Learn Java the Hard Way

Graham Mitchell

Learn Java the Hard Way

Graham Mitchell

This book is for sale at http://leanpub.com/javahard

This version was published on 2015-11-30

Tweet This Book!

Please help Graham Mitchell by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I'm about to learn to code using "Learn Java the Hard Way"! #LJtHW

The suggested hashtag for this book is #LJtHW.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#LJtHW

Contents

CONTENTS

CONTENTS

Acknowledgements

Preface: Learning by Doing

I have been teaching beginners how to code for the better part of two decades. 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 have plenty of students who are exceptional but *most* of my students are regular kids with no experience and no particular aptitude for programming. This book is written for regular people like them.

Most programming books and tutorials online are written by people with great natural ability and very little experience with real beginners. Their books often cover *far* too much material *far* too quickly and overestimate what true beginners can understand.

If you have a lot of experience or extremely high aptitude, you can learn to code from almost any source. I sometimes read comments like "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 lots of 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

Introduction: Java

Java is not a language for beginners. I am convinced that most "beginner" Java books only work for 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 am writing a follow-up book that will cover Object-Oriented Programming and some of 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 (for, while, do-while)
- 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.

All the examples in this book will work in version 1.5 of Java or any newer version.

What You Will *Not* Learn

- Graphics
- Object-oriented programming
- How to make an Android app
- Specifics of different "versions" of Java
- Javascript

No graphics
> 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.

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

No Android
> 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.

No specific version
> I will not cover anything about the differences between Java SE 7 and Java SE 8, for example. If you care about the difference, then this book is not for you.

> I will also not cover anything that was only recently added to Java. This book is for learning the basics of programming and nothing has changed about the basics of Java in many years.

No Javascript
> "Javascript" is the name of a programming language and "Java" is also the name of a programming language. These two languages have nothing to do with each other. They are completely unrelated.

I hope to write more books after this one. My second book will cover 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 Use 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. Then watch the Study Drill videos (if you have them) to compare your solutions to mine. And by the end you will be able to code, at least a little.

License

Some chapters of this book are made available free to read online but you are not allowed to make copies for others without permission.

The materials provided for download may not be copied, scanned, or duplicated, or posted to a publicly accessible website, in whole or in part.

Educators who purchase this book and/or tutorial videos are given permission to utilize the curriculum solely for self-study or for one-to-one, face-to-face tutoring of a single student. Large-group teaching of this curriculum requires a site 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.

> ⚠️ 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, then you might need to go learn that first.
>
> I'll tell you all the commands to type, but if you're interested in more detail you might want to check out the first chapter of "Conquering the Command Line" by Mark Bates. His book is designed for users of a "real" command line that you get on a Linux or Mac OS X machine, but the commands will be similar if you are using PowerShell on Windows.
>
> Read Mark's book here: http://conqueringthecommandline.com/book/basics

You are going to need to do three things no matter what kind of system you have:

1. Install a decent text editor for writing code.
2. Figure out how to open a terminal window so we can type commands.
3. Install the JDK (Java Development Kit).

   And on Windows, you'll need to do a fourth thing:
4. Add the JDK to the system PATH.

(The JDK commands are automatically added to the PATH on Apple computers and on Linux computers.)

I have instructions below for Windows, then for the Mac OS, and finally for Linux. Skip down to the operating system you prefer.

Windows

Installing a Decent Text Editor (Notepad++)

1. Go to http://notepad-plus-plus.org/ with your web browser, download the latest version of the Notepad++ text editor, and install it. You do not need to be an administrator to do this.
2. Once Notepad++ is installed, I always run it and turn off Auto-Completion since it is bad for beginners. (It also annoys me personally.) Open the "Settings" menu and choose "Preferences". Then click on "Auto-Completion" about halfway down the list on the left-hand side. Finally uncheck the box next to "Enable auto-completion on each input" and then click the "Close" button.
3. Finally while Notepad++ is still running I **right**-click on the Notepad++ button down in the Windows taskbar area and then click "Pin this program to taskbar." This will make it easier to launch Notepad++ for future coding sessions.

Opening a Terminal Window (PowerShell)

1. Click the Start button to open the Start Menu. (On Windows 8 and newer, you can open the search box directly by pressing the Windows key + S.) Start typing "powershell" in the search box.
2. Choose "Windows PowerShell" from the list of results.
3. Right-click on the PowerShell button in the taskbar and choose "Pin this program to taskbar."
4. In the Powershell/Terminal window, type

```
javac -version
```

You will probably get an error in red text that says something like "The term 'javac' is not recognized as the name of a cmdlet...."

This just means that the JDK isn't installed and added to the PATH, which is what we expect at this point.

> If you are using a very old version of Windows, PowerShell might not be installed. You *can* do all of the exercises in this book using "Command Prompt" (cmd.exe) instead, but the navigation commands will be different and adding the JDK to the PATH will also be different.
>
> I recommend trying to get PowerShell installed if you can.

Installing the Java Development Kit (JDK)

1. Go to http://www.oracle.com/technetwork/java/javase/downloads/ with your web browser.
2. Click the big "Java" button on the left near the top to download the Java Platform (JDK) 8u66. Clicking this will take you to a different page titled "Java SE Development Kit 8 Downloads."
3. 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 for version **8u66** or any newer version.

   If you know for sure that you are running a 64-bit version of Windows, it is okay to download the "Windows x64" version of the JDK. If you're not sure, then you should download the "x86" (a.k.a. 32-bit) version, since that version will work on both 32-bit Windows and on 64-bit Windows.
4. Once downloaded, run `jdk-8u66-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\jdk1.8.0_66\` or something similar. Make a note of this location; you will need it soon.
5. Just keep clicking "Next" until everything is done. Unless you *really* know what you're doing it's probably best to just let the installer do what it wants.

Adding the JDK to the PATH

1. Now that 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 `jdk1.8.0_66` that has a folder called `bin` inside it. The folder name *must* have `jdk1.8` in it; `jre8` is not the same. Make sure there's a `bin` folder.
2. Once you have clicked your way inside the `bin` folder, you can left-click up in the folder location and it will change to something that looks like `C:\Program Files (x86)\Java\jdk1.8.0_66\bin`. You can write this down or highlight and right-click to copy it to the clipboard.
3. 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\jdk1.8.0_66\bin", "User")
```

Put it all on one line, though. That is:

Type or paste `[Environment]::SetEnvironmentVariable("Path", "$env:Path;`

Don't press ENTER yet. You can paste into PowerShell by right-clicking.

Then type or paste the folder location from above. If you installed the x86 (32-bit) version of JDK version 8u66, it should be

```
C:\Program Files (x86)\Java\jdk1.8.0_66\bin
```

(Still don't press ENTER.)

Then add `, "User")` at the end. Finally, press ENTER.

If you get an error then 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.

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 the PowerShell window the change you just made won't take effect**.

Making Sure the JDK is Installed Correctly

1. Launch PowerShell again.
2. Type `javac -version` at the prompt.

```
javac -version
```

You should see a response like `javac 1.8.0_66`.

1. Type `java -version` at the prompt.

```
java -version
```

You should see a response like `java version "1.8.0_66"`.

Make sure they both report the same version number! If not, you might have two different (incompatible) versions of Java installed and you will have trouble completing the exercises in this book.

If the version numbers don't match, go into the Control Panel and Add/Remove Programs. Remove all programs related to Java, the JDK, the JRE. Remove Eclipse if it is installed. Remove *everything* Java related and start again.

However, if both commands worked and you didn't get any errors and the version numbers did match, then congratulations! Getting all that to work is pretty hard and you probably have what it takes to finish the rest of the book!

Don't move on to the next exercise yet, though! We should get used to navigating using the command-line, since that's how you will be running the programs you write in this book.

Navigation in the Command-Line (PowerShell)

You should create a new folder to put all your code in. After finishing this book, you will have at least 100 new files and it will be better if they are all in one place.

> In the following text, I will use the word "folder" and the word "directory" interchangably. They mean the same thing. The word "directory" is much older, but ever since the 1980s graphical operating systems have used a picture of a folder to represent a directory, so that word is now used, too.

```
ls
```

Type `ls` then press ENTER. (That's an "L" as in "list".) This command will *list* the contents of the current folder/directory.

```
cd Documents
```

The `cd` command means "change directory" and it will move you *into* the "Documents" folder so that future commands will take effect there.

Notice that your prompt will change to show that you are now inside a new folder. For example, when I first open PowerShell on my Windows 7 machine, my prompt is

```
PS C:\Users\Mitchell>
```

Then once I change into the "Documents" directory the prompt changes to

```
PS C:\Users\Mitchell\Documents>
```

You should type `ls` again once you get in there to see the contents of your Documents directory.

> If you are using an older version of Windows, the folder might be called "My Documents" instead of "Documents". If so, you will need to put quotes around the folder name for the `cd` command to work, since the name of the folder contains a space:
> ```
> cd "My Documents"
> ```

```
mkdir javahard
```

`mkdir` means "make directory" and will create a new folder in the current location. Typing `ls` afterward should show you that the new directory is now there. You can call the folder something different than "javahard" if you want to. You will only need to create this folder once per computer.

⌨ `cd javahard`

Change into the `javahard` folder. Afterward, type `ls` and it should list nothing. (The directory is empty, after all.)

⌨ `cd ..`

This is how you use the `cd` command to *back out* one level. After you type it you will be back in just the "Documents" directory, and your prompt should have changed to reflect that.

Issue the command to get back into the "javahard" folder again.

Now, use the mouse to open the text editor you installed earlier. Type a sentence or something and then save the file as `test.txt`. Save it into the "javahard" folder you just created.

Go back to the terminal window and issue the `ls` command to see the file you just created.

> If you're feeling fancy, you won't have to use the mouse to switch back to the terminal; you can just press Alt + Tab on Windows or Linux or press Command + Tab on a Mac to switch applications.
>
> Press and hold the Alt key. Keep it pressed. Then press and release the Tab key a single time. While still holding the Alt key, press Tab several more times until your terminal window is selected, then let go of the Alt key to make the switch.
>
> If you just quickly press Alt+Tab and let go of both keys right away, it usually takes you back to the previous application. I do this a *lot*. When I'm in the text editor I press Alt+Tab to get back to the terminal, then when I'm done in the terminal I press Alt+Tab again to get back to my text editor. It's very fast once you get used to it.

You should skip down to the bottom of this chapter and read the "Warnings for Beginners", but otherwise you're done with the setup and you are ready to begin Exercise 01 on Windows! Nice job.

Mac OS X

I don't own an Apple computer, so I don't currently have a way to test these directions for myself. I have tried to explain things, but there might be some small errors.

If you use these directions, I would appreciate any emails about things that worked or didn't work on your computer.

Installing a Decent Text Editor (TextWrangler)

1. Go to http://www.barebones.com/products/textwrangler/ with your web browser. Download the Disk Image for TextWrangler version 5.0 or any newer version.
2. Run the disk image, then open the Appliciations Folder and drag the icon over to it as indicated. You may have to authenticate with the administrator username and password.
3. Once installed, launch TextWrangler and add it to the dock if that doesn't happen automatically.

Opening a Terminal Window (Terminal)

1. Minimize TextWrangler and switch to Finder. Using the search (Spotlight), start searching for "terminal". That will open a little bash terminal.
2. Put your Terminal in your dock as well.
3. In Terminal window, type

```
javac -version
```

You should probably get an error that tells you that "javac" is an unknown command. (Feel free to email me a screenshot of the error message so I can update this paragraph.)

This just means that the JDK isn't installed, which is what we expect at this point.

If you are using a very old version of Mac OS X, the javac command might not give you an error! It might just print a version number on the screen!

As long as it is version 1.5 or higher, you can do all of the exercises in this book.

Installing the Java Development Kit (JDK)

1. Go to http://www.oracle.com/technetwork/java/javase/downloads/ with your web browser.
2. Click the big "Java" button on the left near the top to download the Java Platform (JDK) 8u66. Clicking this will take you to a different page titled "Java SE Development Kit 8 Downloads."
3. On this page you will have to accept the license agreement and then choose the "Mac OS X x64" version in the middle of the list. Download the file for version **8u66** or any newer version.
4. Once downloaded, run `jdk-8u66-macosx-x64.dmg` to install it.
5. Just keep clicking "Next" until everything is done. Unless you *really* know what you're doing it's probably best to just let the installer do what it wants.

Adding the JDK to the PATH

You get to skip this part, because the JDK installer does this *for* you on Apple computers. You might need to close the terminal and open it again, though, for the change to take effect.

Making Sure the JDK is Installed Correctly

1. Launch Terminal again.
2. Type `javac -version` at the prompt.

⌨ `javac -version`

You should see a response like `javac 1.8.0_66`.

1. Type `java -version` at the prompt.

⌨ `java -version`

You should see a response like `java version "1.8.0_66"`.

Make sure they both report the same version number! If not, you might have two different (incompatible) versions of Java installed and you will have trouble completing the exercises in this book.

If the version numbers don't match, uninstall all programs related to Java, the JDK, the JRE. Remove Eclipse if it is installed. Remove *everything* Java related and start again.

However, if both commands worked and you didn't get any errors and the version numbers did match, then congratulations! Getting all that to work is pretty hard and you probably have what it takes to finish the rest of the book!

Don't move on to the next exercise yet, though! We should get used to navigating using the command-line, since that's how you will be running the programs you write in this book.

Navigation in the Command-Line (Terminal)

You should create a new folder to put all your code in. After finishing this book, you will have at least 100 new files and it will be better if they are all in one place.

> In the following text, I will use the word "folder" and the word "directory" interchangably. They mean the same thing. The word "directory" is much older, but ever since the 1980s graphical operating systems have used a picture of a folder to represent a directory, so that word is now used, too.

```
ls
```

Type `ls` then press ENTER. (That's an "L" as in "list".) This command will *list* the contents of the current folder/directory.

```
cd Documents
```

The `cd` command means "change directory" and it will move you *into* the "Documents" folder so that future commands will take effect there.

Notice that your prompt will change to show that you are now inside a new folder. For example, when I first open Terminal, my prompt is

```
localhost:~ mitchell$
```

Then once I change into the "Documents" directory the prompt changes to

```
localhost:Documents mitchell$
```

You should type `ls` again once you get in there to see the contents of your Documents directory. Now we are ready to create the folder.

```
mkdir javahard
```

mkdir means "make directory" and will create a new folder in the current location. Typing ls afterward should show you that the new directory is now there. You can call the folder something different than "javahard" if you want to. You will only need to create this folder once per computer.

⌨ `cd javahard`

Change into the `javahard` folder. Afterward, type `ls` and it should list nothing. (The directory is empty, after all.)

⌨ `cd ..`

This is how you use the `cd` command to *back out* one level. After you type it you will be back in just the "Documents" directory, and your prompt should have changed to reflect that.

Issue the command to get back into the "javahard" folder again.

Now, use the mouse to open the text editor you installed earlier. Type a sentence or something and then save the file as `test.txt`. Save it into the "javahard" folder you just created.

Go back to the terminal window and issue the `ls` command to see the file you just created.

> If you're feeling fancy, you won't have to use the mouse to switch back to the terminal; you can just press Command + Tab on a Mac or press Alt + Tab on Windows or Linux to switch applications.
>
> Press and hold the Command key. Keep it pressed. Then press and release the Tab key a single time. While still holding the Command key, press Tab several more times until your terminal window is selected, then let go of the Command key to make the switch.
>
> If you just quickly press Command+Tab and let go of both keys right away, it usually takes you back to the previous application. I do this a *lot*. When I'm in the text editor I press Command+Tab to get back to the terminal, then when I'm done in the terminal I press Command+Tab again to get back to my text editor. It's very fast once you get used to it.

You should skip down to the bottom of this chapter and read the "Warnings for Beginners", but otherwise you're done with the setup and you are ready to begin Exercise 01 on Mac OS X! Nice job.

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.

Installing a Decent Text Editor (gedit)

1. On Ubuntu, gedit is already installed by default. It's called "Text Editor". If you search for it in the Dash, you'll find it with "gedit" or "text".

   If it's not installed on your Linux distro, use your package manager to install it.
2. Make sure you can get to it easily by right-clicking on its icon in the Launcher bar and selecting "Lock to Launcher".
3. Run gedit so we can change some of the defaults to be better for programmers:
     A. In the menu bar, open the "Edit" menu then choose "Preferences".
     B. In the "View" tab, put a check mark next to "Display line numbers"
     C. Make sure there's *not* a check mark next to "Enable text wrapping"
     D. Switch to the "Editor" tab and change Tab width: to 4.
     E. Put a check mark next to "Enable automatic indentation"

Opening a Terminal Window (Terminal)

1. Minimize your text editor and search for "Terminal" in the Dash. Other Linux distributions may call it "GNOME Terminal", "Konsole" or "xterm". Any of these ought to work.
2. Lock the Terminal to the Launcher bar as well.
3. In Terminal window, type

```
javac -version
```

You should get an error message that says "The program 'javac' can be found in the following packages" followed by a list of packages.

This just means that the JDK isn't installed, which is what we expect at this point.

Installing the Java Development Kit (JDK)

1. One of the nice things about Linux is the package manager. You can manually install Oracle's "normal" version of Java if you want, but I always just use the OpenJDK release:

⌨ sudo apt-get install openjdk-8-jdk

That's pretty much it. Everything in this book works fine using OpenJDK. (In fact, I *use* Linux for most of my day-to-day work and the exercises in this book were actually *written* and *tested* using OpenJDK!)

If, however, you're determined to have to install something like Windows and Mac users have to, you can download it from Oracle at http://www.oracle.com/technetwork/java/javase/downloads/

You're on your own for installing it, though. Seriously. Just use the version provided by your package manager.

Adding the JDK to the PATH

You get to skip this part, because this is already done *for* you on Linux computers. You might need to close the terminal and open it again, though, for the change to take effect.

However, on my computer running any Java tool prints an annoying message to the terminal window:

```
Picked up JAVA_TOOL_OPTIONS: -javaagent:/usr/share/java/jayatanaag.jar
```

This is because Eclipse doesn't work right without this JAR file. But we aren't going to be using Eclipse, and this message annoys me, so you need to add a line to the end of a hidden file called `.profile`. (The filename starts with a dot/period, which is why it's hidden.)

1. Launch your text editor. Click "Open".
2. Make sure you're in the "Home" directory.
3. Right-click anywhere in the "Open" window and put a checkmark next to "Show Hidden Files".
4. Open the file called `.profile`.
5. Add the following line at the bottom of the file:

```
unset JAVA_TOOL_OPTIONS
```

Save the file and close it. You might want to click "Open" again and remove the checkmark next to "Show Hidden Files".

Making Sure the JDK is Installed Correctly

1. Launch Terminal again.
2. Type `javac -version` at the prompt.

`javac -version`

You should see a response like `javac 1.8.0_45-internal`.

1. Type `java -version` at the prompt.

`java -version`

You should see a response like `openjdk version "1.8.0_45-internal"`.

Make sure they both report the same version number! If not, you might have two different (incompatible) versions of Java installed and you will have trouble completing the exercises in this book.

If the version numbers don't match, uninstall all programs related to Java, the JDK, the JRE. Remove Eclipse if it is installed. Remove *everything* Java related and start again.

However, if both commands worked and you didn't get any errors and the version numbers did match, then congratulations! Getting all that to work is pretty hard and you probably have what it takes to finish the rest of the book!

Don't move on to the next exercise yet, though! We should get used to navigating using the command-line, since that's how you will be running the programs you write in this book.

Navigation in the Command-Line (Terminal)

You should create a new folder to put all your code in. After finishing this book, you will have at least 100 new files and it will be better if they are all in one place.

> In the following text, I will use the word "folder" and the word "directory" interchangably. They mean the same thing. The word "directory" is much older, but ever since the 1980s graphical operating systems have used a picture of a folder to represent a directory, so that word is now used, too.

⌨ `ls`

Type `ls` then press ENTER. (That's an "L" as in "list".) This command will *list* the contents of the current folder/directory.

⌨ `cd Documents`

The `cd` command means "change directory" and it will move you *into* the "Documents" folder so that future commands will take effect there.

Notice that your prompt will change to show that you are now inside a new folder. For example, when I first open Terminal, my prompt is

`mitchell@localhost:∼$`

Then once I change into the "Documents" directory the prompt changes to

`mitchell@localhost:∼/Documents$`

You should type `ls` again once you get in there to see the contents of your Documents directory. Now we are ready to create the folder.

⌨ `mkdir javahard`

`mkdir` means "make directory" and will create a new folder in the current location. Typing `ls` afterward should show you that the new directory is now there. You can call the folder something different than "javahard" if you want to. You will only need to create this folder once per computer.

⌨ `cd javahard`

Change into the `javahard` folder. Afterward, type `ls` and it should list nothing. (The directory is empty, after all.)

⌨ `cd ..`

This is how you use the `cd` command to *back out* one level. After you type it you will be back in just the "Documents" directory, and your prompt should have changed to reflect that.

Issue the command to get back into the "javahard" folder again.

Now, use the mouse to open the text editor you installed earlier. Type a sentence or something and then save the file as `test.txt`. Save it into the "javahard" folder you just created.

Go back to the terminal window and issue the `ls` command to see the file you just created.

> If you're feeling fancy, you won't have to use the mouse to switch back to the terminal; you can just press Alt + Tab on Windows or Linux or press Command + Tab on a Mac to switch applications.
>
> Press and hold the Alt key. Keep it pressed. Then press and release the Tab key a single time. While still holding the Alt key, press Tab several more times until your terminal window is selected, then let go of the Alt key to make the switch.
>
> If you just quickly press Alt+Tab and let go of both keys right away, it usually takes you back to the previous application. I do this a *lot*. When I'm in the text editor I press Alt+Tab to get back to the terminal, then when I'm done in the terminal I press Alt+Tab again to get back to my text editor. It's very fast once you get used to it.

You should read the "Warnings for Beginners" below, but otherwise you're done with the setup and you are ready to begin Exercise 01 on Linux! Nice job.

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, he'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 the Java Development Kit.

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.

❓ Common Student 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.

FirstProg.java

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

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`, a single space, the word `FirstProg`, a space, and then a character called a "brace". You type a brace by holding down SHIFT and then pressing the '[' key, which is usually to the right of the letter 'P'.

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 three spaces.

Before I go on to the second 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 2 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 2.

{ 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 3 and another at the end of line 4.

: 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 line correctly.

You should start the second 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, a close parenthesis, one last space, and a second open curly brace.

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

The third line should start with *two* tabs. Your text editor may have already started your cursor directly underneath the 'p' in "public". If so, you only have to press TAB once. If not, press it twice to get two tabs.

After the tabs, 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 3 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 4 is nearly identical to line 3 except that the sentence says `Today's date is` instead of the determination sentence.

Line 5 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. The close curly brace should line up with the 'p' in `public static`, since that's the line where the matching open brace is.

Finally, line 6 has no tabs and one more closing curly brace. You can press ENTER after line 6 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. They will always pair up like this.

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 closing curly braces had fewer tabs.

Okay, now save this (if you haven't already) as `FirstProg.java` and save it in the "javahard" 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.

Then do a directory listing to make sure the Java file is there. On my computer, those commands look like this:

```
mitchell@localhost:~$ cd Documents
mitchell@localhost:~/Documents$ cd javahard
mitchell@localhost:~/Documents/javahard$ ls
FirstProg.java test.txt
mitchell@localhost:~/Documents/javahard$
```

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

```
cd Documents
cd javahard
ls
```

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.

Okay. We have typed a list of instructions 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"). It is properly pronounced "java-see", though many reasonable people say "jav-ack". (Those people are incorrect, but they are reasonable.)

Anyway, we use `javac` to compile our program 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 terminal will just display the prompt again without showing anything else.

However, if you made some sort of mistake, you will see an error like this:

```
FirstProg.java:6: 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:

```
FirstProg.java:3: 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 3 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:

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

This time it is 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.

> 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 successfully created a bytecode file we can run it (or "execute" it) by running it through the Java Virtual Machine (JVM) program called `java`:

```
java FirstProg
```

What You Should See

```
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`, even though `FirstProg.class` is the name of the bytecode file being executed in the JVM.

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 4 to include today's date. Save the file once you have made your changes, compile the file and run it again.
2. Change what is inside the quotes on line 3 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 Sunday, November 22, 2015.
```

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 `GasolineReceipt.java`. Make sure to match what I have written exactly, including spacing, punctuation, and capitalization.

The little vertical bar ("|") that you see on line 4 is called the "pipe" character, and you can type it using Shift + backslash ("\"). Assuming you are using a normal US keyboard, the pipe/backslash key is located between the `Backspace` and `Enter` keys.

GasolineReceipt.java

```java
 1  public class GasolineReceipt {
 2      public static void main( String[] args ) {
 3          System.out.println( "+------------------------+" );
 4          System.out.println( "|                        |" );
 5          System.out.println( "|      CORNER STORE      |" );
 6          System.out.println( "|                        |" );
 7          System.out.println( "| 2015-03-29   04:38PM   |" );
 8          System.out.println( "|                        |" );
 9          System.out.println( "| Gallons:        10.870 |" );
10          System.out.println( "| Price/gallon: $ 2.089  |" );
11          System.out.println( "|                        |" );
12          System.out.println( "| Fuel total:  $ 22.71   |" );
13          System.out.println( "|                        |" );
14          System.out.println( "+------------------------+" );
15      }
16  }
```

Notice that the first line is the same as the previous assignment, except that the name of the class is now `GasolineReceipt` instead of `FirstProg`. Also note that the name of the file you're putting things into is `GasolineReceipt.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 line of this program is *exactly* the same as the previous assignment. There are no differences whatsoever.

Then, after the second open curly brace, there are twelve printing statements. They are all identical except for what is between the quotation marks.

Once everything is typed in and saved as `GasolineReceipt.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 GasolineReceipt.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:

```
java GasolineReceipt
```

And you should see output like this:

```
    +------------------------+
    |                        |
    |       CORNER STORE     |
    |                        |
    | 2015-03-29   04:38PM   |
    |                        |
    | Gallons:       10.870  |
    | Price/gallon: $ 2.089  |
    |                        |
    | Fuel total:  $ 22.71   |
    |                        |
    +------------------------+
```

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 receipt uses gasoline prices for Texas. Go back to your text editor and change the date and other details to better fit the area where *you* live. Then save it, compile it and run it again.

❓ Common Student Questions

Why doesn't my receipt 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.

Is that really all you pay for petrol in Texas? That's less than 0.51 Euros per liter!
Yep. That's one of the perks of living in an oil-refining state.

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.

PrintingChoices.java

```java
public class PrintingChoices {
    public static void main( String[] args ) {
        System.out.println( "Alpha" );
        System.out.println( "Bravo" );

        System.out.println( "Charlie" );
        System.out.println( "Delta" );
        System.out.println();

        System.out.print( "Echo" );
        System.out.print( "Foxtrot" );

        System.out.println( "Golf" );
        System.out.print( "Hotel" );
        System.out.println();
        System.out.println( "India" );

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

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 8) 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 5, 9, 12, and 17). 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 difficult 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 19, 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 19, 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 19. When you put

---

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

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 19. 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.

Study Drills

1. Add a printing statement at the end of the code to display the sentence "I am learning Java the Hard Way!" But break it up like line 19 so that you only have two or three words in each String and plus signs between. Make sure the spaces get included!

Footnotes:

[2]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 code demonstrates many of Java's escape sequences.
Call it `EscapeSequencesComments.java`.

EscapeSequencesComments.java

```
1   public class EscapeSequencesComments {
2       public static void main( String[] args ) {
3           // This exercise demonstrates escape sequences & comments (like these)!
4           System.out.print( "Learn\tJava\n\tthe\nHard\tWay\n\n" );
5           System.out.print( "\tLearn Java the \"Hard\" Way!\n" );
6           // System.out.frimp( "Learn Java the Hard Way" );
7           System.out.print( "Hello\n" ); // This line prints Hello.
8           System.out.print( "Jello\by\n" ); // This line prints Jelly.
9           /* The quick brown fox jumped over a lazy dog.
10             Quick wafting zephyrs vex bold Jim. */
11          System.out /* testing */ .println( "Hard to believe, eh?" );
12          System.out.println( "Surprised? /* abcde */ Or what did you expect?" );
13          System.out.println( "\\ // -=- \\ //" );
14          System.out.println( "\\\\ \\\\\\ \\\\\\\\" ); // it takes 2 to make 1
15          System.out.print( "I hope you understand \"escape sequences\" now.\n" );
16          // and comments. :)
17      }
18  }
```

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

```
    Learn        Java
            the
    Hard        Way


            Learn Java the "Hard" Way!
    Hello
    Jelly
    Hard to believe, eh?
    Surprised? /* abcde */ Or what did you expect?
    \ // -=- \ //
    \\ \\\ \\\\
    I hope you understand "escape sequences" now.
```

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 somewhere 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. In most terminals, a tab will move the cursor enough to get to the next multiple of 8 columns.

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. Personally, I never ever press the Tab key inside quotation marks.

\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 3 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, as shown in lines 7 and 8, 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.)

Line 8 does something funny. `\b` is the escape sequence for "backspace", so it displays "Jello", then emits a backspace character. That deletes the "o", and then it displays a "y" (and then a `\n` to move to the next line).

Lines 9 and 10 are a block comment. Block comments begin with a `/*` (a slash then a star/asterisk) and end with an star and a slash (`*/`), whether they are on the same line or twenty lines later. Everything between is considered a comment and ignored by the compiler.

You can see a surprising example of a block comment in line 11. And on line 12 you can see that Strings (things in quotes) take precedence over block comments. Block comments are also sometimes called "C-style" comments, since the C programming language was the first one to use them.

Lines 13 and 14 demonstrate that to get a backslash to show up on the screen you must escape the backslash with another backslash. This matters when you're trying to deal with Windows-style paths; trying to open `"C:\Users\Graham_Mitchell\Desktop\foo.txt"` won't work in Java because the compiler will try to interpret "U" as something. You have to double the backslashes. (`"C:\\Users\Graham_Mitchell\Desktop\\foo.txt"`)

✏️ Study Drills

1. What happens if you put a block comment in the middle of the word `println`? Try it on line 13. Then add a comment below line 18 saying whether or not it compiles. Leave the block comment in line 13 if it works and take it out if it fails to compile.

2. Re-open the code for the *previous* exercise and save a copy as `PrintingChoicesEscapes.java`. Rewrite it so that it has identical-looking output but only uses a single println() statement with escape sequences.

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.
2. Variables can hold more than just numbers; they can hold words.
3. You have to choose what type of values the variable will hold when the variable is first created.
4. The value of a variable (but not its type) can change throughout the program. For example, 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       public static void main( String[] args ) {
3           int x, y, age;
4           double seconds, e, checking;
5           String firstName, lastName, title;
6
7           x = 10;
8           y = 400;
9           age = 39;
```

```
10
11              seconds = 4.71;
12              e = 2.71828182845904523536;
13              checking = 1.89;
14
15              firstName = "Graham";
16              lastName = "Mitchell";
17              title = "Mr.";
18
19          System.out.println( "The variable x contains " + x );
20          System.out.println( "The value " + y + " is stored in the variable y." );
21          System.out.println( "The experiment took " + seconds + " seconds." );
22          System.out.println( "A favorite irrational # is Euler's number: " + e );
23          System.out.println( "Hopefully you have more than $" + checking + "!" );
24          System.out.println( "My name's " + title + " " + firstName + lastName );
25      }
26  }
```

What You Should See

```
The variable x contains 10
The value 400 is stored in the variable y.
The experiment took 4.71 seconds.
A favorite irrational # is Euler's number: 2.718281828459045
Hopefully you have more than $1.89!
My name's Mr. GrahamMitchell
```

On lines 3 through 5 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 4 we declare variables named *seconds*, *e*, and *checking*. These three variables are "floating-point", which is the type of variable that can hold a number that might have a fractional part. "Double" is short for "double-precision floating-point". In this book, I will use the terms "double" and

---

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

"floating-point" interchangably. I know that's kind of confusing, but I didn't invent the terminology; I just teach it.

A floating-point variable could hold the value 4.71. It could hold the value -8192. (It *may* have a fractional part but doesn't have to.) A double-precision floating-point variable can pretty much hold *any* value between $\pm$ 1.79769 $\times$ 10^{308} and 4.94065 $\times$ 10^{-324}

However, doubles have limited precision. Notice that on line 12 I store the value 2.71828182845904523536 into the variable named *e*, but when I print out that value on line 22, 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. (And single-precision floating-point variables (a.k.a. floats) have even *fewer* significant figures than doubles do. Which is why they aren't used much any more.)

The last type of variable we are going to look at in this exercise is the String. On line 5 we declare three String variables: *firstName*, *lastName* and *title*. String variables can hold words and phrases; the name is short for "string of characters".

On lines 7 through 9 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 11 through 13 give initial values to the three double variables, and lines 15 through 17 initialize the three String variables. Then lines 19 through 24 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.

✏ Study Drills

1. Add four more variables to the program: another integer, another double, and two Strings. Name them whatever you want. Give them values. Print them out.

Footnotes:
───────────────────────────
[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.

```java
public class MathOperations {
    public static void main( String[] args ) {
        int a, b, c, d, e, f, g;
        double x, y, z;
        String one, two, both;

        a = 10;
        b = 27;
        System.out.println( "a is " + a + ", b is " + b );

        c = a + b;
        System.out.println( "a+b is " + c );
        d = a - b;
        System.out.println( "a-b is " + d );
        e = a+b*3;
        System.out.println( "a+b*3 is " + e );
        f = b / 2;
        System.out.println( "b/2 is " + f );
        g = b % 10;
        System.out.println( "b%10 is " + g );

        x = 1.1;
        System.out.println( "\nx is " + x );
        y = x*x;
        System.out.println( "x*x is " + y );
        z = b / 2;
        System.out.println( "b/2 is " + z );
        System.out.println();

        one = "dog";
        two = "house";
        both = one + two;
        System.out.println( both );
```

```
34        }
35    }
```

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 floating-point value (in either order). With two Strings (like on line 32) 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 floating-point values, or one integer and one double (in either order).

An asterisk (*) is used to represent multiplication. You can also see on line 15 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 17) the result is also an integer and not floating-point.

The percent sign (%) is used to mean 'modulus', which is essentially the remainder left over after dividing. On line 19, *b* is divided by 10 and the remainder (7) is stored into the variable *g*.

Modular arithmetic is a fairly simple mathematical operation that just isn't often taught in public school or even introductory university math curriculum. Wikipedia's example is good enough: we do modular arithmetic every time we add times on a typical 12-hour clock. If it is 7 o'clock now, what time will it be in eight hours? Well, once we hit 12:00 we "wrap around", so it will be 3 o'clock. (8+7 = 15, 15-12 = 3)

---

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

```
    int x = (7+8) % 12; // x has been set to 3
```

Put another way, 15 divided by 12 is 1 with a remainder of 3.

Modular arithmetic is used more than you would think in programming, but I won't be using it too much in the book.

? 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 floating-point variables sometimes give you values that are *slightly* different than you'd expect.

✏ Study Drills

1. Add two new variables to the program (integers, doubles or one of each). Then add two new lines where you initialize those new variables using mathematical expressions. Make sure you use each mathematical operator at least once. Put all this down below all the existing code.

Footnotes:

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.

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

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

---

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

```
    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

```
    What city is the capital of France?
    Paris
    What is 6 multiplied by 7?
    42
    Enter a number between 0.0 and 1.0.
    2.3
    Is there anything else you would like to say?
    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 get information into our program from the keyboard or other places like files on the computer or from the Internet.

Lines 3 and 4 are hopefully boring. On line 5 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 8 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 11 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 14 lets the human type in something which the Scanner object will attempt to convert into a floating-point value, and line 17 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.

✏️ Study Drills

1. Type something different that makes the program blow up on the second question. What did you type? Put a comment at the bottom of the code saying something like "// The 2nd question blows up when I type [BLANK]."
2. Type something that makes the program blow up on the third question. What did you type? Put another comment at the bottom of the code explaining what value blew it up and why.

Footnotes:

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.

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

```
29        }
30  }
```

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 the first printing statement on line 11 is `print()` rather than `println()`. Because of this, the question is printed on the screen, but the cursor is left alone after printing. Thus the cursor is left blinking at the end of the line the question is on.

If you had used `println()`, the cursor would have been moved to the next line *before* the Scanner's `next()` method had a chance to run, and so the cursor would be blinking on the beginning of the line after the question 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 very old.
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 called *name*, one integer variable called *age*, and two floating-point variables named *weight* and *income*.

On line 12 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 12. This means that on line 14, we can display that value on the screen! That's pretty cool, if you ask me.

On line 15 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 17.

Line 19 reads in a double value and stores it into *weight,* and line 23 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 7 (we choose its name and type), but it is undefined (it doesn't have a value) until line 23. On line 23 *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 23, 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.

> In the next part I am going to attempt to explain "declaring" variables a bit better by comparing it to some other ways variables are handled in other languages. If your brain is full, you might want to skip down to the Study Drills to practice what you learned above and come back to this section another day.

More on Declaring Variables

Java is a statically-typed programming language. That means that the type of a variable (`int`, `double`, `String`, etc) is *static*: it can't be changed. Almost all older programming languages are statically-typed, and most languages with static typing also have what is called "manifest typing"; these languages make you declare what types your variables are going to be before you can use them. Java, C, and C# are all languages with manifest typing.

Once you've told the Java compiler that, say, *age* is going to hold an integer, it makes sure that you don't change your mind later and try to put a String in there.

This might seem a bit annoying, but when you start writing much longer programs, the type checker can help prevent certain kinds of errors. And that's nice. But don't feel too sorry for yourself! One of the oldest programming languages ever (Plankalkül) wasn't so nice. You had to write the type of a variable *every time you used it.* This would be *very* annoying. I have written some code to show what Java might be like if you had to add the type of a variable every time you used it. Don't bother typing it up; it won't compile since Java (thankfully) doesn't work like that.

RudeQuestionsTypesEverywhere.java

