lines 41 through 47 except that we let the

for loop handle the repeating and the changing of the index. The initialization expression (the first part) of the for loop sets *i* to start at 0, which happens to be the smallest legal index for an array. The condition says "repeat this as long as *i* is less than arr.length (which is 3)." And note that that is *less than*, not less than or equal to, which would be too far. The update expression (the third part) just adds 1 to *i* each time.

Lines 67 through 72 display the values on the screen.

Here's the thing: this sort of code on lines 61 through 72 might seem a little bit complex, but working with arrays in Java you end up writing code like this *all the time*. I cannot even tell you how many times I have written a for loop just like that for working with an array.

In fact, if your question is "How do I _____ an array?" (Fill in the blank with any task you like.) The answer is "With a for loop." Pretty much guaranteed.

Study Drills

1. At the top of the code, change it so the array has a capacity of 1000 instead of 3. Don't change any other code and recompile and run it again. Guess what? Those for loops at the bottom might have been a little more work to write and to understand, but once written they work just as well for 1000 values as for 3. And that's pretty cool.

Exercise 52: Lowest Temperature

Before we move on from arrays, this exercise will pull together functions, loops, arrays and reading from files to do something (hopefully) interesting!

I have created a text file containing the average daily temperature in Austin, Texas from January 1, 1995 through June 23, 2013. There are a few data points missing, so there are a total of 6717 temperatures in the file. You can see the numbers here:

- http://learnjavathehardway.org/txt/avg-daily-temps-atx.txt

The values are in degrees Fahrenheit. This exercise will read all the values from the file (directly off the Internet, even) into an array of doubles and then use a loop to find the lowest temperature in that entire 17-1/2 year range. Sound interesting? Let's go.

```
1 import java.net.URL;
 2 import java.util.Scanner;
 3
 4 public class LowestTemperature
 5 {
 6     public static void main(String[] args) throws Exception
 7     {
 8         double[] temps =
arrayFromUrl("http://learnjavathehardway.org/txt/avg-daily-temps-atx.txt");
 9
 10         System.out.println( temps.length + " temperatures in database.");
11
 12         double lowest = 9999.99;
13
 14         for ( int i=0; i<temps.length; i++ )
15         {
16             if ( temps[i] < lowest )
17             {
18                 lowest = temps[i];
19             }
20         }
21
 22         System.out.print( "The lowest average daily temperature was " );
23         System.out.println( lowest + "F (" + fToC(lowest) + "C)" );
24     }
25
 26     public static double[] arrayFromUrl( String url ) throws Exception
27     {
28         Scanner fin = new Scanner((new URL(url)).openStream());
29         int count = fin.nextInt();
30
 31         double[] dubs = new double[count];
32
 33         for ( int i=0; i<dubs.length; i++ )
34             dubs[i] = fin.nextDouble();
35         fin.close();
36
 37         return dubs;
38     }
39
```

```
 40     public static double fToC( double f )
41     {
42         return (f-32)*5/9;
43     }
44
 45 }
```

What You Should See

```
6717 temperatures in database.
The lowest average daily temperature was 22.1F (-5.499999999999999C)
```

(If you have to run this program on a machine without Internet access, then the code won't work. Since you know how to read from a text file already, you could modify it yourself to read from a local file (a file in the same folder as your code instead of on the Internet). But if you are lazy I have listed an alternate version down below.)

Well, right off the bat I have thrown a curve ball. On line 8 we declare an array of doubles named *temps*, but instead of just doing the normal thing and setting its capacity like this:

```
double[] temps = new double[6717];
```

…I initialize the array with an array that is the return value from a function! So let's look down at the function before we proceed.

On line 26 the function definition begins. The function is called arrayFromUrl() and it has one parameter: a String. And it returns what? It returns not a double but double[] (an array of doubles).

On line 28 we create a Scanner object to read data from a file, but instead of getting the data from a file, we get the information from a URL. One of the nice things about Java is that this is only a tiny change.

Now, I do a trick with my text file that I learned many years ago. At the time I am writing this chapter, my file contains 6717 temperatures. But maybe you are reading this a year later and I want to update the file to add more temperatures. So the *first* line of the file is just a number: 6717. Then after that I have 6717 lines of temperatures, one per line.

On line 29 in this code I read the *count* from the first thing in the file. And I use that count to decide how big my array should be on line 31. So six months from now if I decide to add more temperatures to the file, all I have to do is change the first line of my file to match and this code will still work. Not a bad trick, eh?

On line 31 we define an array of doubles with *count* slots. (Currently 6717.)

On line 33 there's a for loop that iterates through each slot in the array, and on line 34 we read a double from the file each time (fin.nextDouble()) and store it into the next indexed slot in array.

Then when the loop ends, I close() the file. Then on line 37 the array is returned from the function and that array is what is stored into the array *temps* back on

line 8 of main().

On line 10 we print out the current length of the array to make sure nothing went wrong with the reading.

On line 12 we create a variable that will eventually hold the lowest temperature in the whole array. At first we put a really large value in there, though.

Line 14 is another for loop that is going to give us all the legal indexes in the array. In this case, since the array has 6717 values in it, the indexes will run from 0 to 6716.

Line 16 compares the value we are currently looking at in the array (depending on the current value of *i*). If that value is less than whatever is in *lowest*, then we have a new record! On line 18 we replace what used to be in *lowest* with this new smaller value.

And the loop continues until all the values in the array have been compared. When the loop ends, the variable *lowest* now actually does contain the smallest value.

There is a small function down on lines 40 through 43 to convert a temperature from degrees Fahrenheit to degrees Celsius. So on line 23 we display the lowest temperature as it came from the file and also converted to Celsius.

You may be thinking that 22.1F (-5.5C) is not a very cold temperature. Well, that's Texas for you. Also remember that the temperatures aren't the lowest temperature of the day, they are the average of 24 hourly temperature samples for each day.

Study Drills

1. Change the code to display both the lowest average daily temperature and and the highest.
65. Try to find another temperature file online for a city closer to where you live and change your code to read from it instead!