```java
public class RudeQuestionsTypesEverywhere {
    public static void main( String[] args ) {
        keyboard[Scanner] = new Scanner(System.in);

        System.out.print( "Hello. What is your name? " );
        name[String] = keyboard[Scanner].next();

        System.out.print( "Hi, " + name[String] + "! How old are you? " );
        age[int] = keyboard[Scanner].nextInt();

        System.out.println("So you're " + age[int] + ", eh? That's not so old.");
        System.out.print( "How much do you weigh, " + name[String] + "? " );
        weight[double] = keyboard[Scanner].nextDouble();

        System.out.println( weight[double] + "! Better keep that quiet!!" );
        System.out.print("Finally, what's your income, " + name[String] + "? " );
        income[double] = keyboard[Scanner].nextDouble();

        System.out.print( "Hopefully that is " + income[double] + " per hour" );
        System.out.println( " and not per year!" );
        System.out.print( "Well, thanks for answering my rude questions, " );
        System.out.println( name[String] + "." );
    }
}
```

Whew! Aren't you glad you don't have to do that every time you use a variable?!?

On the other hand, you are allowed to feel a *little* bit sorry for yourself that you even have to declare the types of variables at all. Many modern programming languages feature "type inference", where the compiler can *infer* the type of a variable by how you use it in the code. Python and Swift are some programming languages that can infer types. Java does *not* have type inference, but I have written up some code that shows what this exercise might look like if it did. Again, don't bother typing this one up; it won't compile.

RudeQuestionsImpliedTypes.java

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

For example, on line 6 a compiler for such a language would be able to infer that *name* is a variable and that it must for holding Strings since that's what you're putting into it.

Some programming languages do one step further and allow you to *change* the type of a variable from one part of the code to another. These languages are called "dynamically typed". Ruby, Javascript and Perl are popular dynamically-typed languages. Dynamic typing makes them more powerful, but it also makes certain types of errors possible that would be easily caught by a static type checker.

Not that any of that matters, because you're learning Java right now and not any of those other fine programming languages. So you're stuck with static typing and manifest typing and you'll get used to it soon enough. Is it too much to ask to just specify the type of a variable before you start putting values in there? I think you'll find it isn't too bad.

Maybe you should scroll back up to the top to read the code again for this exercise before attempting the Study Drills, if you didn't do them already.

Study Drills

1. Does the program blow up if you enter an integer value when it is expecting you to type a double? Put an answer in a comment at the bottom of the code, along with your guess why or why not.

2. Does the program blow up if you enter a numeric value (integer or double) when it is expecting a String? Put an answer in a comment at the bottom of the code, along with your guess why or why not.

3. Type something that makes the program blow up on every question possible, and add comments explaining what blew it up and why.

4. Add a new variable of your choosing. Add another question. (It doesn't have to be rude.) Let the human put an answer to your question into your new variable, and display the human's answer on the screen afterward.

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!

```java
import java.util.Scanner;

public class BMICalculator {
    public static void main( String[] args ) {
        Scanner keyboard = new Scanner(System.in);
        double m, kg, bmi;

        System.out.print( "Your height in m: " );
        m = keyboard.nextDouble();

        System.out.print( "Your weight in kg: " );
        kg = keyboard.nextDouble();

        bmi = kg / (m*m);

        System.out.println( "Your BMI is " + bmi );
    }
}
```

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 14 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 value is in the BMI variable, but for now this will have to do.

Today's exercise is hopefully pretty easy to understand, but the Study Drills are quite a bit tougher than usual this time. If you can get them done without help, then your understanding is probably pretty good.

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
   ```

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
```

Common Student Questions

I did the Study Drills correctly, but I'm getting a different BMI than you show in the sample output! What gives?
Well, if your "different" BMI is like `0.0336` instead of `23.6`, then you didn't do the Study Drills correctly at all. You can't just use inches instead of meters and expect the formula to still work. Write extra code to *convert* inches to meters if you want to use the same formula, or use a different formula.

Okay, now my BMI is only *slightly* different from yours. What happened?
If your BMI is something like `23.618` instead of `23.625289` then don't worry about it. The difference is probably how many significant figures you used in your conversions. No one cares about BMI past the first decimal place anyway, and it's not like people are putting in their weight accurate to six decimal places.

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.

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

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 10. 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 17 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 10. The only reason it does have a value is because I did something sneaky on line 8.

Normally we have been declaring variables up at the top of our programs and then initializing them later. But on line 8 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 8, just declared.

So then on line 14 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 8 until 13 the variable *price* contains the value `0`. When line 14 begins executing and while we are waiting for the human to type something, *price* still contains `0`. But by the time line 14 has completed, whatever the human typed has been stored into *price*, replacing the zero. Then from line 15 until the end of the program, the variable *price* contains the value `7.99` (or whatever was typed in).

So with this in mind, we can figure out what really happens on line 10. Line 10 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 10 is beginning, the value of *salesTax* is undefined. (*salesTax* is declared but not defined.) By the end of line 10, *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 17.

Line 11 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.

> (It actually *is* possible, of course, to package up a formula and attach a name to it. Then using that name in your code will cause the formula to run each time it is referenced. We call it "function definition" and a "function call", but it's too complicated to discuss right now. And it *isn't* happening in this program, in any case.)

✏️ Study Drills

1. Remove the `= 0` on line 8, so that *price* no longer gets defined on line 8, 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.)
2. Move the two lines of code that give values to *salesTax* and *total* so they occur after *price* has been given a real value. Confirm that the program now works as expected.
3. Now that these lines occur *after* the variable *price* has been properly given a real value, try removing the `= 0` on line 8 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, tempX;    // x is 10, tempX is undefined
tempX = x + 2;        // x is still 10, tempX is now 12
x = tempX;            // x has been changed to 12
```

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 y = 10;    // y is 10
y = 2 + y;     // y *becomes* (2 plus the current value of y)
```

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

```
int m = 10;
m = m + 2;
```

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

```java
public class VariableChangeShortcuts {
    public static void main( String[] args ) {
        int i, j, k;

        i = 5;
        j = 5;
        k = 5;
        System.out.println( "i: " + i + "\tj: " + j + "\tk: " + k );
        i =  i + 3;
        j =  j - 3;
        k =  k * 3;
        System.out.println( "i: " + i + "\tj: " + j + "\tk: " + k + "\n" );

        i = 5;
        j = 5;
        k = 5;
        System.out.println( "i: " + i + "\tj: " + j + "\tk: " + k );
        i += 3;
        j -= 3;
        k *= 3;
        System.out.println( "i: " + i + "\tj: " + j + "\tk: " + k + "\n" );

        i = j = k = 5;
        System.out.println( "i: " + i + "\tj: " + j + "\tk: " + k );
        i += 1;
        j -= 2;
        k *= 3;
        System.out.println( "i: " + i + "\tj: " + j + "\tk: " + k + "\n" );

        i = j = 5;
        System.out.println( "i: " + i + "\tj: " + j );
        i =+ 1; // Oops!
        j =- 2;
        System.out.println( "i: " + i + "\tj: " + j + "\n" );

        i = j = 5;
        System.out.println( "i: " + i + "\tj: " + j );
        i++;
        j--;
        System.out.println( "i: " + i + "\tj: " + j );
    }
```

```
42    }
```

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
i: 1    j: -2

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

Hopefully lines 1-17 are nice and boring. We create three variables, give them values, display them, change their values and print them again. Then starting on line 14 we give the variables the same values they started with and print them.

On line 18 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 19 we see -= ("minus equals"), which subtracts 3 from *j*, and the next line 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 23 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.) This line 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 25 through 27 are basically the same as lines 18 through 20 except that we are no longer using 3 as the number to add, subtract or multiply.

Line 32 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 33 is similar: "Set *j* equal to negative 2." So watch for that.

On line 38 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.

And finally on the next line 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.

Especially i++. You will see that *all* the time.

✏ Study Drills

1. Add code at the bottom that resets *i*'s value to 5. Then, on the next line, use *only* -= to change *i*'s value to 0. Then on the line after that display *i*'s new value on the screen.
2. Add code below the other Study Drill that resets *i*'s value to 5, then using *only* ++, change *i*'s value to 10 and display it again. You may change the value using several lines of code or with just one line if you can figure it out.

(There is no video for these Study Drills yet.)

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 of characters, hold words, phrases, symbols, sentences, whatever

But as a wise man once said, "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!

```java
1   import java.util.Scanner;
2
3   public class BooleanExpressions {
4       public static void main( String[] args ) {
5           Scanner keyboard = new Scanner(System.in);
6           boolean a, b, c, d, e, f;
7           double x, y;
8
9           System.out.print( "Give me two numbers. First: " );
10          x = keyboard.nextDouble();
11          System.out.print( "Second: " );
12          y = keyboard.nextDouble();
13
14          a = (x <  y);
15          b = (x <= y);
16          c = (x == y);
17          d = (x != y);
18          e = (x >  y);
19          f = (x >= y);
20
21          System.out.println( x + " is LESS THAN " + y + ": " + a );
```

```
22          System.out.println( x + " is LESS THAN / EQUAL TO " + y + ": " + b );
23          System.out.println( x + " is EQUAL TO " + y + ": " + c );
24          System.out.println( x + " is NOT EQUAL TO " + y + ": " + d );
25          System.out.println( x + " is GREATER THAN " + y + ": " + e );
26          System.out.println( x + " is GREATER THAN / EQUAL TO " + y + ": " + f );
27          System.out.println();
28
29          System.out.println( !(x < y) + " " + (x >= y) );
30          System.out.println( !(x <= y) + " " + (x > y) );
31          System.out.println( !(x == y) + " " + (x != y) );
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      }
36  }
```

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 / 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 / EQUAL TO 4.0: false

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

On line 14 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*'s value is less than *y*'s, 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 15 is similar, except that the comparison is "less than or equal to", and the Boolean result is stored into *b*.

Line 16 is "equal to": *c* will be set to the value `true` if *x* holds the same value as *y*. The comparison in line 17 is "not equal to". Lines 18 and 19 are "greater than" and "greater than or equal to", respectively.

On lines 21 through 26, we display the values of all those Boolean variables on the screen.

Line 29 through line 34 introduce the "not" operator, which is an exclamation point (`!`). It takes the logical opposite. So on line 29 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 30 through 34 show the opposites of the remaining relational operators.

✎ Study Drills

1. Add comments at the bottom re-typing all six operators and their meanings. For example:

```
// less than is <
// greater than is >
```

(There is no video for this Study Drill yet.)

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.

```java
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:

```java
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.

```java
 1  import java.util.Scanner;
 2
 3  public class WeaselOrNot {
 4      public static void main( String[] args ) {
 5          Scanner keyboard = new Scanner(System.in);
 6          String word;
 7          boolean yep, nope;
 8
 9          System.out.println( "Type the word \"weasel\", please." );
10          word = keyboard.next();
11
```

```
12          yep  =   word.equals("weasel");
13          nope = ! word.equals("weasel");
14
15          System.out.println( "You typed what was requested: " + yep );
16          System.out.println( "You ignored polite instructions: " + nope );
17      }
18  }
```

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 12 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:

ShallowGrandmother.java

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

What You Should See

```
Enter your age: 40
Enter your yearly income: 49000
How cute are you, on a scale from 0.0 to 10.0? 7.5
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
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
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 when BOTH of its components are false.

✎ Study Drills

1. Many people have two grandmothers. Let's pretend the other grandmother just wants you to be happy. Add a question "How happy do you make them?" and a variable to hold its answer. Then add a Boolean variable called *allowed2* and write a new expression that's true when the person is close to your age and makes you happy (a happiness number more than 7 out of 10).

(There is no video for this Study Drill yet.)

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.

AgeMessages.java

```java
import java.util.Scanner;

public class AgeMessages {
    public static void main( String[] args ) {
        Scanner keyboard = new Scanner(System.in);
        int age;

        System.out.print( "How old are you? " );
        age = keyboard.nextInt();

        System.out.println( "You are: " );
        if ( age < 13 ) {
            System.out.println( "\ttoo young to create a Facebook account" );
        }
        if ( age < 16 ) {
            System.out.println( "\ttoo young to get a driver's license" );
        }
        if ( age < 18 ) {
            System.out.println( "\ttoo young to get a tattoo" );
        }
        if ( age < 21 ) {
            System.out.println( "\ttoo young to drink alcohol" );
        }
        if ( age < 35 ) {
            System.out.println( "\ttoo young to run for President of the U.S." );
            System.out.println( "\t\t(How sad!)" );
        }
    }
}
```

What You Should See

```
How old are you? 17
You are:
        too young to get a tattoo
        too young to drink alcohol
        too young to run for President of the U.S.
                (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 12 is false, and so the code in the body of the first `if` statement (lines 13 and 14) was skipped.

The second `if` statement was also false because 17 is not less than 16, so the code in its body (lines 16 and 17) 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 "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?
2. 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!".
3. 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 "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.

ComparingNumbers.java

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

What You Should See

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

On line 29 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 29 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?

2. 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.)

ClubBouncer.java

```java
public class ClubBouncer {
    public static void main( String[] args ) {
        int age = 22;
        boolean onGuestList = false;
        double allure = 7.5;
        String gender = "F";

        if ( onGuestList || age >= 21 || (gender.equals("F") && allure >= 8) ) {
            System.out.println("You are allowed to enter the club.");
        }
        else {
            System.out.println("You are not allowed to enter the club.");
        }
    }
}
```

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 lines 10 and 11, 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.

SecretWord.java

```java
import java.util.Scanner;

public class SecretWord {
    public static void main( String[] args ) {
        Scanner keyboard = new Scanner(System.in);
        String secret = "please", guess;

        System.out.print( "What's the secret word? " );
        guess = keyboard.next();

        if ( guess.equals(secret) ) {
            System.out.println( "That's correct!" );
        }
        else {
            System.out.println( "No, the secret word isn't \"" + guess + "\"." );
        }

        if ( guess == secret ) {
            System.out.println( "(This will never ever show, no matter what.)" );
        }
    }
}
```

What You Should See

```
What's the secret word? abracadabra
No, the secret word isn't "abracadabra".
```

Or, when you get it right:

```
    What's the secret word? please
    That's correct!
```

Notice that as usual I'm sneaking something *else* new into this exercise. On line 6 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 18 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.

✏️ Study Drills

1. Pick a second secret word that will also work. Then add an OR to the first if statement so that typing EITHER word makes it print out "That's correct!". For example, if I wanted the new secret word to be "mogul", I could change the if statement to look like this:

   `if ( guess.equals(secret) || guess.equals("mogul") ) {`

   Pick a different word for yours, though.

(There is no video for this Study Drill yet.)

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?

BMICategories.java

```java
import java.util.Scanner;

public class BMICategories {
    public static void main( String[] args ) {
        Scanner keyboard = new Scanner(System.in);
        double bmi;

        System.out.print( "Enter your BMI: " );
        bmi = keyboard.nextDouble();

        System.out.print( "BMI category: " );
        if ( bmi < 15.0 ) {
            System.out.println( "very severely underweight" );
        }
        else if ( bmi <= 16.0 ) {
            System.out.println( "severely underweight" );
        }
        else if ( bmi < 18.5 ) {
            System.out.println( "underweight" );
        }
        else if ( bmi < 25.0 ) {
            System.out.println( "normal weight" );
        }
        else if ( bmi < 30.0 ) {
            System.out.println( "overweight" );
        }
        else if ( bmi < 35.0 ) {
            System.out.println( "moderately obese" );
        }
```

```
30            else if ( bmi < 40.0 ) {
31                System.out.println( "severely obese" );
32            }
33            else {
34                System.out.println( "very severely/\"morbidly\" obese" );
35            }
36        }
37    }
```

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 a registered dietitian or 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 12 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 15 begins with an `else`. That else pays attention to the preceding `if` statement – the one on line 12 – 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 18 makes *that* `if` statement defer to the `if` statement that starts on line 15: if it's true, the line-18 `if` statement will skip *even if* it would have been true on its own. The `else` on line 21 makes its `if` statement defer to the previous `if` statement, and so on. And the `else` at the very end on line 33 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 21. Run the program, and enter `15.5` for the BMI. Do you see how that makes the `if` statement on line 21 "break rank" and no longer care about the `if` statements before it?
2. 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.

CollegeAdmission.java

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

public class CollegeAdmission {
    public static void main( String[] args ) {
        Scanner keyboard = new Scanner(System.in);
        int math;

        out.println( "Welcome to the UT Austin College Admissions Interface!" );
        out.print( "Please enter your SAT math score (200-800): " );
        math = keyboard.nextInt();

        out.print( "Admittance status: " );

        if ( math >= 790 )
            out.print( "CERTAIN " );
        else if ( math >= 710 )
            out.print( "SAFE " );
        else if ( math >= 580 )
            out.print( "PROBABLE " );
        else if ( math >= 500 )
            out.print( "UNCERTAIN " );
        else if ( math >= 390 )
            out.print( "UNLIKELY " );
        else // below 390
            out.print( "DENIED " );
```

```
27
28              out.println();
29          }
30  }
```

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 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.

CollegeAdmissionExpanded.java

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

public class CollegeAdmissionExpanded {
    public static void main( String[] args ) {
        Scanner keyboard = new Scanner(System.in);
        int math;

        out.println( "Welcome to the UT Austin College Admissions Interface!" );
        out.print( "Please enter your SAT math score (200-800): " );
        math = keyboard.nextInt();

        out.print( "Admittance status: " );

        if ( math >= 790 ) {
            out.print( "CERTAIN " );
        }
        else {
            if ( math >= 710 ) {
                out.print( "SAFE " );
```

```
21                }
22            else {
23                if ( math >= 580 ) {
24                    out.print( "PROBABLE " );
25                }
26                else {
27                    if ( math >= 500 ) {
28                        out.print( "UNCERTAIN " );
29                    }
30                    else {
31                        if ( math >= 390 ) {
32                            out.print( "UNLIKELY " );
33                        }
34                        else { // below 390
35                            out.print( "DENIED " );
36                        }
37                    }
38                }
39            }
40        }
41        out.println();
42    }
43 }
```

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.
2. Move lines 23 and 24 so they appear between lines 16 and 17. 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.
3. If you really don't trust me, 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.

GenderTitles.java

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

```
31                 else {
32                     title = "Mr.";
33                 }
34             }
35
36         System.out.println( "\n" + title + " " + last );
37     }
38 }
```

What You Should See

```
First name: Graham
Last name: Mitchell
Gender (M/F): M
Age: 40


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 18, 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 23 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 23 down through line 30 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 22 or on lines 31 through 38 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 18, 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 36, I need to make sure I declare it inside the block of code that ends on line 37.

I could have waited until line 16 to declare it, though.

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

Study Drills

1. Change the `else` on line 31 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 6 but is not given a value right away. Then on line 36, 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 31 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 a 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.

ThirtyDays.java

```java
import java.util.Scanner;

public class ThirtyDays {
    public static void main( String[] args ) {
        Scanner keyboard = new Scanner(System.in);
        int month, days;
        String monthName;

        System.out.print( "Which month? (1-12) " );
        month = keyboard.nextInt();

        switch(month) {
            case  1: monthName = "January";
                    break;
            case  2: monthName = "February";
                    break;
            case  3: monthName = "March";
                    break;
            case  4: monthName = "April";
                    break;
            case  5: monthName = "May";
                    break;
            case  6: monthName = "June";
                    break;
            case  7: monthName = "July";
                    break;
            case  8: monthName = "August";
                    break;
            case  9: monthName = "September";
                    break;
```

```
31              case 10: monthName = "October";
32                       break;
33              case 11: monthName = "November";
34                       break;
35              case 12: monthName = "December";
36                       break;
37              default: monthName = "error";
38          }
39
40          /* Thirty days hath September
41             April, June and November
42             All the rest have thirty-one
43             Except the second month alone....
44          */
45
46          switch(month) {
47              case  9:
48              case  4:
49              case  6:
50              case 11: days = 30;
51                       break;
52              case  2: days = 28;
53                       break;
54              default: days = 31;
55          }
56
57          System.out.println( days + " days hath " + monthName );
58      }
59 }
```

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 the 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 46 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, and yet another until it finally gets stopped by a break.

Exercise 23: More String Comparisons

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

DictionaryOrder.java

```java
import java.util.Scanner;

public class DictionaryOrder {
    public static void main( String[] args ) {
        Scanner keyboard = new Scanner(System.in);
        String name;

        System.out.print( "Make up the name of a programming language! " );
        name = keyboard.nextLine();

        if ( name.compareTo("c++") < 0 )
            System.out.println( name + " comes BEFORE c++" );
        if ( name.compareTo("c++") == 0 )
            System.out.println( "c++ isn't a made-up language!" );
        if ( name.compareTo("c++") > 0 )
            System.out.println( name + " comes AFTER  c++" );

        if ( name.compareTo("go") < 0 )
            System.out.println( name + " comes BEFORE go" );
        if ( name.compareTo("go") == 0 )
            System.out.println( "go isn't a made-up language!" );
        if ( name.compareTo("go") > 0 )
            System.out.println( name + " comes AFTER  go" );

        if ( name.compareTo("java") < 0 )
            System.out.println( name + " comes BEFORE java" );
        if ( name.compareTo("java") == 0 )
            System.out.println( "java isn't a made-up language!" );
        if ( name.compareTo("java") > 0 )
            System.out.println( name + " comes AFTER  java" );

        if ( name.compareTo("lisp") < 0 )
```

```
33                 System.out.println( name + " comes BEFORE lisp" );
34         if ( name.compareTo("lisp") == 0 )
35                 System.out.println( "lisp isn't a made-up language!" );
36         if ( name.compareTo("lisp") > 0 )
37                 System.out.println( name + " comes AFTER  lisp" );
38
39         if ( name.compareTo("python") < 0 )
40                 System.out.println( name + " comes BEFORE python" );
41         if ( name.compareTo("python") == 0 )
42                 System.out.println( "python isn't a made-up language!" );
43         if ( name.compareTo("python") > 0 )
44                 System.out.println( name + " comes AFTER  python" );
45
46         if ( name.compareTo("ruby") < 0 )
47                 System.out.println( name + " comes BEFORE ruby" );
48         if ( name.compareTo("ruby") == 0 )
49                 System.out.println( "ruby isn't a made-up language!" );
50         if ( name.compareTo("ruby") > 0 )
51                 System.out.println( name + " comes AFTER  ruby" );
52
53         if ( name.compareTo("visualbasic") < 0 )
54                 System.out.println( name + " comes BEFORE visualbasic" );
55         if ( name.compareTo("visualbasic") == 0 )
56                 System.out.println( "visualbasic isn't a made-up language!" );
57         if ( name.compareTo("visualbasic") > 0 )
58                 System.out.println( name + " comes AFTER  visualbasic" );
59     }
60 }
```

What You Should See

```
Make up the name of a 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 adding something new. On line 9 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:

```java
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:

```java
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

```java
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 technique of your choice, make this program work correctly even if the human types in words with different capitalization.

Footnotes:

---

[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 couple of 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.

RandomNumbers.java

```java
public class RandomNumbers {
    public static void main( String[] args ) {
        int a, b, c, d;
        double r, rps;

        rps = Math.random();
        if ( rps < 0.3333333 ) {  // will be true 1/3 of the time
            System.out.println( "ROCK" );
        }
        else if ( rps < 0.6666667 ) {
            System.out.println( "PAPER" );
        }
        else {
            System.out.println( "SCISSORS" );
        }

        // pick four random integers, each 1-10
        a = 1 + (int)( 10*Math.random() );
        b = 1 + (int)( 10*Math.random() );
        c = 1 + (int)( 10*Math.random() );
        d = 1 + (int)( 10*Math.random() );
        System.out.println( "1-10:\t" + a + "\t" + b + "\t" + c + "\t" + d );

        // pick four random integers, each 1-6
        a = 1 + (int)( 6*Math.random() );
        b = 1 + (int)( 6*Math.random() );
        c = 1 + (int)( 6*Math.random() );
        d = 1 + (int)( 6*Math.random() );
```

```
29          System.out.println( "1-6:\t" + a + "\t" + b + "\t" + c + "\t" + d );
30
31          // pick a single random integer, 1-100
32          a = 1 + (int)( 100*Math.random() );
33          System.out.println( "1-100:\t" + a + "\t" + a + "\t" + a + "\t" + a );
34
35          // pick four random integers, each 1-100
36          a = 1 + (int)( 100*Math.random() );
37          b = 1 + (int)( 100*Math.random() );
38          c = 1 + (int)( 100*Math.random() );
39          d = 1 + (int)( 100*Math.random() );
40          System.out.println( "1-100:\t" + a + "\t" + b + "\t" + c + "\t" + d );
41
42          // pick four random integers, each 0-99
43          a = 0 + (int)( 100*Math.random() );
44          b = 0 + (int)( 100*Math.random() );
45          c = (int)( 100*Math.random() );
46          d = (int)( 100*Math.random() );
47          System.out.println( "0-99:\t" + a + "\t" + b + "\t" + c + "\t" + d );
48
49          // pick four random integers, each 10-20
50          a = 10 + (int)( 11*Math.random() );
51          b = 10 + (int)( 11*Math.random() );
52          c = 10 + (int)( 11*Math.random() );
53          d = 10 + (int)( 11*Math.random() );
54          System.out.println( "10-20:\t" + a + "\t" + b + "\t" + c + "\t" + d );
55
56          // display four random doubles, each [0-1)
57          System.out.println( "0-1:\t" + Math.random() + "\t" + Math.random() );
58          System.out.println( "0-1:\t" + Math.random() + "\t" + Math.random() );
59
60          r = 10 * Math.random();
61          System.out.println( "0-9.99:\t" + r );
62          System.out.println( "0-9:\t" + (int)r );
63          System.out.println( "1-10:\t" + (1 + (int)r)  );
64      }
65 }
```

What You Should See

```
    SCISSORS
    1-10:   5       9       3       6
    1-6:    4       6       6       4
    1-100:  49      49      49      49
    1-100:  28      88      37      3
    0-99:   25      33      2       80
    10-20:  13      14      17      19
    0-1:    0.524564531320864       0.16490799299718129
    0-1:    0.8993174099533012      0.5885409176629621
    0-9.99: 1.4591530595519309
    0-9:    1
    1-10:   2
```

ⓘ *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 [0.0, 1.0) (that is, it might be exactly `0.0` but will never be exactly `1.0` and will most likely be something in between. So if I write:

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

...then *x* could have a value of `0.0`, or `0.123544`, or `0.3`, or `0.999999999` but never `1.0` and never greater than 1. So on line 6 the function `Math.random()` is called, and the result is stored into a variable named *rps*.

So, about one out of every three runs, *rps*' random value should be smaller than 0.3333333. If it is, we print "ROCK". If it's smaller than 0.6666667 (but not smaller than 0.3333333 because of the `else`) we print out "PAPER". And the rest of the time it'll print "SCISSORS".

If you could easily run this program 1000 times and tally up the results, you would find that each word should show up roughly 1/3 of the time.

So that's the idea. However, sometimes I don't want a random `double` between zero and one; I want a random *integer* between 1 and 10. Or between 1 and 100.

In most versions of Java we can't control the range of the value that `Math.random()` gives us, so we will have to do some funky math to transform that random value between 0.0 and 1.0 so that it ends up being in a different range.

> Alternatively, there is a class in Java you can import that can give you random numbers
> in different forms, but it's still not easy to get a single integer in a certain range, so this
> is the method I'm showing you for now.

On lines 18 through 21 we pick four new random numbers. We multiply each one of them by 10, then we cast them all to integers, then finally add 1 to each.

This sequence of math operations turns each number into something that's always from 1 to 10.

The next chunk of code picks four more random numbers and does the same steps but ends up with random integers from 1 to 6. (That's inclusive, so both 1 and 6 are possible.) Here are the steps:

- `Math.random()` generates a random `double` from `0.0` to `0.99999999999`
- `6*Math.random()` scales it to be from `0.0` to `5.999999999999`
- `(int)(6*Math.random())` casts it to an integer, which throws aways the part after the decimal point (truncates); now from `0` to `5`
- `1 + (int)(6*Math.random())` adds one, translating it to be from `1` to `6`

Line 32 makes the computer choose a random integer from `1` to `100` and puts it into the variable *a*. Notice that *a* only contains a copy of the *value* of the number, so printing it out more than once just displays the same random number over and over. Just printing the value of *a* doesn't magically make it pick again somehow.

The next chunk of code picks four random numbers from `1` to `100` and puts their values into four variables, then displays them.

The chunk of code starting on line 43 picks four more random numbers, but since we didn't add `1` to all of them, their range is `0` to `99`.

(It's hard to tell from the output, but trust me. If you ran this program 10,000 times line 47 would occasionally print out a `0` and just as occasionally print out a `99` but never `100` for any of the random numbers.)

Starting on line 50, there's a chunk of code that picks four random numbers from `10` to `20`. Surprised?

- `Math.random()` generates a random `double` from `0.0` to `0.99999999999`
- `11*Math.random()` makes it range from `0.0` to `10.999999999999`
- `(int)(11*Math.random())` casts it to an integer from `0` to `10`
- `10 + (int)(11*Math.random())` adds *10* so it's now from `10` to `20`

In general, the number you multiply it by determines *how many* random integers are in the range. Multiplying by 10 gives you ten possible numbers. Mutiplying by 6 gives you six possible numbers.

Then the number you add (after truncating/casting) is called the "origin" and is the *smallest* random you'll end up with after all the math is done.

There are eleven numbers from 10 to 20 (if you count both 10 and 20), so that's why we multiply by `11`. And we add ten because we want them to start at `10`.

So if I wanted a random number for a card game from 2-13:

- There are 12 numbers from 2 to 13 (inclusive), and
- The smallest number we want is 2.

So, `2 + (int)( 12*Math.random() )` will give us a random number from 2 to 13.

In fact,

```
int q = lo + (int)( (hi-lo+1) * Math.random() ); // picks int from lo to hi
```

The number of values from `lo` to `hi` is `(hi-lo+1)`; the +1 is to account for the fact that subtracting gives you the distance between two numbers, not the count of stopping points along the way.[8]

The rest of the code (from line 57 on) just shows some of the mathematical steps broken down one at a time. Study them a bit if you are still confused or skip them if you want.

✏️ Study Drills

1. Add extra code at the bottom to make the computer pick three random integers from 1 to 3 and print them out on the screen.
2. Add code to pick three random integers from 5 to 10 and display them. Run the program a bunch of times to confirm your numbers always stay in that range.

(There is no video for these Study Drills yet.)

Footnotes:

---

[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 25: 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.

We will get back to random numbers after a we spend a few exercises learning the basics of loops.

EnterPIN.java

```java
import java.util.Scanner;

public class EnterPIN {
    public static void main( String[] args ) {
        Scanner keyboard = new Scanner(System.in);
        int pin, entry;

        pin = 12345;

        System.out.println("WELCOME TO THE BANK OF JAVA.");
        System.out.print("ENTER YOUR PIN: ");
        entry = keyboard.nextInt();

        while ( entry != pin ) {
            System.out.println("\nINCORRECT PIN. TRY AGAIN.");
            System.out.print("ENTER YOUR PIN: ");
            entry = keyboard.nextInt();
        }

        System.out.println("\nPIN ACCEPTED. YOUR ACCOUNT BALANCE IS $425.17");
    }
}
```

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. YOUR ACCOUNT BALANCE IS $425.17
```

On line 14 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 is 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` statements 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.

✏ Study Drills

1. Before you ask the person for a PIN, ask them for a password. (It should be a String.) Then add a second `while` loop before the first one that loops as long as their answer is *not* the right password. So after you're done, the person must enter the correct password and *then* enter the correct PIN in order to see their account balance.

   Remember that you need to use `.equals()` when comparing Strings, and "not equals" looks like this:

   ```
   ( ! typedPassword.equals("hunter2") )
   ```

   The exclamation point in front there means "not" or "the opposite of".

(There is no video for this Study Drill yet.)

Exercise 26: 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?!?

We're also going to use a random number, which you learned how to do a couple of exercises back.

HighLow.java

```java
import java.util.Scanner;

public class HighLow {
    public static void main( String[] args ) {
        Scanner keyboard = new Scanner(System.in);
        int secret, guess;

        secret = 1 + (int)( 100*Math.random() );

        System.out.println( "I'm thinking of a number between 1-100." );
        System.out.println( "Try to guess it!" );
        System.out.print( "> " );
        guess = keyboard.nextInt();

        while ( secret != guess ) {
            if ( guess < secret ) {
                System.out.print( "Sorry, your guess is too low." );
                System.out.print( " Try again.\n> " );
                guess = keyboard.nextInt();
            }
            if ( guess > secret ) {
                System.out.print( "Sorry, your guess is too high." );
                System.out.print( " Try again.\n> " );
                guess = keyboard.nextInt();
            }
        }

        System.out.println( "You guessed it! What are the odds?!?" );
    }
}
```

What You Should See

```
I'm thinking of a number between 1-100.
Try to guess it!
> 50
Sorry, your guess is too low. Try again.
> 75
Sorry, your guess is too low. Try again.
> 87
Sorry, your guess is too high. Try again.
> 81
Sorry, your guess is too low. Try again.
> 84
Sorry, your guess is too low. Try again.
> 86
Sorry, your guess is too high. Try again.
> 85
You guessed it! What are the odds?!?
```

So on line 8 the computer chooses a random integer from 1 to 100 and stores it into the variable *secret*. We let the human make a guess.

Line 15 begins 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 16 through 25 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, 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.

Since the body of the loop has executed once all the way through, the program pops back up to line 15 and checks the condition again. If that condition is *still* true (their guess is still not equal to the secret number) then the whole loop body will execute a second time. If the condition is now false (because they guessed it) then the whole loop body will be skipped and the program will skip down to line 27.

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."

✏️ Study Drills

1. This code should be shorter! Notice that lines 18 and 19 are identical to lines 22 and 23. This is silly; there's no reason to write the same code in both `if` statements.

   Make it so there is only *one* copy of these two lines and put if *after* both `if` statements (but still inside the body of the `while` loop).

   Confirm that the program still works. When testing programs like this, I usually cheat by adding code to print out the value of the secret number before my first guess. Then, once I'm certain that everything works fine I take the code back out.

(There is no video for this Study Drill yet.)

Exercise 27: 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:

KeepGuessing.java

```java
import java.util.Scanner;

public class KeepGuessing {
    public static void main( String[] args ) {
        Scanner keyboard = new Scanner(System.in);
        int secret, guess;

        secret = 1 + (int)(Math.random()*10);

        System.out.println( "I have chosen a number between 1 and 10." );
        System.out.println( "Try to guess it." );
        System.out.print( "Your guess: " );
        guess = keyboard.nextInt();

        while ( secret != guess ) {
            System.out.println( "That is incorrect. Guess again." );
            System.out.print( "Your guess: " );
        }

        System.out.println( "That's right! You're a good guesser." );
    }
}
```

What You Should See

---

[9]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.

```
    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.
    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 didn't actually 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 15 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 19.

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 15 through 18 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 contains an infinite loop.

Footnotes:

Exercise 28: 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.