(I mentioned it above, but this is the modified code to read the temperature data from a local file in case you can't run your Java program on a machine with Internet access.)

```
1 import java.io.File;
 2 import java.util.Scanner;
 3
 4 public class LowestTemperatureLocal
 5 {
 6     public static void main(String[] args) throws Exception
 7     {
 8         double[] temps;
 9         double lowest = 9999.99;
10
 11         // Read values from file
12         Scanner fin = new Scanner(new File("avg-daily-temps-atx.txt"));
13         temps = new double[fin.nextInt()];
14
 15         System.out.println( temps.length + " temperatures in database.");
```

```
16
 17        for ( int i=0; i<temps.length; i++ )
18           temps[i] = fin.nextDouble();
19        fin.close();
20
 21        for ( int i=0; i<temps.length; i++ )
22           if ( temps[i] < lowest )
23              lowest = temps[i];
24
 25        System.out.print( "The lowest average daily temperature was " );
26        System.out.println( lowest + "F (" + fToC(lowest) + "C)" );
27     }
28
 29     public static double fToC( double f ) { return (f-32)*5/9; }
30 }
```

Exercise 53: Mailing Addresses (Records)

Today's exercise is about what I call "records". In the programming languages C and C++ they are called "structs". An array is a bunch of different values in one variable where the values are *all the same type* and they are distinguished by *index* (slot number). A record is a few different values in one variable but the values can be different types and they are distinguished by *name* (usually called a "field").[12]

Type the following code into a single file named MailingAddresses.java. (The line that says class Address is correct but you can't name your file Address.java or it won't work.)

```
1 class Address
 2 {
 3     String street;
 4     String city;
 5     String state;
 6     int zip;
 7 }
 8
 9 public class MailingAddresses
10 {
11     public static void main(String[] args)
12     {
13         Address uno, dos, tres;
14
 15         uno = new Address();
16         uno.street = "191 Marigold Lane";
17         uno.city   = "Miami";
18         uno.state  = "FL";
19         uno.zip    = 33179;
20
 21         dos = new Address();
22         dos.street = "3029 Losh Lane";
23         dos.city   = "Crafton";
24         dos.state  = "PA";
25         dos.zip    = 15205;
26
 27         tres = new Address();
28         tres.street = "2693 Hannah Street";
29         tres.city   = "Hickory";
30         tres.state  = "NC";
31         tres.zip    = 28601;
32
 33         System.out.println(uno.street + "\n" + uno.city + ", " + uno.state + "  " + uno.zip +
"\n");
34         System.out.println(dos.street + "\n" + dos.city + ", " + dos.state + "  " + dos.zip + "\n");
35         System.out.println(tres.street + "\n" + tres.city + ", " + tres.state + "  " + tres.zip +
```

[12][12]There's only one problem will all of this: Java doesn't actually have records. It turns out that if you make a nested class with no methods and only public variables it works just like a struct even if it isn't the Java Way.

I don't care if it is the Java Way or not. I have been teaching students a long time and I firmly believe that you can't understand object-oriented programming very well if you don't first understand records. So I am going to fake them in a way that works perfectly fine and would be very nice code in lots of different programming languages.

Some die-hard object-oriented Java-head is going to stumble across this exercise and send me an nasty email that I am Doing It Wrong and why am I filling these poor kids' heads with lies? Oh, well.

```
"\n");
36     }
37 }
```

What You Should See

```
191 Marigold Lane
Miami, FL  33179

3029 Losh Lane
Crafton, PA  15205

2693 Hannah Street
Hickory, NC  28601
```

So up on lines 1 through 7 we have defined a record called Address.

(I know it says class, not record. If I could do something about this I promise I would. You should call this a "record" anyway, or a "struct" if you really want. If you call it a "class" it will confuse any Java programmer that loves object-oriented programming and if you call it a "struct" at least the C and C++ programmers will understand you.)

Our record has four fields. The first field is a String named *street*. The second field is a String called *city*. And so on.

Then on line 9 our "real" class starts.

On line 13 we declare three variables named *uno*, *dos* and *tres*. These variables are not integers or Strings; they are records. Of type Address. Each one has four fields in it.

On line 15 we have to store an Address object in the variable because remember we only declared the variables and they haven't been initialized yet.

Once that is taken care of you will see that we can store the String "191 Marigold Lane" into the *street* field of the Address record named *uno*, and that's exactly what we do on line 16.

Line 17 stores the String "Miami" into the *city* field of the record *uno*.

I'm not going to bother to explain what is happening for the rest of the program because I think it is pretty clear. I guess the only thing worth mentioning is that although three of the fields in the record are Strings, the *zip* field is an integer. The fields of a record can be whatever type you want.

Study Drills

1. Create a fourth Address variable on line 13 and change the code to put *your* mailing address in it. Don't forget to print it out at the bottom.

Frequently-Asked Questions

- Where did you get these addresses?

I made them up. I'm fairly certain those streets don't exist in those cities. If by some miracle I made up a real address, let me know and I'll change it.

Exercise 54: Records from a File

This exercise will show you how to read values into a record from a text file. There is also an example of a loop that reads in the entire file, no matter how long it is.

If you run this program on a machine that isn't connected to the Internet, this code won't work as written, although the change is very small. The code accesses this file, which you can download if you need to.

- http://learnjavathehardway.org/txt/s01e01-cast.txt

Type the following code into a single file named ActorList.java. (The line that says class Actor is correct but you can't name your file Actor.java or it won't work.)

```
1 import java.util.Scanner;
 2
 3 class Actor
 4 {
 5     String name;
 6     String role;
 7     String birthdate;
 8 }
 9
 10 public class ActorList
11 {
12     public static void main(String[] args) throws Exception
13     {
14         String url = "http://learnjavathehardway.org/txt/s01e01-cast.txt";
15         // Scanner inFile = new Scanner(new java.io.File("s01e01-cast.txt"));
16         Scanner inFile = new Scanner((new java.net.URL(url)).openStream());
17
 18         while ( inFile.hasNext() )
19         {
20             Actor a = getActor(inFile);
21             System.out.print(a.name + " was born on " + a.birthdate);
22             System.out.println(" and played the role of " + a.role);
23         }
24         inFile.close();
25     }
26
 27     public static Actor getActor( Scanner input )
28     {
29         Actor a = new Actor();
30         a.name = input.nextLine();
31         a.role = input.nextLine();
32         a.birthdate = input.nextLine();
33
 34         return a;
35     }
36 }
```

What You Should See

```
Sean Bean was born on 1959-04-17 and played the role of Eddard 'Ned' Stark
Mark Addy was born on 1964-01-14 and played the role of Robert Baratheon
Nikolaj Coster-Waldau was born on 1970-07-27 and played the role of Jaime Lannister
Michelle Fairley was born on 1964 and played the role of Catelyn Stark
```

This time our record is called Actor and has three fields, all of which are Strings.

On line 16 we create a Scanner object that is connected to the Internet address of the input text file. Did you notice that I didn't import java.net.URL at the top? You only need to import a class if you want to be able to type the short version of the class name.

In this case, if I had imported java.net.URL at the top of the code, I could have just written:

```
Scanner inFile = new Scanner((new URL(url)).openStream());
// instead of
Scanner inFile = new Scanner((new java.net.URL(url)).openStream());
```

Sometimes if I am going to be using a class only once, I'd rather just use the full name in my code instead of bothering to import it. I used the same trick on line 15; instead of importing java.io.File I just used the full classname here.

(If your machine doesn't have Internet access, remove the two slashes at the beginning of line 15 so that it is no longer a comment and then *add* two slashes at the beginning on line 16 to *make* it a comment. Then the program will read the file locally instead of over the Internet.)

Whether you open the file from the Internet or your own machine, after line 17 we have a Scanner object named *inFile* which is connected to a text file.

A lot of the time when we are reading from a text file, we don't know in advance how long it is going to be. In the lowest temperature exercise I showed you one trick for dealing with this: storing the number of items as the first line of the file. But a more common technique is the one I have used here: just use a loop that repeats until we reach the end of the file.

The .hasNext() method of the Scanner object will return true if there is more data that hasn't been read in yet. And it returns false if there is no more data. So on line 18 we create a while loop that repeats as long as .hasNext() continues to return true.

Before we look at line 20 let us skip down to lines 27 through 35 where I have created a function that will read all the data for a single Actor record from the file.

The function is called *getActor*. It has one parameter: a Scanner object! That's right, you pass in an already-open Scanner object to the function and it reads from it. And the *getActor* function returns an Actor. It returns an entire record.

If we are going to return an Actor object from the function we need a variable of type Actor to return, so we define one on line 29. I just called it *a* because inside the function we don't know anything about the purpose of this variable. You should give good names to variables but in a situation like this a short, meaningless name like *a* is perfectly fine.

Lines 30 through 32 read three lines from the text file and store them into the three fields of the record.

Then the function has done its job and we return the record back up to line 20 in main().

Why must we create an Actor variable named *a* here in main() and also down in the function? Because of variable scope. A variable is only in scope (aka "visible") within the block in which it was declared. Period. It doesn't matter if the variable is "returned" from a function or not because remember it is not the variable itself which is returned but a copy of the *value* of the variable.

There is an Actor variable called *a* declared (and defined) on line 20 in main(), but that variable goes out of scope when the close curly brace occurs on line 23. There is a different Actor variable called *a* declared (and defined) on line 29 in the getActor() function, but it goes out of scope when the close curly brace occurs on line 35.

Okay, back to line 20. The variable *a* gets its value from the return value of the function getActor(). We pass in the open Scanner object *inFile* as the argument to the function and it returns to us an Actor object with all its fields filled in.

(Why is the argument called *inFile* and the parameter named *input*? Because they are not the same variable. The parameter *input* is declared on line 27 and gets a copy of the value from the argument *inFile*. They are two different variables that have the same value.)

After all that, lines 21 and 22 are pretty boring: they simply display the values of all the fields of the record. On line 23 the loop repeats back up to check the condition again: now that we have read another record from the file, does the file still have more? If so, keep looping. If not, skip down to line 24 where we close the file.

Notice that both in the function and in the while loop in main() the variable *a* only holds one record at a time. We read all the records from the file and print them all out on the screen, but when the program is finishing its last time through the loop, the variable *a* only holds the most recent record. All the other records are still in the file and have been displayed on the screen, but their values are not currently being held in any variables.

We can fix that, but not until the next exercise.

Exercise 55: An Array of Records

Records are great and arrays are better, but there is not much in this life you can't code when you put records *into* an array.

```
1 class Student
 2 {
 3     String name;
 4     int credits;
 5     double gpa;
 6 }
 7
 8 public class StudentDatabase
 9 {
10     public static void main( String[] args )
11     {
12         Student[] db;
13         db = new Student[3];
14
 15         db[0] = new Student();
16         db[0].name = "Esteban";
17         db[0].credits = 43;
18         db[0].gpa = 2.9;
19
 20         db[1] = new Student();
21         db[1].name = "Dave";
22         db[1].credits = 15;
23         db[1].gpa = 4.0;
24
 25         db[2] = new Student();
26         db[2].name = "Michelle";
27         db[2].credits = 132;
28         db[2].gpa = 3.72;
29
 30         for ( int i=0; i<db.length; i++ )
31         {
32             System.out.println("Name: " + db[i].name);
33             System.out.println("\tCredit hours: " + db[i].credits);
34             System.out.println("\tGPA: " + db[i].gpa + "\n");
35         }
36
 37         int max = 0;
38         for ( int i=1; i<db.length; i++ )
39             if ( db[i].gpa > db[max].gpa )
40                 max = i;
41
 42         System.out.println(db[max].name + " has the highest GPA.");
43     }
44 }
```

What You Should See

```
Name: Esteban
        Credit hours: 43
        GPA: 2.9
```

```
Name: Dave
      Credit hours: 15
      GPA: 4.0

Name: Michelle
      Credit hours: 132
      GPA: 3.72

Dave has the highest GPA.
```

When you see the square brackets just to the right of something in a variable definition, that's an "array of" whatever. In fact, since this book is almost over, maybe I should explain that public static void main business. At least partially.

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

> This line declares a function named *main*. That function requires one parameter: an array of Strings named *args* (which is short for "arguments"). The function doesn't return any value; it is void.