SafeSquareRoot.java

```java
import java.util.Scanner;

public class SafeSquareRoot {
    public static void main( String[] args ) {
        Scanner keyboard = new Scanner(System.in);
        double x, y;

        System.out.print("Give me a number, and I'll find it's square root. ");
        System.out.print("(No negatives, please.) ");
        x = keyboard.nextDouble();

        while ( x < 0 ) {
            System.out.println("I won't take the square root of a negative.");
            System.out.print("\nNew number: ");
            x = keyboard.nextDouble();
        }

        y = Math.sqrt(x);

        System.out.println("The square root of "+x+" is "+y);
    }
}
```

What You Should See

```
    Give me a number, and I'll find it's square root. (No negatives, please.) -8
    I won't take the square root of a negative.


    New number: -7
    I won't take the square root of a negative.


    New number: -200
    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 12 is an example of what I call an "input protection loop." On line 18 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 10 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 skillz aren't advanced enough to survive that, yet.)

So on line 12 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 12 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 17 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:

```java
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:

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

You might have noticed that on line 20 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 the Right Way to format your code 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 instead of tabs to indent your code, or always put the open brace of a code block at the beginning of the next line:

```java
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. Add another input protection loop toward the top of the program. Ask them "Are you ready?!?" and don't go on until they reply "YES!".

   Remember to use .equals() to compare Strings and use ! for "not".

(There is no video for this Study Drill yet.)

Exercise 29: 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.

CoinFlip.java

```java
1   import java.util.Scanner;
2
3   public class CoinFlip {
4       public static void main( String[] args ) {
5           Scanner keyboard = new Scanner(System.in);
6           String coin, again;
7           int streak = 0;
8           boolean gotHeads;
9
10          do {
11              gotHeads = Math.random() < 0.5;
12
13              if ( gotHeads )
14                  coin = "HEADS";
15              else
16                  coin = "TAILS";
17
18              System.out.println("You flip a coin and it is... " + coin);
19
20              if ( gotHeads ) {
21                  streak++;
22                  System.out.println("\tThat's " + streak + " in a row....");
```

```
23                    System.out.print("\tWould you like to flip again (y/n)? ");
24                    again = keyboard.next();
25                }
26              else {
27                    System.out.println("\tYou lose everything!");
28                    System.out.println("\tShould've quit while you were aHEAD....");
29                    streak = 0;
30                    again = "n";
31                }
32          } while ( again.equals("y") );
33
34          System.out.println( "Final score: " + streak );
35      }
36 }
```

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.)
2. `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 loop 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 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:

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

This is fine, and would work, but the variable *again* doesn't get a value until line 24. 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.

Before we get to the Study Drills, I should explain a bit more about how I "flipped a coin" in this code. Hopefully you remember that `Math.random()` returns a random `double` between 0 and 1. So the Boolean expression

```java
Math.random() < 0.5
```

...is true exactly 50% of the time. We store that truth value (true or false) into the variable `gotHeads` and then use `if` statements to test it later.

I need this result twice (lines 13 and 20), so I stored it into a variable. I understand that feels a little weird. If we had only needed it one time, it would have been easier to just write something like so:

```java
if ( Math.random() < 0.5 ) {
    // etc
```

Writing the gross expression to pick a random number between one and ten is fine if that's what you need, but if you just need something that is `true` 15% of the time, then:

```java
if ( Math.random() < 0.15 )
```

…is way easier.

✏️ 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.
2. 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 30: 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:

RunningTotal.java

```java
import java.util.Scanner;

public class RunningTotal {
    public static void main( String[] args ) {
        Scanner keyboard = new Scanner(System.in);
        int current, total = 0;

        System.out.print("Type in a bunch of values and I'll add them up. ");
        System.out.println("I'll stop when you type a zero.");

        do {
            System.out.print("Value: ");
            current = keyboard.nextInt();
            total += current;
            System.out.println("The total so far is: " + total);
        } while ( current != 0 );

        System.out.println("The final total is: " + total);
    }
}
```

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 6, we make sure to start by putting a zero into *total*. You'll see why soon.

On line 13 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 14 the magic happens. We add the value the human typed to *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 new number back into *total*. Thus *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 16 check to see if *current* was zero. If not, the loop repeats back up line 11.

The human gets to type in a second number. Let's say it is a 4. The variable *total* gets changed to *current* (4) plus *total*'s existing value (3), so *total* is now 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.

✎ Study Drills

1. Rewrite the code to use a `while` loop instead of a do-while loop. (It won't compile at first.) Add/change something before the loop starts so that it compiles and make sure it still works. Then change it back to a do-while. (And leave in whatever you changed to make the `while` version compile.)
2. When the human types in a zero to end the program, it prints "The total so far" and then prints "The final total" right afterward. That looks awkward, so fix the code so that it skips printing "The total so far" if the loop is about to end. Make sure it still prints it the rest of the time, though.

Exercise 31: 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[10] 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.

PigDiceComputer.java

```
1   public class PigDiceComputer {
2       public static void main( String[] args ) {
3           int roll, total;
4           total = 0;
5
6           do {
7               roll = 1 + (int)(Math.random()*6);
8               System.out.println( "Computer rolled a " + roll + "." );
9               if ( roll == 1 ) {
10                  System.out.println( "\tThat ends its turn." );
11                  total = 0;
12              }
13              else {
14                  total += roll;
15                  System.out.print( "\tComputer has " + total );
16                  System.out.print( " points so far this round.\n" );
17                  if ( total < 20 ) {
18                      System.out.println( "\tComputer will roll again." );
19                  }
20              }
```

---

[10]http://en.wikipedia.org/wiki/Pig_(dice)

```
21            } while ( roll != 1 && total < 20 );
22
23            System.out.println("Computer ends the round with " + total + " points.");
24        }
25  }
```

What You Should See

```
Computer rolled a 6.
        Computer has 6 points so far this round.
        Computer will roll again.
Computer rolled a 6.
        Computer has 12 points so far this round.
        Computer will 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 of 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 6). 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 7: a random number from 1-6 is a good substitute for rolling a dice.

On line 9 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 17 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 (7) in parentheses next to the roll value, since the die roll occurs on line 7 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 (11) there, since *total* would become zero on line 11 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 (4)
2 (7)	
	2 (14)
3 (7)	
	5 (14)
1 (7)	
	0 (11)

Exercise 32: 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[11] 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 37 through 59 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 *ctot* variable that holds the computer's overall points from round to round.

PigDice.java

```java
1   import java.util.Scanner;
2
3   public class PigDice {
4       public static void main( String[] args ) {
5           Scanner keyboard = new Scanner(System.in);
6           int roll, ptot, ctot, turnTotal;
7           String choice = "";
8
9           ptot = 0;
10          ctot = 0;
11
12          do {
13              turnTotal = 0;
14              System.out.println( "You have " + ptot + " points." );
15
16              do {
17                  roll = 1 + (int)(Math.random()*6);
18                  System.out.println( "\tYou rolled a " + roll + "." );
19                  if ( roll == 1 ) {
20                      System.out.println( "\tThat ends your turn." );
21                      turnTotal = 0;
22                  }
23                  else {
24                      turnTotal += roll;
25                      System.out.print( "\tYou have " + turnTotal + " points" );
```

---

[11]http://en.wikipedia.org/wiki/Pig_(dice)

```
26                    System.out.print( " so far this round.\n" );
27                    System.out.print( "\tWould you like to \"roll\" again" );
28                    System.out.print( " or \"hold\"? " );
29                    choice = keyboard.next();
30                }
31            } while ( roll != 1 && choice.equals("roll") );
32
33            ptot += turnTotal;
34            System.out.println("\tYou end the round with " + ptot + " points.");
35
36            if ( ptot < 100 ) {
37                turnTotal = 0;
38                System.out.println( "Computer has " + ctot + " points." );
39
40                do {
41                    roll = 1 + (int)(Math.random()*6);
42                    System.out.println( "\tComputer rolled a " + roll + "." );
43                    if ( roll == 1 ) {
44                        System.out.println( "\tThat ends its turn." );
45                        turnTotal = 0;
46                    }
47                    else {
48                        turnTotal += roll;
49                        System.out.print( "\tComputer has " + turnTotal );
50                        System.out.print( " points so far this round.\n" );
51                        if ( turnTotal < 20 ) {
52                            System.out.println( "\tComputer will roll again." );
53                        }
54                    }
55                } while ( roll != 1 && turnTotal < 20 );
56
57                ctot += turnTotal;
58                System.out.print( "\tComputer ends the round with " );
59                System.out.print( ctot + " points.\n" );
60            }
61
62        } while ( ptot < 100 && ctot < 100 );
63
64        if ( ptot > ctot ) {
65            System.out.println( "Humanity wins!" );
66        }
67        else {
```

```
68            System.out.println( "The computer wins." );
69        }
70    }
71 }
```

What You Should See

```
You have 0 points.
        You rolled a 6.
        You have 6 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 2.
        Computer has 2 points so far this round.
        Computer will roll again.
        Computer rolled a 5.
        Computer has 7 points so far this round.
        Computer will roll again.
        Computer rolled a 6.
        Computer has 13 points so far this round.
        Computer will roll again.
        Computer rolled a 6.
        Computer has 19 points so far this round.
        Computer will roll again.
        Computer rolled a 6.
        Computer has 25 points so far this round.
        Computer ends the round with 25 points.

    ...skip a bit, brother

You have 70 points.
        You rolled a 1.
        That ends your turn.
        You end the round with 70 points.
Computer has 85 points.
        Computer rolled a 2.
        Computer has 2 points so far this round.
        Computer will roll again.
        Computer rolled a 5.
```

```
                Computer has 7 points so far this round.
                Computer will roll again.
                Computer rolled a 6.
                Computer has 13 points so far this round.
                Computer will roll again.
                Computer rolled a 4.
                Computer has 17 points so far this round.
                Computer will roll again.
                Computer rolled a 4.
                Computer has 21 points so far this round.
                Computer ends the round with 106 points.
        The computer wins.
```

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

Then on line 12 begins a really huge do-while loop that basically contains the whole game and doesn't end until line 62. Scroll down and you can see that this loop repeats as long as both *ptot* and *ctot* 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 64) 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 13. 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 16 is the beginning of a do-while loop that contains the human's turn. It ends on line 31, and all the code between lines 16 and 31 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 18.

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 27-29 we give the human the choice to chance it by rolling again or play it safe by holding. Then on line 31 the condition of the do-while loop will check and repeat back up to line 16 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 37 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 (*ptot*) is greater than or equal to 100. This `if` statement begins here on line 36 and ends on line 60.

So on line 37 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 62 ends the do-while loop containing the whole game, and lines 64 through 69 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 "d" in `do` on line 40 to the matching close curly brace on line 55 to see what is inside that do-while loop and what isn't.

✏️ Study Drills

1. In the example output given, the computer's last roll was risky. The computer only needed to get 15 points in the round to win the game, and the computer *already* had earned 17 points in the round. So that last roll wasn't necessary, and the computer might have rolled a 1 and lost everything.

   Modify the code for the computer's turn so that it will stop after 20 points *or* when it has earned enough points to win the game.

(There is no video for this Study Drill yet.)

Exercise 33: 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"[12] in Java and how to make it execute by "calling" it.

ThereAndBackAgain.java

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

What You Should See

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

---

[12]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."

So lines 3 through 7 are pretty boring, except that on lines 4 and 6 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 10 through 12 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 10 through 12 define a *function* named `erebor()`. (The word `erebor()` doesn't mean anything in particular to Java. We could have named it `bilbo()` or `smaug()` or anything we like.)

This function has an open curly brace on line 10 just like `main()` always has an open curly brace. And on line 12 is the end of the function's body, and there's a close curly brace. So the function definition starts on line 10 and ends on line 12.

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 3 we print the String `"Here."` on the screen. Then on line 4 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 4 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 4, execution of the program pauses in `main()`, skips over all the rest of the code in `main()`, jumps down to line 10, 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 12, it returns back up to the end of line 4 and unpauses the execution in `main()`. It runs line 5 next.

On line 5 it displays another message on the screen, then on line 6 there is another function call. The function *erebor* is called a second time. It pauses `main()` on line 6, jumps down and runs through the body of *erebor* (which prints the String `"There."` again), then returns back up to line 6 where execution of `main()` resumes.

Finally line 7 prints one last String on the screen. Execution then proceeds to the close curly brace of `main()` which is on line 8. 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 4 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 6). 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.

Footnotes:

Exercise 34: 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 36 and 40 feature `print()` instead of `println()`.

OverlyComplexFlag.java

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

public class OverlyComplexFlag {
    public static void main( String[] args ) {
        printTopHalf();

        print48Colons();
        print48Ohs();
        print48Colons();
        print48Ohs();
        print48Colons();
        print48Ohs();
    }

    public static void print48Colons() {
        out.println( "|::::::::::::::::::::::::::::::::::::::::::::::::|" );
    }

    public static void print48Ohs() {
        out.println( "|OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO|" );
    }

    public static void print29Colons() {
        out.println( "|:::::::::::::::::::::::::::::|" );
    }

    public static void printPledge() {
        out.println( "I pledge allegiance to the flag." );
```

```
29          }
30
31      public static void print29Ohs() {
32          out.println( "|OOOOOOOOOOOOOOOOOOOOOOOOOOOOO|" );
33      }
34
35      public static void print6Stars() {
36          out.print( "| *  *  *  *  *  * " );
37      }
38
39      public static void print5Stars() {
40          out.print( "|   *  *  *  *  *  " );
41      }
42
43      public static void printSixStarLine() {
44          print6Stars();
45          print29Ohs();
46      }
47
48      public static void printFiveStarLine() {
49          print5Stars();
50          print29Colons();
51      }
52
53      public static void printTopHalf() {
54          out.println( " _____" );
55          // the line above has 1 space then 48 underscores between the quotes
56          printSixStarLine();
57          printFiveStarLine();
58          printSixStarLine();
59          printFiveStarLine();
60          printSixStarLine();
61          printFiveStarLine();
62          printSixStarLine();
63      }
64  }
```

What You Should See

```
 _____
| *   *   *   *   *   * |OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO|
|   *   *   *   *   *    |::::::::::::::::::::::::::::::::|
| *   *   *   *   *   * |OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO|
|   *   *   *   *   *    |::::::::::::::::::::::::::::::::|
| *   *   *   *   *   * |OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO|
|   *   *   *   *   *    |::::::::::::::::::::::::::::::::|
| *   *   *   *   *   * |OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO|
|::::::::::::::::::::::::::::::::::::::::::::::::|
|OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO|
|::::::::::::::::::::::::::::::::::::::::::::::::|
|OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO|
|::::::::::::::::::::::::::::::::::::::::::::::::|
|OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO|
```

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 5 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 53.

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 56, 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 44, 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 37, it returns control back to the second line (line 45) of printSixStarLine(), which then has another function call. This runs through the body of the function print29Ohs() and comes back to line 46. So then printSixStarLine() ends, which returns control to the end of line 56.

At this point, I think explaining all the function calls will be more confusing than just following the execution path on your own, which you will do in the Study Drill.

✏ Study Drills

1. This is mean, but I'm going to make you trace through the whole program. 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! If you watch the Study Drill video I trace through the entire thing with you.
2. On lines 27 through 29 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 35: 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.)

YachtDice.java

```java
1  public class YachtDice {
2      public static void main( String[] args ) {
3          int r1, r2, r3, r4, r5;
4          boolean allSame;
5
6          do {
7              r1 = 1 + (int)(Math.random()*6);
8              r2 = 1 + (int)(Math.random()*6);
9              r3 = 1 + (int)(Math.random()*6);
10             r4 = 1 + (int)(Math.random()*6);
11             r5 = 1 + (int)(Math.random()*6);
12             System.out.print("\nYou rolled: " + r1 + " " + r2 + " ");
13             System.out.println(r3 + " " + r4 + " " + r5);
14             showDice(r1);
15             showDice(r2);
16             showDice(r3);
17             showDice(r4);
18             showDice(r5);
19             allSame = ( r1 == r2 && r2 == r3 && r3 == r4 && r4 == r5 );
20
21         } while ( ! allSame );
22         System.out.println("The Yacht!!");
23     }
24
25     public static void showDice( int roll ) {
```

```
26            System.out.println("+---+");
27            if ( roll == 1 ) {
28                System.out.println("|   |");
29                System.out.println("| o |");
30                System.out.println("|   |");
31            }
32            else if ( roll == 2 ) {
33                System.out.println("|o  |");
34                System.out.println("|   |");
35                System.out.println("|  o|");
36            }
37            else if ( roll == 3 ) {
38                System.out.println("|o  |");
39                System.out.println("| o |");
40                System.out.println("|  o|");
41            }
42            else if ( roll == 4 ) {
43                System.out.println("|o o|");
44                System.out.println("|   |");
45                System.out.println("|o o|");
46            }
47            else if ( roll == 5 ) {
48                System.out.println("|o o|");
49                System.out.println("| o |");
50                System.out.println("|o o|");
51            }
52            else if ( roll == 6 ) {
53                System.out.println("|o o|");
54                System.out.println("|o o|");
55                System.out.println("|o o|");
56            }
57            System.out.println("+---+");
58        }
59 }
```

What You Should See

```
    You rolled: 4 3 5 2 6
    +---+
    |o o|
    |   |
    |o o|
    +---+
    +---+
    |o  |
    | o |
    |  o|
    +---+
    +---+
    |o o|
    | o |
    |o o|
    +---+
    +---+
    |o  |
    |   |
    |  o|
    +---+
    +---+
    |o o|
    |o o|
    |o o|
    +---+

    You rolled: 3 3 3 3 2
    +---+
    |o  |
    | o |
    |  o|
    +---+
    +---+
    |o  |
    | o |
    |  o|
    +---+
    +---+
    |o  |
    | o |
    |  o|
    +---+
    +---+
```

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

...etc

You rolled: 4 4 4 4 4
+---+
|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 19, the interesting thing in this exercise is a single function called showDice.

On lines 7 through 11 we choose five random numbers (each from 1 to 6) and store the results into the five integer variables *r1* through *r5*.

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 25 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
                        //       rather than 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(r2);           // YES
showDice(r5);           // YES
showDice( (r3+r4) / 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 26 the parameter variable *roll* has already been declared *and* initialized with the value 3.

If we call the function using a variable like `showDice(r2);` then the function is called and a copy of whatever value is currently in *r2* will have been stored into the parameter variable *roll* before the body of the function is executed.

So on line 14 the `showDice` function is executed, and *roll* will have been set equal to whatever value is in *r1*.

Then on line 15 `showDice` is called again, but this time *roll* will be set equal to whatever value is in *r2*. Line 16 calls `showDice` while setting its parameter equal to the value of *r3*. 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.

YachtDiceNoFunctions.java

```java
public class YachtDiceNoFunctions {
    public static void main( String[] args ) {
        int r1, r2;

        do {
            r1 = 1 + (int)(Math.random()*6);
            r2 = 1 + (int)(Math.random()*6);
            int total = r1 + r2;
            System.out.println("\nYou rolled a " + r1 + " and a " + r2);
            System.out.println("+---+");
            if ( r1 == 1 ) {
                System.out.println("|   |");
                System.out.println("| o |");
                System.out.println("|   |");
            }
            else if ( r1 == 2 ) {
                System.out.println("|o  |");
                System.out.println("|   |");
                System.out.println("|  o|");
            }
            else if ( r1 == 3 ) {
                System.out.println("|o  |");
                System.out.println("| o |");
                System.out.println("|  o|");
            }
            else if ( r1 == 4 ) {
                System.out.println("|o o|");
                System.out.println("|   |");
                System.out.println("|o o|");
            }
            else if ( r1 == 5 ) {
                System.out.println("|o o|");
                System.out.println("| o |");
                System.out.println("|o o|");
```