Anyway.

Line 12 declares *db* as a variable that can hold an "array of Students". There's no array yet, just a variable that can potentially hold one. Just like when we say…

```
int n;
```

…there's no integer yet. The variable *n* can potentially hold an integer, but there's no number in it yet. *n* is declared but undefined. In the same way, once line 12 has finished executing, *db* is a variable that *could* refer to an array of Students, but is still undefined.

Fortunately we don't have to wait long; line 13 initializes *db* by creating an *actual* array of Students with three slots. At this point *db* is defined, db.length is 3 and *db* has three legal indexes: 0, 1 and 2.

Okay, at this point, *db* is an array of Student records. Except that it isn't. *db* is an array of Student *variables*, each of which can *potentially* hold a Student record, but none of which do. All three slots in the array are undefined.

> (Technically they contain the value null, which is the special value that reference variables in Java have when there's no object in them yet.)

So on line 15 it is important that a Student object is created and stored into the first slot (index 0) of the array. Then on line 16 we can store a value into the *name* field of the Student record which is in index 0 of the array *db*.

Let's trace it from the outside in:

expression	***type***	***description***
db	Student[]	an array of Student records
db[0]	Student	a single Student record (the first one)

db[0].name	String	the *name* field of the first Student in the array
db.name	error	the whole array doesn't have a single *name* field

So line 16 stores a value into the *name* field of the first record in the array. Lines 17 and 18 store values into the remaining fields in that record. Lines 20 through 28 create then fill the other two records in the array

On lines 30 though 34 we use a loop to display all the values on the screen.

Then lines 37 through 42 find the student with the highest GPA. This is worth explaining in more detail. On line 37 an int called *max* is defined. But *max* is not going to hold the *value* of the highest GPA; it is going to hold only its *index*.

So when I put 0 into *max* I mean "As this point in the code, as far as I know, the highest-scoring student is in slot 0." This is probably not true, but since we haven't looked at any of the values in the database yet it is as good a starting place as any.

Then on line 38 we set up the loop to look through each slot of the array. Notice, however, that the loop starts with index 1 (the second slot). Why?

Because *max* is already 0. So if *i* started at 0 too then the if statement would be comparing

```
if ( db[0].gpa > db[0].gpa )
```

…which is a waste. So by starting *i* at 1, then the first time through the loop the if statement makes the following comparison instead:

```
if ( db[1].gpa > db[0].gpa )
```

"If Dave's GPA is greater than Esteban's GPA, then change *max* from 0 to the current value of *i* (1)."

So by the time the loop is over, *max* contains the **index** of the record with the highest GPA. Which is exactly what we display on line 42.

Study Drills

1. Change the array to have a capacity of 4 instead of 3. Change nothing else and compile and run the program. Do you understand why the program blows up?
66. Now add some more code to put values into the fields for your new student. Give this new student a higher GPA than "Dave" and confirm that the code correctly labels them as having the highest GPA.
67. Change the code so that it finds the person with the fewest credits instead of the person with the highest GPA.

Exercise 56: Array of Records from a File (Temperatures Revisited)

This exercise populates an array of records from a file on the Internet. By now you should know if you need to download a copy of this file or if your computer can just open it from the Internet.

- http://learnjavathehardway.org/txt/avg-daily-temps-with-dates-atx.txt

Unlike all the other files you have used so far in this book this data file is exactly the way I downloaded it from the University of Dayton's average daily temperature archive. This means three things:

1. There is no number in the first line of the file telling us how many records there are.
68. In addition to the temperature each record also has the month, day and year for the sample.
69. There is bad data in the file. In particular, "We use '-99' as a no-data flag when data are not available."

So some days have a temperature of -99. We will have to handle this in the code.

```
1 import java.util.Scanner;
 2
 3 class TemperatureSample
 4 {
 5     int month, day, year;
 6     double temperature;
 7 }
 8
 9 public class TemperaturesByDate
10 {
11     public static void main(String[] args) throws Exception
12     {
13         String url = "http://learnjavathehardway.org/txt/avg-daily-temps-with-dates-atx.txt";
14         Scanner inFile = new Scanner((new java.net.URL(url)).openStream());
15
16         TemperatureSample[] tempDB = new TemperatureSample[10000];
17         int numRecords, i = 0;
18
19         while ( inFile.hasNextInt() && i < tempDB.length )
20         {
21             TemperatureSample e = new TemperatureSample();
22             e.month = inFile.nextInt();
23             e.day   = inFile.nextInt();
24             e.year  = inFile.nextInt();
25             e.temperature = inFile.nextDouble();
26             if ( e.temperature == -99 )
27                 continue;
28             tempDB[i] = e;
29             i++;
30         }
31         inFile.close();
32         numRecords = i;
33
34         System.out.println(numRecords + " daily temperatures loaded.");
```

```
35
 36        double total = 0, avg;
37        int count = 0;
38        for ( i=0; i<numRecords; i++ )
39        {
40           if ( tempDB[i].month == 11 )
41           {
42              total += tempDB[i].temperature;
43              count++;
44           }
45        }
46
 47        avg = total / count;
48        avg = roundToOneDecimal(avg);
49        System.out.println("Average daily temperature over " + count + " days in November: " +
avg);
50     }
51
 52     public static double roundToOneDecimal( double d )
53     {
54        return Math.round(d*10)/10.0;
55     }
56 }
```

What You Should See

```
6717 daily temperatures loaded.
Average daily temperature over 540 days in November: 59.7
```

Lines 3 through 7 declare our record, which will store a single average daily temperature value (a double) but also has fields for the month, day and year.

Line 16 defines an array of records. We have a problem, though. We can't define an array without providing a capacity and we don't know the capacity we need until we see how many records are in the file. There are three possible solutions to this problem:

1. Don't use an array. Use something else like an array that can automatically grow as you add entries. This is actually probably the right solution, but that "something else" is beyond the scope of this book.
70. Read the file twice. Do it once just to count the number of records and then create the array with the perfect size. Then read it again to slurp all the values into the array. This works but it's slow.
71. Don't worry about making the array the right size. Just make it "big enough". Then count how many records you actually have while reading them in and use that count instead of the array's capacity for any loops. This is not perfect, but it works and it's easy. Writing software sometimes requires compromise, and this is one of them.

So line 16 declares the array and defines it to have ten thousand slots: "big enough."

On line 19 we start a loop to read all the values from the file. We are using an index variable *i* to keep track of which slot in the array needs to be filled next. So our loop keeps going as long as the file has more integers in it **and** we

haven’t run out of capacity in our array.

Just because we took a shortcut by making our array “big enough” doesn’t mean we are going to be stupid about it. If the file ended up bigger than our array’s capacity we want to stop reading the file too early rather than blow up the program with an ArrayIndexOutOfBounds exception.

Line 21 defines a TemperatureSample record named *e*. Lines 22 through 25 load the next few values from the file into the appropriate fields of that record.

But! Remember that there are “missing” values in our file. Some days have a temperature reading of -99, so we put in an if statement on line 26 to detect that before we put them into our database.

Then on line 27 there is something new: the Java keyword continue. continue is only legal inside the body of a loop. And it means “skip the rest of the lines of code in the body of the loop and just go back up to the top for the next iteration.”

This effectively throws away the current (invalid) record because it skips lines 28 and 29, which store the current record in the next available slot in the array and then increment the index.

Some people don’t like to use continue and would write it like this:

```
if ( e.temperature != -99 )
{
   tempDB[i] = e;
   i++;
}
```

And that’s perfectly fine, too. Only put this entry into the array when the temperature is *not* -99. I prefer using continue because the code feels cleaner to me but reasonable people disagree. Do whichever makes the most sense to you.

Once the loop is done on line 31 we make sure to close the file and then store the final index into *numRecords* so we can use it instead of tempDB.length for any loops. After all, we made the array bigger than we needed and the last 3283 slots (in this example) are empty. Not only is looping only up to *numRecords* slightly more efficient, we can avoid examining any invalid records that way.

On line 34 we display the number of records on the screen, which can help you to see if anything went wrong while reading.

Lines 36 through 45 loop through all our records. Any record with a *month* field of 11 (November) gets added to a running total, and we also count the total number of matching records while we are at it.

Then when that loop is over, we can get the average daily temperature of all November days in the database by dividing the sum by the count.

Now, my first version of this program had an overall average temperature of 59.662962962963. Not only does this look bad but it’s not correct: all the input temperatures were only accurate to a tenth of a degree. So displaying a result with a dozen significant figures looks more accurate than it really is.

So on lines 52 through 55 you will find a tiny little function to round to one

decimal place. Java doesn't have a built-in function for this as far as I know, but it *does* have a built-in function to round to the nearest whole number: Math.round(). So I multiply the number by ten, round it and then divide by ten again. Maybe there's a better way to do that but I like it.

Line 48 passes the average temperature as the argument to my function and then takes the rounded return value and stores that as the new value of *avg*.

Study Drills

1. Visit the University of Dayton's temperature archive and download a file with temperature data for a city near you! Make your code read data from that file instead.
72. Change the code to find out other things, like the highest temperature in February or whatever suits your fancy.
73. Try printing an entire TemperatureSample record on the screen. Something like this:

```
TemperatureSample ts = tempDB[0];
System.out.println( ts );
```

Notice that isn't printing an integer like ts.year or a double like ts.temperature; it is attempting to display a whole record on the screen. Compile and run the file. What gets displayed on the screen?

Try changing the index to pull different values out of the array and see how it changes what gets printed.

Exercise 57: A Deck of Playing Cards

Before this book ends I need to show you how to use an array of records to simulate a deck of playing cards.

```
1 class Card
 2 {
 3     int value;
 4     String suit;
 5     String name;
 6
 7     public String toString()
 8     {
 9        return name + " of " + suit;
10     }
11 }
12
 13 public class PickACard
14 {
15      public static void main( String[] args )
16     {
17        Card[] deck = buildDeck();
18        // displayDeck(deck);
19
 20        int chosen = (int)(Math.random()*deck.length);
21        Card picked = deck[chosen];
22
 23        System.out.println("You picked a " + picked + " out of the deck.");
24        System.out.println("In Blackjack your card is worth " + picked.value + " points.");
25     }
26
 27     public static Card[] buildDeck()
28     {
29        String[] suits = { "clubs", "diamonds", "hearts", "spades" };
30        String[] names = { "ZERO", "ONE", "two", "three", "four", "five", "six",
31           "seven", "eight", "nine", "ten", "Jack", "Queen", "King", "Ace" };
32
 33        int i = 0;
34        Card[] deck = new Card[52];
35
 36        for ( String s: suits )
37        {
38           for ( int v = 2; v <= 14 ; v++ )
39           {
40              Card c = new Card();
41              c.suit = s;
42              c.name = names[v];
43              if ( v == 14 )
44                 c.value = 11;
45              else if ( v > 10 )
46                 c.value = 10;
47              else
48                 c.value = v;
49
 50              deck[i] = c;
51              i++;
52           }
53        }
54        return deck;
55     }
56
 57     public static void displayDeck( Card[] deck )
58     {
59        for ( Card c : deck )
60           System.out.println(c.value + "\t" + c);
61     }
```

```
62 }
```

What You Should See

```
You picked a seven of hearts out of the deck.
In Blackjack your card is worth 7 points.
```

Of course, even though this is almost the final exercise, I can't resist sneaking in some more new things in. You want to learn something new, don't you?

First of all, I snuck a function into the record. (Actually, because this function is inside a class it isn't a function but a "method".)

This method is named *toString*. It has no parameters and returns a String. In the body of this method we create a String by concatenating the *name* field, the *suit* field, and the word " of ". The method doesn't need any parameters because it has access to the fields of the record. (In fact, that is what makes it a "method" and not a "function".)

Otherwise, the Card record is hopefully what you would expect: it has fields for the value of the card (2-11), the suit name and the name of the card itself.

On lines 17 though 24 you can see the main(), which is really short. Line 17 declares an array of cards and initializes it using the return value of the buildDeck() function.

Line 18 is commented out, but when I was writing this program originally I used the displayDeck() function to make sure that buildDeck() was working correctly.