```
35                  }
36              else if ( r1 == 6 ) {
37                  System.out.println("|o o|");
38                  System.out.println("|o o|");
39                  System.out.println("|o o|");
40              }
41              System.out.println("+---+");
42
43
44              System.out.println("+---+");
45              if ( r2 == 1 ) {
46                  System.out.println("|   |");
47                  System.out.println("| o |");
48                  System.out.println("|   |");
49              }
50              else if ( r2 == 2 ) {
51                  System.out.println("|o  |");
52                  System.out.println("|   |");
53                  System.out.println("|  o|");
54              }
55              else if ( r2 == 3 ) {
56                  System.out.println("|o  |");
57                  System.out.println("| o |");
58                  System.out.println("|  o|");
59              }
60              else if ( r2 == 4 ) {
61                  System.out.println("|o o|");
62                  System.out.println("|   |");
63                  System.out.println("|o o|");
64              }
65              else if ( r2 == 5 ) {
66                  System.out.println("|o o|");
67                  System.out.println("| o |");
68                  System.out.println("|o o|");
69              }
70              else if ( r2 == 6 ) {
71                  System.out.println("|o o|");
72                  System.out.println("|o o|");
73                  System.out.println("|o o|");
74              }
75              System.out.println("+---+");
76
```

```
77                  System.out.println("The total is " + total + "\n");
78          } while ( r1 != r2 );
79
80          System.out.println("Doubles!  Nice job.");
81      }
82  }
```

What You Should See

```
You rolled a 1 and a 5
+---+
|   |
| o |
|   |
+---+
+---+
|o o|
| o |
|o o|
+---+
The total is 6


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


You rolled a 5 and a 6
+---+
|o o|
| o |
|o o|
```

```
+---+
+---+
|o o|
|o o|
|o o|
+---+
The total is 11


You rolled a 4 and a 4
+---+
|o o|
|   |
|o o|
+---+
+---+
|o o|
|   |
|o o|
+---+
The total is 8

Doubles!  Nice job.
```

✏️ Study Drills

1. Add a sixth dice. Notice how easy it is to display *r6* by just adding a single function call. (Don't forget to also add the sixth dice to the Boolean expression involving *allSame*.)

Exercise 36: 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 response: the area of that triangle using Heron's Formula.

HeronsFormula.java

```java
public class HeronsFormula {
    public static void main( String[] args ) {
        double a, g;
        String tws = "A triangle with sides ";

        a = triangleArea(3, 3, 3);
        System.out.println("A triangle with sides 3,3,3 has area " + a );

        a = triangleArea(3, 4, 5);
        System.out.println("A triangle with sides 3,4,5 has area " + a );
        g = triangleArea(7, 8, 9);
        System.out.println(tws + "7,8,9 has area " + g );

        System.out.println(tws + "5,12,13 has area " + triangleArea(5, 12, 13) );
        System.out.println(tws + "10,9,11 has area " + triangleArea(10, 9, 11) );
        System.out.println(tws + "8,15,17 has area " + triangleArea(8, 15, 17) );
    }

    // This function computes the area of a triangle with side lengths a, b, & c.
    public static double triangleArea( int a, int b, int c ) {
        double s, A;

        s = (a+b+c) / 2;
        A = Math.sqrt( s*(s-a)*(s-b)*(s-c) );

        // After computing the area, you must "return" the computed value:
        return A;
    }
}
```

What You Should See

```
A triangle with sides 3,3,3 has area 2.0
A triangle with sides 3,4,5 has area 6.0
A triangle with sides 7,8,9 has area 26.832815729997478
A triangle with sides 5,12,13 has area 30.0
A triangle with sides 10,9,11 has area 42.42640687119285
A triangle with sides 8,15,17 has area 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 20 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:

```java
// a function defined this way will return a value that is an int
public static int     triangleArea(int a, int b, int c)

// this one must return a value that is a String
public static String  triangleArea(int a, int b, int c)

// this function must return either the value true or the value false
public static boolean triangleArea(int a, int b, int c)

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

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 is like a function that does not return a value.

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

rollUpWindows("Toyota", "Corolla", 2008, "blue"); // if you 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 is like a function that returns a value.

```
String plate;

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

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

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:

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

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

This is usually a bad idea, but maybe you have your reasons. Anyway, back to the code.

On line 9 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 20). The 4 is stored into *b*, and 5 is put into *c*. It runs all the code on lines 23 and 24 with those values for the parameters. By the end, the variable *A* has a value stored in it.

On line 27 we *return* the value that is in the variable *A*.[13] This value travels back up to line 9, where it is stored into the variable *a*. Notice that both `main()` and `triangleArea()` have a variable called *a*, but these variables are different from each other. They have different types and different meanings, and they hold different values. This is not a problem.

Also notice that on line 11 we store the return value from the function into a variable named *g* instead of *a*. This is also fine. The `triangleArea()` function does not know or care about the name of the variable where its return value is stored. It doesn't even know if you store the return value or throw it away! It returned a value, and that's enough.

Okay, 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.

HeronsFormulaNoFunction.java

```java
public class HeronsFormulaNoFunction {
    public static void main( String[] args ) {
        int a, b, c;
        double s, A;

        a = 3;
        b = 3;
        c = 3;
        s = (a+b+c) / 2;
        A = Math.sqrt( s*(s-a)*(s-b)*(s-c) );
        System.out.println("A triangle with sides 3,3,3 has area " + A );

        a = 3;
        b = 4;
        c = 5;
        s = (a+b+c) / 2;
        A = Math.sqrt( s*(s-a)*(s-b)*(s-c) );
        System.out.println("A triangle with sides 3,4,5 has area " + A );

        a = 7;
        b = 8;
```

[13] (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`.)

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

Gross.

✏️ Study Drills

1. Which one is longer, the version that uses a function or the one without? Answer in a comment.
2. 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?
3. 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 area 3.897114317029974
A triangle with sides 3,4,5 has area 6.0
A triangle with sides 7,8,9 has area 26.832815729997478
A triangle with sides 5,12,13 has area 30.0
A triangle with sides 10,9,11 has area 42.42640687119285
A triangle with sides 8,15,17 has area 60.0
```

That's better.

❓ Common Student Questions

Why did you create a variable called `tws` and store words in it? Why not just put those words directly between the quotes in the `println()` statement?
It's a dumb reason: if I hadn't done that, lines 15-17 would have been too long for the print version of this book.

Footnotes:

Exercise 37: 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.

ShapeArea.java

```java
1   import java.util.Scanner;
2
3   public class ShapeArea {
4       public static void main( String[] args ) {
5           Scanner keyboard = new Scanner(System.in);
6           int choice;
7           double area = 0;
8
9           System.out.print("Shape Area Calculator version 0.1");
10          System.out.println(" (c) 2015 LJtHW Sample Output, Inc.");
11
12          do {
13              System.out.println("\n-=-=-=-=-=-=-=-=-=-=-=-=-\n");
14              System.out.println("1) Triangle");
15              System.out.println("2) Circle");
16              System.out.println("3) Rectangle");
17              System.out.println("4) Quit");
18              System.out.print("> ");
19              choice = keyboard.nextInt();
20
21              if ( choice == 1 ) {
22                  System.out.print("\nBase: ");
23                  int b = keyboard.nextInt();
24                  System.out.print("Height: ");
25                  int h = keyboard.nextInt();
26                  area = computeTriangleArea(b, h);
27                  System.out.println("The area is " + area);
28              }
29              else if ( choice == 2 ) {
30                  System.out.print("\nRadius: ");
31                  int radius = keyboard.nextInt();
32                  area = computeCircleArea(radius);
33                  System.out.println("The area is " + area);
```

```
34                    }
35                else if ( choice == 3 ) {
36                    System.out.print("\nLength: ");
37                    int q = keyboard.nextInt();
38                    System.out.print("Width: ");
39                    int w = keyboard.nextInt();
40                    System.out.println("The area is " + computeRectangleArea(q, w) );
41                }
42                else if ( choice != 4 ) {
43                    System.out.println("ERROR.");
44                }
45
46        } while ( choice != 4 );
47
48    }
49
50    public static double computeTriangleArea( int base, int height ) {
51        double A;
52        A = 0.5 * base * height;
53        return A;
54    }
55
56    public static double computeCircleArea( int radius ) {
57        double A;
58        A = Math.PI * radius * radius;
59        return A;
60    }
61
62    public static int computeRectangleArea( int length, int width ) {
63        return (length * width);
64    }
65 }
```

What You Should See

```
    Shape Area Calculator version 0.1 (c) 2015 LJtHW 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 50 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 51 we declare a variable named *A*. This variable is "local" to the function. Even though there is a variable named *A* declared on line 57, 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 23) is passed in as the initial value of the parameter *base* in the function call on line 26. *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 53 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 62.

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 62 and the arguments are the variables in the parentheses in the function call on line 40.) 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 63. 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 40, 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 38: 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[14], making it better with functions.

And, because I always like to cover something new, I have added special comments above the class and above each function called "Javadoc comments". You should type them in.

ThirtyDaysFunctions.java

```java
import java.util.Scanner;

/**
 * Contains functions that make it easier to work with months.
 */
public class ThirtyDaysFunctions {
    public static void main( String[] args ) {
        Scanner kb = new Scanner(System.in);

        System.out.print( "Which month? (1-12) " );
        int month = kb.nextInt();

        System.out.println(monthDays(month) + " days hath " + monthName(month));
    }

    /**
     * Returns the name for the given month number (1-12).
     *
     * @author Graham Mitchell
     * @param  month  the month number (1-12)
     * @return        the English name of the month, or "error" if out of range
     */
    public static String monthName( int month ) {
        String monthName = "error";

        if ( month == 1 )
            monthName = "January";
```

```
28          else if ( month == 2 )
29              monthName = "February";
30          else if ( month == 3 )
31              monthName = "March";
32          else if ( month == 4 )
33              monthName = "April";
34          else if ( month == 5 )
35              monthName = "May";
36          else if ( month == 6 )
37              monthName = "June";
38          else if ( month == 7 )
39              monthName = "July";
40          else if ( month == 8 )
41              monthName = "August";
42          else if ( month == 9 )
43              monthName = "September";
44          else if ( month == 10 )
45              monthName = "October";
46          else if ( month == 11 )
47              monthName = "November";
48          else if ( month == 12 )
49              monthName = "December";
50
51          return monthName;
52      }
53
54      /**
55       * Returns the number of days in a given month.
56       *
57       * @author Graham Mitchell
58       * @param  month  the month number (1-12)
59       * @return        the number of days in the month, or 31 if out of range
60       */
61      public static int monthDays( int month ) {
62          int days;
63
64          /* Thirty days hath September
65             April, June and November
66             All the rest have thirty-one
67             Except the second month alone.... */
68
69          switch(month)
```

```
70              {
71                  case  9:
72                  case  4:
73                  case  6:
74                  case 11: days = 30;
75                           break;
76                  case  2: days = 28;
77                           break;
78                  default: days = 31;
79              }
80
81          return days;
82      }
83  }
```

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 are 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 documentation 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[15]. 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!

---

[15]http://docs.oracle.com/javase/7/docs/api/index.html?java/util/Scanner.html

Exercise 39: 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 6.

PasswordDigest.java

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

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: hunter2
    F52FBD32B2B3B86FF88EF6C490628285F482AF15DDCB29541F94BCF526A3F6C7
```

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.
2. 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-16 representation of those bits. Since each hexadecimal (base-16) digit corresponds to 4 bits, we end up with an output 64 characters long.

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[16]
- javax.xml.bind.DatatypeConverter[17]

On lines 1 and 3 we import the two libraries we will be using to do the hard parts of this exercise.

On line 10 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 12 and 13 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 15 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[18]

On line 16 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!

---

[16] http://docs.oracle.com/javase/7/docs/api/java/security/MessageDigest.html

[17] http://docs.oracle.com/javase/7/docs/api/javax/xml/bind/DatatypeConverter.html

[18] http://docs.oracle.com/javase/7/docs/api/java/lang/String.html

✏️ 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.

2. Remove the `throws Exception` from the end of line 6. Try to compile it. (Then put it back.) You will learn a tiny bit about exceptions in the next exercise.

Exercise 40: 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 4. (And in this exercise, I'll actually explain what that means.)

ReceiptRevisited.java

```java
1   import java.io.PrintWriter;
2
3   public class ReceiptRevisited {
4       public static void main( String[] args ) throws Exception {
5           PrintWriter fileout = new PrintWriter("receipt.txt");
6
7           fileout.println( "+-----------------------+" );
8           fileout.println( "|                       |" );
9           fileout.println( "|      CORNER STORE     |" );
10          fileout.println( "|                       |" );
11          fileout.println( "| 2014-06-25   04:38PM  |" );
12          fileout.println( "|                       |" );
13          fileout.println( "| Gallons:       12.464 |" );
14          fileout.println( "| Price/gallon: $ 3.459 |" );
15          fileout.println( "|                       |" );
16          fileout.println( "| Fuel total:  $ 43.11  |" );
17          fileout.println( "|                       |" );
18          fileout.println( "+-----------------------+" );
19          fileout.close();
20      }
21  }
```
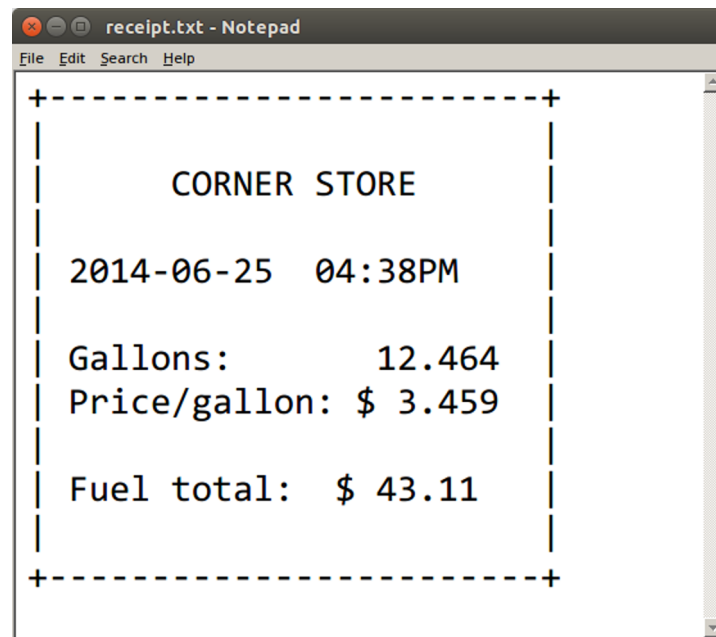
What You Should See

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 `receipt.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 Vista, it will look a little something like this:

```
+----------------------------+
|                            |
|        CORNER STORE        |
|                            |
|  2014-06-25   04:38PM      |
|                            |
|  Gallons:          12.464  |
|  Price/gallon: $ 3.459     |
|                            |
|  Fuel total:  $ 43.11      |
|                            |
+----------------------------+
```

a screenshot of the file "receipt.txt" opened in Notepad

On line 1 there is a new `import` statement for the Java class that will make this easy.

On line 5 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 this same line 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 String containing the desired output filename. (The name of the file to write to.)

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 7 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 `receipt.txt`!

If a file named `receipt.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 19. 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 `receipt.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 expression `new PrintWriter("receipt.txt")` because it tries to open a file for writing in the current folder. This could fail if there is already a file called "receipt.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:

ReceiptRevisitedException.java

```java
1  import java.io.IOException;
2  import java.io.PrintWriter;
3
4  public class ReceiptRevisitedException {
5      public static void main( String[] args ) {
6          PrintWriter fileout;
7
8          try {
9              fileout = new PrintWriter("receipt.txt");
10         }
11         catch ( IOException err ) {
12             System.out.println("Sorry, I can't open 'receipt.txt' for writing.");
13             System.out.println("Maybe the file exists and is read-only?");
14             fileout = null;
15             System.exit(1);
16         }
17
18         fileout.println( "+-----------------------+" );
19         fileout.println( "|                       |" );
20         fileout.println( "|     CORNER STORE      |" );
21         fileout.println( "|                       |" );
```

```
22          fileout.println( "| 2014-06-25  04:38PM     |" );
23          fileout.println( "|                         |" );
24          fileout.println( "| Gallons:       12.464   |" );
25          fileout.println( "| Price/gallon: $ 3.459   |" );
26          fileout.println( "|                         |" );
27          fileout.println( "| Fuel total:  $ 43.11    |" );
28          fileout.println( "|                         |" );
29          fileout.println( "+-----------------------+" );
30          fileout.close();
31      }
32  }
```

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.

✏ Study Drills

1. Hey, you know how to work with variables! Pick a price per gallon and put it into a double variable, then let the human enter in how many gallons of gas they want. Figure out the total cost.

   Then print the receipt to the file, but substitute their details so the file will look different depending on what they put in. I would remove the bars on the right side of the receipt, since making things line up would be hard.

(There is no video for this Study Drill yet.)

What You Should See After Doing the Study Drill

```
    Gas costs $ 1.959 per gallon.
    How many gallons do you want? 11.607
    Total cost: $ 22.738113


    Writing customized receipt to 'receipt.txt'... done.
```

Exercise 41: 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:
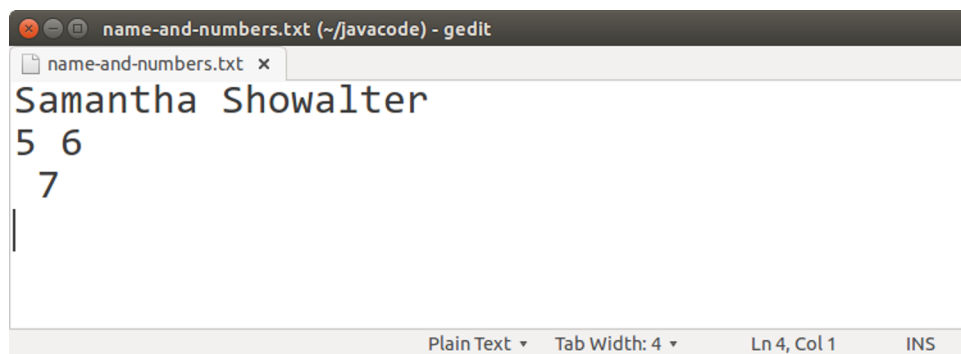
a screenshot of the file "name-and-numbers.txt" opened in Notepad

Okay, to the code!

GettingFromFile.java

```java
import java.io.File;
import java.util.Scanner;

public class GettingFromFile {
    public static void main( String[] args ) throws Exception {
        String name;
        int a, b, c, sum;

        System.out.print("Getting name and three numbers from file...");
        Scanner fileIn = new Scanner(new File("name-and-numbers.txt"));
            name = fileIn.nextLine();
            a = fileIn.nextInt();
            b = fileIn.nextInt();
            c = fileIn.nextInt();
```

```
15            fileIn.close();
16
17            System.out.println("done.");
18            System.out.println("Your name is " + name);
19            sum = a + b + c;
20            System.out.println( a + "+" + b + "+" + c + " = " + sum );
21        }
22    }
```

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 11 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 12 through 14 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 15 the file is closed, which means your Scanner object isn't connected to it anymore. If you ever write a program that uses the same file for both reading and writing, then forgetting to .close() the file when reading will make it so you can't write to it either.

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 42: Getting ALL the Data from a File

In the previous exercise, we knew the file's contents: one line with a number and then exactly three numbers. If there had been seven numbers in the file only the first three would have been used. And if there had been only *two* numbers in the file, the program would have ended with an error when run.

So we often use a loop to get *all* the values in a file, no matter how many or how few!

Start by creating new text file called some-words.txt in the same folder as the code. My file looks like this:

some-words.txt

```
The man in black fled across the desert, and the gunslinger followed.