Line 20 chooses a random number between 0 and deck.length - 1. You might notice that this is exactly the range of legal indexes into the array, and that is not a coincidence.

In fact, you could also say that line 20 chooses a random index into the array or that line 20 chooses a slot of the array randomly.

Then on line 21 we declare a new Card variable called *picked* and give it a value from the randomly-chosen slot of the array.

Line 23 looks pretty boring but there is actually magic happening. What type of variable is *picked*? It is a Card. Normally when you try to print an entire record on the screen like this, Java doesn't know which fields you want printed or in what order so it just prints garbage on the screen. (You saw that in the Study Drill for the previous exercise, right?)

But if you provide a method called toString() inside your record, which returns a String and has no parameters, then in a situation like this Java will call that method behind the scenes. It will take the return value and print that out instead of garbage.

So line 23 will print on the screen the result of running the picked card's toString() method.

By contrast, line 24 really is boring. It prints out the *value* field of the chosen card.

Before we get to buildDeck(), which is the most complex part of this exercise, let us skip down to the displayDeck() function. displayDeck() expects you to pass in an array of Cards as an argument.

Then on line 59 we see something we haven't seen for a few exercises: a foreach loop. This says "for each Card *c* in the deck…" And since there is only one line of code in the body of this for loop, I omitted the curly braces.

Line 60 displays the value of the current card, a tab, and then the result of calling the toString() method on behalf of Card *c*.

Okay, let us tackle this buildDeck() function. buildDeck() doesn't need any parameters because it just creates the deck out of nothing. It does return a value, though: an array of Cards.

On lines 29 through 31 we create two arrays of Strings. The first one (on line 29) contains the names of the suits. The second one contains the names of the cards.

You may notice that I have a card called "ZERO" and another called "ONE". Why? This is so I can this array as a "lookup table". I am going to write my loop so that my card values go from 2 through 14 and I want the word "two" to have *index* 2 in this array. So I needed to put some Strings into slots 0 and 1 to take up space.

Originally I had just put in two empty Strings like so:

```
String[] names = { "", "", "two", "three", "four", "five", "six",
```

…but then I was worried that if I had a bug in my code it would be hard to tell if nothing was being printed or if it was the value of names[0] (or names[1]). So I put words in for those two indexes but made them all-caps so it would catch my attention if they got printed out.

On line 33 we create *i*, which will keep track of which index needs to have a Card put into it next.

And line 34 defines our array of 52 cards (indexed 0` through ``51).

Line 36 is another foreach loop. The variable *s* is going to be set equal to "clubs", then "diamonds", then "hearts" and then finally "spades".

Line 38 is another for loop but this one is *nested*. Remember that means that this loop is going to make *v* go from 2 through 14 before the outer loop ever changes *s* from "clubs".

Line 40 defines a Card named *c*. On line 41 we set the *suit* field of this card to whatever value is currently in *s* ("clubs", at first).

Depending on which time through the loop this is, *v* will be some value between 2 and 14, so on line 42 we use *v* as an index into the *names* array. That is, when *v* is 5 we go into the sixth(!) slot of the array, where we will find the String "five". We put a copy of this value into the *name* field of the current card.

Lines 43 through 48 store an integer from 2 to 11 into the *value* field of the current card. We needed *v* to go from 2-14 for our lookup table, but now that that is already done, we need to make sure that no card gets a value of 12-14.

Card number 14 is the ace, so we use 11 for the card value. Then card numbers 11, 12 and 13 are the face cards, so they all have 10 for their card values. And any other card value is fine as-is.

Finally we store this card into the next available slot of the *deck* (indexed with *i*) and make *i* bigger by 1.

When the nested loops finished we have successfully created all 52 cards in a standard deck and given them card values that match how they are used in Blackjack. Uncomment the call to displayDeck() on line 18 if you want to be sure.

The last thing that buildDeck() needs to do is return the now-full array of Cards so it can be stored into the *deck* variable on line 17 of main().

Study Drills

1. Add a function called shuffleDeck(). It should take in an array of cards as a parameter and return an array of cards. One way to shuffle is to choose two random numbers from 0-51 and "swap" the cards in those slots. Then put that code in a loop that repeats 1000 times or so. This is a bit tricky to get right.

Exercise 58: Final Project - Text Adventure Game

If you have done all the exercises up to this point, then you should be ready for this final project. It is longer than any other exercise that you have done, but it isn't much more difficult than the last few.

Your final exercise is a text-based adventure game *engine*. By *engine* I mean that the code doesn't know anything about the adventure itself; the game is determined 100% by what is in the file. Change the file and you change the game play.

So start by downloading a copy of the game data file and saving it into the same folder as you are going to put your code.

- http://learnjavathehardway.org/txt/text-adventure-rooms.txt

Then, better get started typing. This is a long one, but I think it will be worth it.

```
 1 import java.util.Scanner;
 2
 3 class Room
 4 {
 5     int roomNumber;
 6     String roomName;
 7     String description;
 8     int numExits;
 9     String[] exits = new String[10];
10      int[] destinations = new int[10];
11 }
12
13 public class TextAdventureFinal
14 {
15     public static void main( String[] args )
16     {
17         Scanner keyboard = new Scanner(System.in);
18
19         // initialize rooms from file
20         Room[] rooms = loadRoomsFromFile("text-adventure-rooms.txt");
21
22         // showAllRooms(rooms); // for debugging
23
24         // Okay, so let's play the game!
25         int currentRoom = 0;
26         String ans;
27         while ( currentRoom >= 0 )
28         {
29             Room cur = rooms[currentRoom];
30             System.out.print( cur.description );
31             System.out.print("> ");
32             ans = keyboard.nextLine();
33
34             // See if what they typed matches any of our exit names
35             boolean found = false;
36             for ( int i=0; i<cur.numExits; i++ )
37             {
38                 if ( cur.exits[i].equals(ans) )
39                 {
```

```
40                found = true;
41                // if so, change our next room to that exit's room number
42                currentRoom = cur.destinations[i];
43             }
44          }
45          if ( ! found )
46             System.out.println("Sorry, I don't understand.");
47       }
48
49    }
50
51    public static Room[] loadRoomsFromFile( String filename )
52    {
53       Scanner file = null;
54       try
55       {
56          file = new Scanner(new java.io.File(filename));
57       }
58       catch ( java.io.IOException e )
59       {
60          System.err.println("Sorry, I can't read from the file '" + filename + "'.");
61          System.exit(1);
62       }
63
64       int numRooms = file.nextInt();
65       Room[] rooms = new Room[numRooms];
66
67       // initialize rooms from file
68       int roomNum = 0;
69       while ( file.hasNext() )
70       {
71          Room r = getRoom(file);
72          if ( r.roomNumber != roomNum )
73          {
74             System.err.println("Reading room # " + r.roomNumber + ", but " + roomNum + "
was expected.");
75             System.exit(2);
76          }
77          rooms[roomNum] = r;
78          roomNum++;
79       }
80       file.close();
81
82       return rooms;
83    }
84
85    public static void showAllRooms( Room[] rooms )
86    {
87       for ( Room r : rooms )
88       {
89          String exitString = "";
90          for ( int i=0; i<r.numExits; i++ )
91             exitString += "\t" + r.exits[i] + " (" + r.destinations[i] + ")";
92          System.out.println( r.roomNumber + ") " + r.roomName + "\n" + exitString );
93       }
94    }
95
96    public static Room getRoom( Scanner f )
97    {
98       // any rooms left in the file?
```

```
 99        if ( ! f.hasNextInt() )
100            return null;
101
 102        Room r = new Room();
103        String line;
104
 105        // read in the room # for error-checking later
106        r.roomNumber = f.nextInt();
107        f.nextLine();   // skip "\n" after room #
108
 109        r.roomName = f.nextLine();
110
 111        // read in the room's description
112        r.description = "";
113        while ( true )
114        {
115            line = f.nextLine();
116            if ( line.equals("%%") )
117                break;
118            r.description += line + "\n";
119        }
120
 121        // finally, we'll read in the exits
122        int i = 0;
123        while ( true )
124        {
125            line = f.nextLine();
126            if ( line.equals("%%") )
127                break;
128            String[] parts = line.split(":");
129            r.exits[i] = parts[0];
130            r.destinations[i] = Integer.parseInt(parts[1]);
131            i++;
132        }
133        r.numExits = i;
134
 135        // should be done; return the Room
136        return r;
137    }
138
 139 }
```

What You Should See

```
This is the parlor.
It's a beautiful room.

There looks to be a kitchen to the "north".
And there's a shadowy corridor to the "east".
> north
There is a long countertop with dirty dishes everywhere. Off to one side
there is, as you'd expect, a refrigerator. You may open the "refrigerator"
or "go back".
> go back
This is the parlor.
It's a beautiful room.
```

```
There looks to be a kitchen to the "north".
And there's a shadowy corridor to the "east".
> east
The corridor has led to a dark room. The moment you step inside, the door
slams shut behind you. There is no handle on the interior of the door.

There is no escaping. Type "quit" to die.
> quit
```

Before I start talking about the code, let me take a moment to talk about the adventure game “file format”.

The game consists of several “rooms”. Each room has a room number and a room name; these are only used for the game engine and are never shown to the player.

Each room also has a description and one or more “exits”, which is a path to another room.

The adventure game file starts with a number: the total number of locations (rooms) in the game. After that are records for each room. Here’s an example:

```
1
KITCHEN
There is a long countertop with dirty dishes everywhere. Off to one side
there is, as you'd expect, a refrigerator. You may open the "refrigerator"
or "go back".
%%
fridge:3
refrigerator:3
go back:0
back:0
%%
```

The first line of this record is the room number, so this is room number 1. The second line of the record is the room name, which we only use for debugging.

Starting with the third line of the record is the description of the room, which continues until there is a line with nothing but %% on it. Blank lines *are* allowed in the description.

After the first double-percent there is a list of exits. Each line has the name of the exit (what the player will type to take that route) followed by a colon, followed by the room number where that exit leads.

For example, in this room if the player types "fridge" then the game engine will move them from this room (room #1) into room #3. And if they type "go back" then they will “travel” to room #0 instead. You may notice that in order to make it easier for the player to decide what to type I have duplicate exits in the list. Either the word "fridge" or "refrigerator" will take them to room #3.

The list of exits ends with another line containing only %%. And that’s the end of the record.

Okay, now let’s turn to the code. Lines 3 through 11 declare the record for one room. You can see we have fields for everything in the adventure game file. The only thing you might not have guessed is that the array of exit Strings (*exits*)

and the array of destination room numbers (*destinations*) have an arbitrary capacity of 10 and then there's a *numExits* field to keep track of how many exits there actually are in this room. Feel free to make this capacity larger if you think you'll need more than 10 exits in a room.

Moving into main(), line 20 declares the array of rooms and initializes it from the loadRoomsFromFile() function that I will explain later.

Line 22 has a commented-out call to a showAllRooms() function that I use for debugging.

On line 25 you will see the definition of our *currentRoom* variable, which holds the room number of the room the player is inside. They start in room 0, which is the first room in the file. And on line 26 is the declaration of the String *ans*, which will hold whatever the player types.

Line 27 is the beginning of the main game loop. It repeats as long as the *currentRoom* variable is 0 or more. So we will use this to stop the game: when the player dies (or wins) we will set *currentRoom* equal to -1.

The array *rooms* contains a list of all the locations in the game. The number of the room containing the player is stored in the variable *currentRoom*. So rooms[currentRoom] is the entire record for the… um, current room. In line 29 we store a copy of this room into the Room variable *cur*. (I only do this because I'm lazy and want to type things like cur.description instead of rooms[currentRoom].description.)

Speaking of which, line 30 prints out the description of the current room, which is stored in the *description* field.

On lines 31 and 32 we print out a little prompt and let the player enter in a String for where they want to go.

Lines 36 through 44 search through this room's array of exits looking to see if any of them match what the player typed. Remember that the *exits* array has a capacity of 10, but there are probably not that many exits actually present in this room. So in the for loop we count up to the value of the *numExits* field instead of 10.