Ask not what your country can do for you; ask what you can do for your country.

i will be
   M o ving in the Street of her

   bodyfee l inga ro undMe the traffic of
   lovely;muscles-sinke x p i r i n   g S
       udden1
   Y           totouch
```

The contents don't matter too much, though, which is sort-of the whole point! Once this file exists and has some words in it you should be able to type in and run the code below.

GettingWholeFile.java

```java
import java.io.File;
import java.util.Scanner;

public class GettingWholeFile {
    public static void main( String[] args ) throws Exception {
        int fourLetter = 0;
        int caps = 0;

```

```
 9          String fn1 = "some-words.txt";
10          String fn2 = "GettingWholeFile.java";
11
12          Scanner wordReadr = new Scanner(new File(fn1));
13
14          while ( wordReadr.hasNext() ) {
15              String w = wordReadr.next();
16              if ( w.length() == 4 ) {
17                  fourLetter++;
18              }
19          }
20          wordReadr.close();
21
22          Scanner selfInput = new Scanner(new File(fn2));
23          while ( selfInput.hasNext() ) {
24              String token = selfInput.next();
25              if ( Character.isUpperCase( token.charAt(0) ) ) {
26                  caps++;
27              }
28          }
29          selfInput.close();
30
31          System.out.println( fourLetter + " four-letter words in " + fn1 );
32          System.out.println( caps + " words start with capitals in " + fn2 );
33      }
34 }
```

What You Should See

```
9 four-letter words in some-words.txt
16 words start with capitals in GettingWholeFile.java
```

Just like before, we create a Scanner object and attach it to read from a file (line 12). This time, the file name is stored in a String variable first. Our Scanner is called *wordReadr* but we could have called it anything.

Line 14 is where the magic starts. .hasNext() returns true if the Scanner has stuff in it we haven't seen yet, and it returns false if the file is empty or if we've already read everything.

So while (wordReadr.hasNext()) will be true (and keep looping) as long as there is stuff in the file and will stop once we have seen everything.

Inside the body of the loop, we read a single String into a new local variable called *w*. And we add to a counter if *w* is four characters long.

Once the loop is over, we've read everything in the file so we `.close()` it.

Then, starting on line 22, we create a second Scanner object and attach it to read from a second file. (This file is the Java file you just wrote and compiled!)

Line 25 needs some explaining.

- `token.charAt(0)` is the first letter of the current word. For example, if *token* contains `"public"` then `token.charAt(0)` is `'p'`.
- `Character.isUpperCase( token.charAt(0) )` is `true` when the first letter is upper-case. `'F'` is upper-case. `'f'` is not.

So this second loop counts the number of "words" in `GettingWholeFile.java` that begin with a capital letter. Not bad, eh?

✏️ Study Drills

1. Add a comment at the end of the Java file that has some words in it, and make some of them begin with a capital letter. Then re-compile and re-run the program to confirm that they get counted.

(There is no video for this Study Drill yet.)

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.

CoinFlipSaved.java

```java
1  import java.io.File;
2  import java.io.PrintWriter;
3  import java.util.Scanner;
4
5  public class CoinFlipSaved {
6      public static void main( String[] args ) throws Exception {
7          Scanner keyboard = new Scanner(System.in);
8          String coin, again, bestName, saveFileName = "coin-flip-score.txt";
9          int streak = 0, best;
10         boolean gotHeads;
11
12         File f = new File(saveFileName);
13         if ( f.exists() && f.length() > 0 ) {
14             Scanner input = new Scanner(f);
15                 bestName = input.next();
16                 best = input.nextInt();
17             input.close();
18             System.out.print("High score is " + best);
19             System.out.println(" flips in a row by " + bestName);
20         }
21         else {
22             System.out.println("Save game file doesn't exist or is empty.");
23             best = -1;
24             bestName = "";
25         }
26
27         do {
28             gotHeads = Math.random() < 0.5;
29
30             if ( gotHeads )
```

```java
31                  coin = "HEADS";
32              else
33                  coin = "TAILS";
34
35          System.out.println( "You flip a coin and it is... " + coin );
36
37          if ( gotHeads ) {
38              streak++;
39              System.out.println( "\tThat's " + streak + " in a row...." );
40              System.out.print( "\tWould you like to flip again (y/n)? " );
41              again = keyboard.next();
42          }
43          else {
44              streak = 0;
45              again = "n";
46          }
47      } while ( again.equals("y") );
48
49      System.out.println( "Final score: " + streak );
50
51      if ( streak > best ) {
52          System.out.println("That's a new high score!");
53          System.out.print("Your name: ");
54          bestName = keyboard.next();
55          best = streak;
56      }
57      else if ( streak == best ) {
58          System.out.println("That ties the high score. Cool.");
59      }
60      else {
61          System.out.print("You'll have to do better than ");
62          System.out.println(streak + " if you want a high score.");
63      }
64
65      // Save this name and high score to the file.
66      PrintWriter out = new PrintWriter(f);
67          out.println(bestName);
68          out.println(best);
69      out.close();
70  }
71 }
```

What You Should See

```
    Save game file doesn't exist or is empty.
    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)? y
    You flip a coin and it is... TAILS
    Final score: 0
    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 12 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 13 there is an `if` statement, and in the condition I call the `exists()` method of the File object. This will check to see if the file exists. I also check to make sure the length of the file (in bytes) is more than zero. Sometimes students have bugs in their code that cause an empty file to be created, and this helps with that.

(Technically I could have *only* checked if `f.length()` is greater than zero because the `.length()` method also returns `0` when the file doesn't exist. But I've heard it said that a good programmer looks both ways before crossing a one-way street. So sometimes I like to be overly cautious.)

When the `if` statement is true, then, it means the save game file does exist and has *something* stored in it. (Hopefully a name and high score!) So, we use a Scanner object to get the existing name and high score out of the file.

If the file doesn't exist or doesn't have anything in it, we say so and put suitable initial values into the variables *best* and *bestName.* Cool, eh?

Lines 27 through 49 are the existing coin flip game. I didn't change any of this code at all.

On line 51 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 60 the `else` will run if they didn't beat or tie the high score. So we taunt them, of course.

On lines 66 through 69 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 there is a *new* winner. It currently rewrites the high score file from scratch *every time* the program runs, even if there are no changes.
2. "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 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.

CountingFor.java

```java
import java.util.Scanner;

public class CountingFor {
    public static void main( String[] args ) {
        Scanner keyboard = new Scanner(System.in);
        int n;
        String message;

        System.out.println( "Enter a message and I'll display it five times." );
        System.out.print( "Message: " );
        message = keyboard.nextLine();

        for ( n = 1 ; n <= 5 ; n++ ) {
            System.out.println( n + ". " + message );
        }

        System.out.println( "\nNow I'll show it ten times and count by 5s." );
        for ( n = 5 ; n <= 50 ; n += 5 ) {
            System.out.println( n + ". " + message );
        }

        System.out.println( "\nFinally, three times counting backward." );
        for ( n = 3 ; n > 0 ; n -= 1 ) {
            System.out.println( n + ". " + message );
        }
    }
}
```

What You Should See

```
    Enter 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 show 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 13 demonstrates a very basic `for` loop. Every `for` loop has three parts with semicolons between.

Most `for` loops pick a single variable and name that variable in all three parts.

1. The *initialization statement* tells the variable where to start.
2. The loop keeps going as long as the *condition* is true.
3. The *update statement* controls how the variable is changed each time.

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. This first part is called the "initialization" statement.

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. The third part is sometimes called an "update statement".

After the three parts in parentheses there are curly braces which contain the "body" of the loop.

Note that the first and last parts of the for loop are complete statements; they would compile if you placed them on a line by themselves in some other Java program. But the middle part (the condition) isn't a complete statement, it's just an expression that evaluates to either true or false.

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 <= 5 ), 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 28 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" statement) 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 23 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 statement) from the *third* loop. If you remove it correctly, it will still compile. What happens when you run it?

❓ Common Student Questions

I've tried the Study Drill every which way, but I can't get it to compile!
Remove the first part, but leave the semicolon behind. A `for` loop must always have exactly two semicolons, even when one of the parts is missing.

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.

This exercise is pretty complicated but it is not very important. Don't worry if you get confused.

CaesarCipher.java

```java
import java.util.Scanner;

public class CaesarCipher {
    /**
     * Returns the character shifted by the given number of letters.
     */
    public static char shiftLetter( char c, int n ) {
        int u = c;

        if ( ! Character.isLetter(c) )
            return c;

        u = u + n;
        if ( Character.isUpperCase(c) && u > 'Z'
          || Character.isLowerCase(c) && u > 'z' ) {
            u -= 26;
        }
        if ( Character.isUpperCase(c) && u < 'A'
          || Character.isLowerCase(c) && u < 'a' ) {
            u += 26;
        }

        return (char)u;
    }

    public static void main( String[] args ) {
        Scanner keyboard = new Scanner(System.in);
```

```
28          String plaintext, cipher = "";
29          int shift;
30
31          System.out.print("Message: ");
32          plaintext = keyboard.nextLine();
33          System.out.print("Shift (0-26): ");
34          shift = keyboard.nextInt();
35
36          for ( int i=0; i<plaintext.length(); i++ ) {
37              cipher += shiftLetter( plaintext.charAt(i), shift );
38          }
39          System.out.println( cipher );
40      }
41  }
```

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 `String`s 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 7 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[19] class to tell us.

And since we are going to be doing a little math with the character, we store the character's Unicode value into an `int` on line 8 to make this easier. Then on line 13 we add the desired offset to the character.

---

[19]http://docs.oracle.com/javase/7/docs/api/java/lang/Character.html

This would be it, except that we want the offset to "wrap around", so lines 14 through 21 make sure that the final value is still a letter. Then finally on line 23 we take the value of *u*, 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:

```java
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 statement *declares* 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 statement will add `1` to *i* each time.

On line 37, a lot of things are happening. We use the `charAt` method to pull out only the *i*-th character of the plaintext 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.

The Study Drill in this exercise is pretty tough, too. If you were overwhelmed by the code so far, you should probably just skip ahead to the next exercise.

✏️ Study Drills

1. Modify the code to read the plaintext message from a text file instead of letting the human type it in. Then, instead of just displaying the ciphertext on the screen, add code to store the ciphertext into a file.

❓ Common Student Questions

Should I read the shift amount from the file or still make the human type

it in? Either way is fine. It makes more sense to let the human type it in, since that way you can "encrypt" any file you want.

Should I write to the same file I read from?

Probably not. The output file should probably be different. For extra cool points, let the human type in the *name* of the file to encrypt and make the output filename based on the input filename.

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 this means the inner loop will repeat all its iterations every time the outer loop does one more iteration.

(One "iteration" is a single time through a repeated process.)

NestingLoops.java

```java
public class NestingLoops {
    public static void main( String[] args ) {
        // this is #1 - I'll call it "CN"
        for ( char c='A'; c <= 'E'; c++ ) {
            for ( int n=1; n <= 3; n++ ) {
                System.out.println( c + " " + n );
            }
        }

        System.out.println("\n");

        // this is #2 - I'll call it "AB"
        for ( int a=1; a <= 3; a++ ) {
            for ( int b=1; b <= 3; b++ ) {
                System.out.print( "(" + a + "," + b + ") " );
            }
            // * You will add a line of code here.
        }

        System.out.println("\n");
    }
}
```

What You Should See
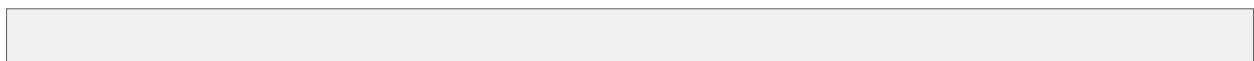
```
    A 1
    A 2
    A 3
    B 1
    B 2
    B 3
    C 1
    C 2
    C 3
    D 1
    D 2
    D 3
    E 1
    E 2
    E 3


    (1,1) (1,2) (1,3) (2,1) (2,2) (2,3) (3,1) (3,2) (3,3)
```

I'm not really going to explain this one. Just look closely at the output and the code and do the Study Drills.

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)?
2. 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?
3. 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()`.)
4. Add a `System.out.println()` statement where indicated (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? (He wants at least one of each type of animal.)"

After he left, I thought about it for a few minutes and then wrote the following program. (True story, by the way.)

FarmerBrown.java

```
1  public class FarmerBrown {
2      public static void main( String[] args ) {
3          for ( int s = 1 ; s <= 100 ; s++ ) {
4              for ( int g = 1 ; g <= 100 ; g++ ) {
5                  for ( int c = 1 ; c <= 100 ; c++ ) {
6                      if ( s+g+c == 100 && 10.00*s + 3.50*g + 0.50*c == 100.00 ) {
7                          System.out.print( s + " sheep, " );
8                          System.out.print( g + " goats, and " );
9                          System.out.println( c + " chickens." );
10                     }
11                 }
12             }
13         }
14     }
15 }
```

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 6 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 4 will execute, the condition on line 4 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 statement of the innermost loop to run again, which resets *c* to 1. So the next combination of variables 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:

FarmerBrownCompact.java

```java
public class FarmerBrownCompact {
    public static void main( String[] args ) {
        for ( int s = 1 ; s <= 100 ; s++ )
            for ( int g = 1 ; g <= 100 ; g++ )
                for ( int c = 1 ; c <= 100 ; c++ )
                    if ( s+g+c == 100 && 10.00*s + 3.50*g + 0.50*c == 100.00 ) {
                        System.out.print( s + " sheep, " + g + " goats, and " );
                        System.out.println( c + " chickens." );
                    }
    }
}
```

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:

FarmerBrownWhile.java

```java
public class FarmerBrownWhile {
    public static void main( String[] args ) {
        int s = 1;
        while ( s <= 100 ) {
            int g = 1;
            while ( g <= 100 ) {
                int c = 1;
                while ( c <= 100 ) {
                    if ( s+g+c == 100 && 10.00*s + 3.50*g + 0.50*c == 100.00 ) {
                        System.out.print( s + " sheep, " );
                        System.out.print( g + " goats, and " );
                        System.out.println( c + " chickens." );
                    }
                    c++;
                }
                g++;
            }
            s++;
        }
    }
}
```

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 value. In my opinion, you're not a Real Programmer until you can work with arrays. So, that's good news. You're almost there!

ArrayIntro.java

```java
public class ArrayIntro {
    public static void main( String[] args ) {
        String[] planets = { "Mercury", "Venus", "Earth", "Mars",
                             "Jupiter", "Saturn",  "Uranus",  "Neptune" };

        for ( String p : planets ) {
            System.out.println( p + "\t" + p.toUpperCase() );
        }
    }
}
```

What You Should See

```
Mercury MERCURY
Venus   VENUS
Earth   EARTH
Mars    MARS
Jupiter JUPITER
Saturn  SATURN
Uranus  URANUS
Neptune NEPTUNE
```

On line 3 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.

(I split the initializer list into two lines because it wouldn't have fit on one line in the printed version of this book. You can leave it like I wrote it and it will work or you can put all the Strings in one longer line.)

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`. It is also sometimes called an "enhanced for loop".)

On line 6 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 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 7) 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.

✏️ Study Drills

1. Create a second array of Strings and use an initializer list to put several values into it.

   Then add a second foreach loop to display the values from the new array on the screen, one at a time.

(There is no video for this Study Drill yet.)

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.

ArrayLinearSearch.java

```java
import java.util.Scanner;

public class ArrayLinearSearch {
    public static void main( String[] args ) {
        Scanner keyboard = new Scanner(System.in);
        int[] orderNumbers = { 12345, 54321, 101010, 8675309, 31415, 271828 };
        int toFind;

        System.out.print("There are " + orderNumbers.length);
        System.out.println(" orders in the database.");

        System.out.print("Orders: ");
        for ( int num : orderNumbers ) {
            System.out.print( num + "   " );
        }
        System.out.println();

        System.out.print("Which order to find? ");
        toFind = keyboard.nextInt();

        for ( int num : orderNumbers ) {
            if ( num == toFind ) {
                System.out.println( num + " found.");
            }
        }
    }
}
```

What You Should See

```
    There are 6 orders in the database.
    Orders: 12345  54321  101010  8675309  31415  271828
    Which order to find? 78753
```

This time the array is named *orderNumbers* and it is an array of integers. It has six slots. `12345` is the first slot, and `271828` is in the last slot of the array. Each of the six slots can hold an integer.

When we create an array Java gives us a built-in variable called `.length` which tells us the capacity of the array. This variable is read-only; you can retrieve its value but not change it. In this case, since the array *orderNumbers* has six slots, the variable `orderNumbers.length` is equal to `6`. The `.length` field is used on line 9.

On line 13 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 19 we let the human type in an order number. Then we use a 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 six 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 7 and then removed the word `int` from both loops? Try it and see.
2. 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. (Mr. Mitchell's real-life students don't need to get this working to get credit, but they must try.)

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.

ItemNotFound.java

```
1   import java.util.Scanner;
2
3   public class ItemNotFound {
4       public static void main( String[] args ) {
5           Scanner keyboard = new Scanner(System.in);
6
7           String[] heroes = {
8               "Abderus", "Achilles", "Aeneas", "Ajax", "Amphitryon",
9               "Bellerophon", "Castor", "Chrysippus", "Daedalus", "Diomedes",
10              "Eleusis", "Eunostus", "Ganymede", "Hector", "Iolaus", "Jason",
11              "Meleager", "Odysseus", "Orpheus", "Perseus", "Theseus"
12          };
13          String guess;
14          boolean found;
15
16          System.out.println( "Pop Quiz!" );
17          System.out.print( "Name any *mortal* hero from Greek mythology: " );
18          guess = keyboard.next();
```

```
19
20          found = false;
21          for ( String hero : heroes ) {
22              if ( guess.equals(hero) ) {
23                  System.out.println( "That's one of them!" );
24                  found = true;
25              }
26          }
27
28          if ( found == false ) {
29              System.out.println("No, " + guess + " wasn't a Greek mortal hero.");
30          }
31      }
32  }
```

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 never 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 and the value is changed when/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 24. Notice that inside the body of 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 28, 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.