If we find an exit that matches the player's command, we set our flag to true (so we know if we should complain if they end up typing something that's not in our list). Then since the words in the *exits* array line up with the room numbers in the *destinations* array, we pull the room number out of the corresponding slot of the *destinations* array and make that our new room number. This way, when the main game loop repeats again, we will have automatically traveled to the new room.

On line 45 we check our flag. If it's still false, it means the human typed something we never found in the list of exits. We can politely complain. Because *currentRoom* hasn't changed, looping around again in the main game loop will just print out the description again for the room they were already in.

And that's the end of the main game loop and the end of main(). All that's left is to actually fill up the array of rooms from the adventure game file.

Line 51 is the beginning of the loadRoomsFromFile() function, which takes the filename to open as a parameter and returns an array of Rooms.

(I decided that I didn't want to have throws Exception anywhere in this file, so there's a try-catch block here. It opens the file.

If we make it down to line 64 it means the file was opened successfully. We read in the first line of the file to tell us how many rooms there are. Then line 65 defines an array of Room records with the appropriate capacity.

On line 68 I made a variable called *roomNum*, which has a dual purpose. First of all: it is the index for the next available slot in the room array. But secondly, it is used to double-check that the room number (from the file) and the slot number of the room are the same. If not, there's probably some sort-of error in the game's data file. If we detect such an error (on line 72), we complain and end the program. (System.exit() ends the program, even from inside a function call.)

Line 69 is the beginning of the "read all rooms" loop. It keeps going as long as there is stuff in the file we haven't seen yet. There's a potential error here: if the number of rooms at the top of your data file is a lie, then this loop could go too far in the array and blow up. (For example, if the first line of the file says you only have 7 rooms but then you have 8 room records then this loop will repeat too many times.)

On line 71 we get read a single room record using the getRoom() function I'll explain later.

Lines 72 through 76 are the room number sanity check I already mentioned, and then line 77 just stores this new room into the next available slot in the rooms array. And line 78 increments the room index.

After that loop is over, all the rooms have been read in from the file and stored each into their own slot of the array. So on line 82 we can return the array of rooms back up to line 20 of main().

Lines 85 through 94 are the showAllRooms() function that I use for debugging. It just displays all the rooms in the array on the screen, and for each room it also shows all the exits and where they lead.

Our final function is getRoom(), which expects a Scanner object to be passed in as a parameter and which returns a single Room object.

On lines 99 and 100 there is a simple sanity check in case there is a malformatted data file. If the next thing in the file is *not* an integer, then just return null (the value of an uninitialized object). Putting a return up here will return from the function *right away* without bothering to run any of the remaining code.

On line 102 the empty room object is defined. Line 103 creates a String called *line*, which I use for a couple of different things.

Line 106 reads in the room number from the file. The room number is the first part of the room record. The rest of this function is going to use only the Scanner object's nextLine() method, and a nextLine() after a nextInt() usually doesn't work because it reads only the end of the line after the integer that was just read.

So line 107 calls the nextLine() method but doesn't bother to store its return value anywhere because it doesn't read anything worth saving.

Line 109 reads in the room name from the file. We only use this for debugging.

On line 112 we start by setting this room's *description* field to an empty String. This is so we can add on to it without getting an error. (Just like we would set a "total" variable to 0 before adding to it in a loop.)

Okay. So I like writing infinite loops. Sue me. Line 113 is the beginning of an infinite loop. This is because we don't know how many lines are going to be in the room's description; it just goes however long until we see a line consisting of nothing but %%. There are other ways to do this, but I like the "write an infinite loop and then break out of it when you see what you're looking for" approach. Like I've said before, reasonable people disagree.

Once we're inside the "infinite" loop, we read a line of description into the *line* variable. Then, on line 116 we check to see if what we just read was %%. If so, we don't want to add it to the description so we break out of the loop. break is sort-of like the opposite of continue; continue skips back up to the condition of a loop and break just skips to the end and stops looping.

If we're still around to see line 118, it means that we read in a line of description and it *wasn't* %%. So we use += to add that line (and a \n) to the end of whatever was already in the *description* field. And the loop repeats. (No matter what.)

Eventually we hopefully hit a %% and the loop stops looping.

Line 122 defines *i*, which I use for the index of which slot in the *exits* and *destinations* arrays we're going to put something in next. And then starting on line 123 there's another infinite loop. I use a very similar approach to read in all the exits.

Line 125 reads in the whole line, which means that *line* contains something like "refrigerator:3". (If it's not something like that but is actually %%, lines 126 and 127 stop looping.)

So now we need to split this line into two parts. Fortunately for us, the String class has a built-in method called split().

line.split(":") searches through the String *line* and breaks it up every time it sees a : (colon). And then it returns an array of Strings. For example, if *line* contained thisXisXaXtest then line.split("X") would return an array containing {"this", "is", "a", "test" }. In our case there's only one colon in *line*, so it returns something like {"refrigerator", "3" }.

So, after line 128 parts[0] contains the exit word (like "refrigerator") and parts[1] contains a **String** for the destination room number (like "3"). This doesn't quite work for us, because we need the room number to be an integer, not a String.

Fortunately for us (again), Java's standard library comes to the rescue. There is a built-in function to convert a String to an integer: Integer.parseInt(). We use this on line 130.

Recall that *i* is the index of the slot in the exits array where we need to store the next value. So line 129 stores parts[0] (the name of the exit) into the appropriate slot of the exits array. And line 130 converts parts[1] (the room number to move to) from a String to an int and stores that in the same slot of

the destinations array. Then line 131 increments the exit index for the next go-around.

Eventually we will hit a %% and this loop, too, will stop looping. There is a potential bug here, however. The exits array only has ten slots. If the data file has a room with more than ten exits, this loop will just keep on going past the end of the array and blow up the program. So don't do that.

After the loop ends, then our index *i* will contain the true number of rooms that we read in. So we store that into the *numExits* field of the current room on line 133.

And that should be it. All the fields in the room have been given values, and we return this Room object to line 71 of the loadRoomsFromFile() function.

Study Drills

1. Write your own text adventure. If you think it turns out pretty good, email it to me!
74. Add a save-game feature, so that the player can type something to stop the game, and the game will store their current room number to a text file and then load it back up when the game begins again.