This is a pretty important trick. I use code like this *all* the time when I'm writing programs.

✏️ Study Drills

1. A different way to say that something is *not* in an array is to *count* the number of matches. Then, when the loop is over if the counter is still zero, you know it's not there.

   Either create a new array with different values or change some of the names in the array of Greek mortal heroes so that there are some duplicates. Then add a second `for` loop that uses a counter to determine if a value is found.

(There is no video for this Study Drill yet.)

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 array. The foreach loop variable is a read-only copy of what's in the array and changing it doesn't affect 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 first 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.

ArraySlotAccess.java

```java
public class ArraySlotAccess
{
    public static void main( String[] args ) {
        int[] arr = new int[3];

        // We almost always use a for loop to access each slot of an array.
        for ( int i=0 ; i < arr.length ; i++ ) {
            arr[i] = 1 + (int)(Math.random()*100);
        }

        // Displaying all the values in an array usually looks like this
```

```
12              System.out.print("Values: ");
13              for ( int i=0 ; i < arr.length ; i++ ) {
14                  System.out.print(arr[i] + " ");
15              }
16              System.out.println();
17
18              ////////////////////////////////////////
19              // But let's break this down step-by-step...
20              // Put a number into each slot of the array, one at a time.
21              arr[0] = 6;
22              arr[1] = 7;
23              arr[2] = 8;
24
25              // The display the values in those slots, one at a time.
26              System.out.println("Values: " + arr[0] + " " + arr[1] + " " + arr[2]);
27
28              ////////////////////////////////////////
29              // Put a random number 1-100 into each slot of the array, one at a time.
30              arr[0] = 1 + (int)(Math.random()*100);
31              arr[1] = 1 + (int)(Math.random()*100);
32              arr[2] = 1 + (int)(Math.random()*100);
33
34              // Display them again, one at a time.
35              System.out.println("Values: " + arr[0] + " " + arr[1] + " " + arr[2]);
36
37              ////////////////////////////////////////
38              // This is a bit silly, but try to understand it.
39              int m = 0;
40              arr[m] = 1 + (int)(Math.random()*100);
41              m = 1;
42              arr[m] = 1 + (int)(Math.random()*100);
43              m = 2;
44              arr[m] = 1 + (int)(Math.random()*100);
45
46              // Display them again.
47              System.out.print("Values: ");
48              m = 0;
49              System.out.print(arr[m] + " ");
50              m = 1;
51              System.out.print(arr[m] + " ");
52              m = 2;
53              System.out.print(arr[m] + " ");
```

```
54          System.out.println();
55
56          /////////////////////////////////////////
57          // This is even more silly but it works.
58          int n = 0;
59          arr[n] = 1 + (int)(Math.random()*100);
60          n++;
61          arr[n] = 1 + (int)(Math.random()*100);
62          n++;
63          arr[n] = 1 + (int)(Math.random()*100);
64          n++;
65
66          // Display them again.
67          System.out.print("Values: ");
68          n = 0;
69          System.out.print(arr[n] + " ");
70          n++;
71          System.out.print(arr[n] + " ");
72          n++;
73          System.out.print(arr[n] + " ");
74          n++;
75          System.out.println();
76
77          /////////////////////////////////////////
78          // Now does using a loop make more sense?
79          for ( int q=0 ; q < arr.length ; q++ ) {
80              arr[q] = 1 + (int)(Math.random()*100);
81          }
82
83          // I hope so. If not, read through this code again more slowly.
84          System.out.print("Values: ");
85          for ( int q=0 ; q < arr.length ; q++ ) {
86              System.out.print(arr[q] + " ");
87          }
88          System.out.println();
89      }
90 }
```

What You Should See

```
    Values: 82 42 59
    Values: 6 7 8
    Values: 13 21 41
    Values: 42 44 78
    Values: 83 93 78
    Values: 61 74 96
```

On line 4 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 I skip straight to the hard part: using a for loop to make a variable whose value changes into all the *locations* of the values in the array. That is, the variable *i* will equal `0` then `1` then `2` instead of the *values* in the array.

In the body of the loop we store a random number from 1-100 into each slot in the array. I'll explain more later.

On lines 13 through 17 we use an identical loop to display all the random values from the array onto the screen. If both of these loops don't make total sense to you, you're in luck! The rest of the code explains the technique.

So let's skip down to line 22, which uses square brackets and a number to store the *value* 6 into *slot* 0 of the array.

Maybe this seems weird. But the first slot in an array is slot number `0` in most programming languages. You could also say that the first slot in an array has the index 0, because 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 Java – like almost all other programming languages – has "0-based array indexes." This means that though our array has room for three values, they are numbered 0-1-2.

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 21 through 23 store values into all three slots in the array. (All the slots used to have random numbers in them, but those have now been overwritten.)

On line 26 we print out all three current values in the array so you can see that they have been changed.

On lines 30 through 32 we put random numbers into each slot of the array. And print them out again on line 35.

Starting on line 39 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 40 is essentially identical to line 30? Line 40 stores a random number into a spot in the array. Which spot? The index depends on the current value of *m*. And *m* is currently `0`. So we are storing the random number into the slot with index `0`. Okay?

So on line 41 we *change* the value of *m* from `0` to `1`. Then on line 42 we store a random value in the slot indexed by the value of *m*, so index `1`. Clear? Weird, but legal.

I have used similar shenanigans on lines 47 through 54 to display all the values on the screen again. Now, this is clearly objectively worse than what I was doing on line 26. I mean, it took me 8 lines of code to do what I had been doing in one line. (Stay with me.)

On lines 58 through 64 we do something that might even be worse than lines 39 through 44. Lines 58 and 59 are the same, but instead of putting a `1` directly into *n* on line 60, I just say "increase the value of *n* by 1." So *n* 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 61 and 62 are *literally identical* to lines 59 and 60. And the same for lines 63 and 64. I mean, like byte-for-byte the same.

We display them in a similar silly fashion on lines 67 through 75.

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 79 through 81 are the same as lines 58 through 64 except that we let the `for` loop handle the repeating and the changing of the index. The initialization statement (the first part) of the `for` loop sets *q* to start at `0`, which happens to be the smallest legal index for an array. The condition says "repeat this as long as *q* is less than `arr.length` (which is `3`)." And note that it says *less than*, not less than or equal to, which would be too far. The update statement (the third part) just adds 1 to *q* each time.

Lines 84 through 88 display the values on the screen.

Here's the thing: this sort of code on lines 79 through 88 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 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.

Common Student Questions

Why did you use the variable *i* to hold the array index in the first few

loops, then use *m* elsewhere and *n* somewhere else and *q* at the end?

Do you have to change it like that?

No. In fact, in real code I almost always use *i* as my array index in loops. (*i* is short for "index", actually.) But there's nothing magical about *i*. The only thing that matters is that your index variable holds 0 then 1 then 2, etc. You can use the same index variable over and over again (and most people do) or you can use a different one every time.

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:

- https://learnjavathehardway.org/txt/avg-daily-temps-atx.txt[20]

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.

LowestTemperature.java

```
1  import java.net.URL;
2  import java.util.Scanner;
3
4  public class LowestTemperature {
5      public static void main(String[] args) throws Exception {
6          String urlbase = "https://learnjavathehardway.org/txt/";
7          double[] temps = arrayFromUrl(urlbase + "avg-daily-temps-atx.txt");
8
9          System.out.println( temps.length + " temperatures in database.");
10
11         double lowest = 9999.99;
12
13         for ( int i=0; i<temps.length; i++ ) {
14             if ( temps[i] < lowest ) {
15                 lowest = temps[i];
16             }
17         }
18
19         System.out.print( "The lowest average daily temperature was " );
20         System.out.println( lowest + "F (" + fToC(lowest) + "C)" );
21     }
```

[20]https://learnjavathehardway.org/txt/avg-daily-temps-atx.txt

```
22
23      public static double[] arrayFromUrl( String url ) throws Exception {
24          Scanner fin = new Scanner((new URL(url)).openStream());
25          int count = fin.nextInt();
26
27          double[] dubs = new double[count];
28
29          for ( int i=0; i<dubs.length; i++ )
30              dubs[i] = fin.nextDouble();
31          fin.close();
32
33          return dubs;
34      }
35
36      public static double fToC( double f ) {
37          return (f-32)*5/9;
38      }
39  }
```

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 up front I have done something difficult. On line 7 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 23 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 24 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 25 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 27. 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 27 we define an array of doubles with *count* slots. (Currently 6717.)

On line 29 there's a `for` loop that iterates through each slot in the array, and on line 30 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 33 the array is returned from the function and that array is what is stored into the array *temps* back on line 7 of `main()`.

On line 9 we print out the current length of the array to make sure nothing went wrong with the reading.

On line 11 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 13 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 14 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 15 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 36 through 38 to convert a temperature from degrees Fahrenheit to degrees Celsius. So on line 20 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.
2. Try to find another temperature file online for a city closer to where you live and change your code to read from it instead! (Mr. Mitchell's real-life students don't need to do this Study Drill to receive credit.)

(I mentioned it above, but this is the modified code to read the temperature data from a local file. You only need to type it in if you can't run your Java programs on a machine with Internet access.)

LowestTemperatureLocal.java

```java
import java.io.File;
import java.util.Scanner;

public class LowestTemperatureLocal {
    public static void main(String[] args) throws Exception {
        double[] temps;
        double lowest = 9999.99;

        // Read values from local file instead of URL
        Scanner fin = new Scanner(new File("avg-daily-temps-atx.txt"));
        temps = new double[fin.nextInt()];

        System.out.println( temps.length + " temperatures in database.");

        for ( int i=0; i<temps.length; i++ )
            temps[i] = fin.nextDouble();
        fin.close();

        for ( int i=0; i<temps.length; i++ )
            if ( temps[i] < lowest )
                lowest = temps[i];

        System.out.print( "The lowest average daily temperature was " );
        System.out.println( lowest + "F (" + fToC(lowest) + "C)" );
    }

    public static double fToC( double f ) { return (f-32)*5/9; }
}
```

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").[21]

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.)

MailingAddresses.java

```
1   class Address {
2       String street;
3       String city;
4       String state;
5       int zip;
6   }
7
8   public class MailingAddresses {
9       public static void main(String[] args) {
10          Address uno, dos, tres;
11
12          uno = new Address();
13          uno.street = "191 Marigold Lane";
14          uno.city   = "Miami";
15          uno.state  = "FL";
16          uno.zip    = 33179;
17
18          dos = new Address();
19          dos.street = "3029 Losh Lane";
20          dos.city   = "Crafton";
```

---

[21]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.

But 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.

```
21              dos.state  = "PA";
22              dos.zip    = 15205;
23
24          tres = new Address();
25          tres.street = "2693 Hannah Street";
26          tres.city   = "Hickory";
27          tres.state  = "NC";
28          tres.zip    = 28601;
29
30      System.out.println( uno.street );
31      System.out.println( uno.city + ", " + uno.state + "  " + uno.zip );
32      System.out.println( "\n" + dos.street );
33      System.out.println( dos.city + ", " + dos.state + "  " + dos.zip );
34      System.out.println( "\n" + tres.street );
35      System.out.println( tres.city + ", " + tres.state + "  " + tres.zip );
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 6 we have defined a record called Address.

(I know it says class, not record. If I could do something about this I promise that 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 8 our "real" class starts.

On line 10 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 12 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 13.

Line 14 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 10 and change the code to put *your* mailing address in it. Don't forget to print it out at the bottom.

Common Student 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.

Footnotes:

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.

- https://learnjavathehardway.org/txt/s01e01-cast.txt[22]

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.)

ActorList.java

```
1   import java.util.Scanner;
2
3   class Actor {
4       String name;
5       String role;
6       String birthdate;
7   }
8
9   public class ActorList {
10      public static void main(String[] args) throws Exception {
11          String url = "https://learnjavathehardway.org/txt/s01e01-cast.txt";
12          // Scanner inFile = new Scanner(new java.io.File("s01e01-cast.txt"));
13          Scanner inFile = new Scanner((new java.net.URL(url)).openStream());
14
15          while ( inFile.hasNext() ) {
16              Actor a = getActor(inFile);
17              System.out.print(a.name + " was born on " + a.birthdate);
18              System.out.println(" and played " + a.role);
19          }
20          inFile.close();
21      }
22
```

_____

[22]https://learnjavathehardway.org/txt/s01e01-cast.txt

```
23      public static Actor getActor( Scanner input ) {
24          Actor a = new Actor();
25          a.name = input.nextLine();
26          a.role = input.nextLine();
27          a.birthdate = input.nextLine();
28
29          return a;
30      }
31  }
```

What You Should See

```
Sean Bean was born on 1959-04-17 and played Eddard 'Ned' Stark
Mark Addy was born on 1964-01-14 and played Robert Baratheon
Nikolaj Coster-Waldau was born on 1970-07-27 and played Jaime Lannister
Michelle Fairley was born on 1964-01-17 and played Catelyn Stark
```

This time our record is called Actor and has three fields, all of which are Strings.

On line 13 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 13; 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 12 so that it is no longer a comment and then *add* two slashes at the beginning on line 13 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 13 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[23] I showed you one trick for dealing with this: storing

---

[23]ex52.html

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 15 we create a `while` loop that repeats as long as `.hasNext()` continues to return `true`.

Before we look at line 16 let us skip down to lines 23 through 30 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 24. 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 25 through 27 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 16 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 (a.k.a. "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 16 in `main()`, but that variable goes out of scope when the close curly brace occurs on line 19. There is a different `Actor` variable called *a* declared (and defined) on line 24 in the `getActor()` function, but it goes out of scope when the close curly brace occurs on line 30.

Okay, back to line 16. 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 23 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 17 and 18 are pretty boring: they simply display the values of all the fields of the record. On line 19 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 20 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.

✏️ Study Drills

1. Add a counter inside the loop that reads in all the records. After the loop is over display a message on the screen about how many records were processed.

(There is no video for this Study Drill yet.)

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.

StudentDatabase.java

```java
class Student {
    String name;
    int credits;
    double gpa;
}

public class StudentDatabase {
    public static void main( String[] args ) {
        Student[] db;
        db = new Student[3];

        db[0] = new Student();
        db[0].name = "Esteban";
        db[0].credits = 43;
        db[0].gpa = 2.9;

        db[1] = new Student();
        db[1].name = "Dave";
        db[1].credits = 15;
        db[1].gpa = 4.0;

        db[2] = new Student();
        db[2].name = "Michelle";
        db[2].credits = 132;
        db[2].gpa = 3.72;

        for ( int i=0; i<db.length; i++ ) {
            System.out.println("Name: " + db[i].name);
            System.out.println("\tCredit hours: " + db[i].credits);
            System.out.println("\tGPA: " + db[i].gpa + "\n");
        }

        int maxLoc = 0;
```

```
34            for ( int i=1; i<db.length; i++ )
35                if ( db[i].gpa > db[maxLoc].gpa )
36                    maxLoc = i;
37
38            System.out.println(db[maxLoc].name + " has the highest GPA.");
39        }
40  }
```

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 9 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 9 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 10 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 12 it is important that a Student object is created and stored into the first slot (index 0) of the array. Then on line 13 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 13 stores a value into the *name* field of the first record in the array. Lines 14 and 15 store values into the remaining fields in that record. Lines 17 through 25 create then fill the other two records in the array.

On lines 27 though 31 we use a loop to display all the values on the screen.

Then lines 33 through 36 find the student with the highest GPA. This is worth explaining in more detail. On line 33 an int called *maxLoc* is defined. But *maxLoc* 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 *maxLoc* 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 34 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 *maxLoc* 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 *maxLoc* from 0 to the current value of *i* (1)."

So by the time the loop is over, *maxLoc* contains the **index** of the record with the highest GPA. Which is exactly what we display on line 38.

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?
2. 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.
3. Add code so that the program also finds the student with the fewest credits.

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.

- https://learnjavathehardway.org/txt/avg-daily-temps-with-dates-atx.txt[24]

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[25]. This means three things:

1. There is no number in the first line of the file telling us how many records there are.
2. In addition to the temperature each record also has the month, day and year for the sample.
3. 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.

TemperaturesByDate.java

```
1   import java.util.Scanner;
2
3   class TemperatureSample {
4       int month, day, year;
5       double temperature;
6   }
7
8   public class TemperaturesByDate {
9       public static void main(String[] args) throws Exception {
10          String site = "https://learnjavathehardway.org";
11          String path = "/txt/avg-daily-temps-with-dates-atx.txt";
12          Scanner inFile = new Scanner((new java.net.URL(site+path)).openStream());
13
14          TemperatureSample[] tempDB = new TemperatureSample[10000];
```

---

[24]https://learnjavathehardway.org/txt/avg-daily-temps-with-dates-atx.txt
[25]http://academic.udayton.edu/kissock/http/Weather/

```
15          int numRecords, i = 0;
16
17          while ( inFile.hasNextInt() && i < tempDB.length ) {
18              TemperatureSample e = new TemperatureSample();
19              e.month = inFile.nextInt();
20              e.day   = inFile.nextInt();
21              e.year  = inFile.nextInt();
22              e.temperature = inFile.nextDouble();
23              if ( e.temperature == -99 )
24                  continue;
25              tempDB[i] = e;
26              i++;
27          }
28          inFile.close();
29          numRecords = i;
30
31          System.out.println(numRecords + " daily temperatures loaded.");
32
33          double total = 0, avg;
34          int count = 0;
35          for ( i=0; i<numRecords; i++ ) {
36              if ( tempDB[i].month == 11 ) {
37                  total += tempDB[i].temperature;
38                  count++;
39              }
40          }
41
42          avg = total / count;
43          avg = roundToOneDecimal(avg);
44          System.out.print("Average daily temperature over " + count);
45          System.out.println(" days in November: " + avg);
46      }
47
48      public static double roundToOneDecimal( double d ) {
49          return Math.round(d*10)/10.0;
50      }
51  }
```

What You Should See

```
    6717 daily temperatures loaded.
    Average daily temperature over 540 days in November: 59.7
```

Lines 3 through 6 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 14 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.
2. 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.
3. 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 14 declares the array and defines it to have ten thousand slots: "big enough."

On line 17 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 18 defines a TemperatureSample record named *e*. Lines 19 through 22 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 23 to detect that before we put them into our database.

Then on line 24 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 25 and 26, 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 27 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 31 we display the number of records on the screen, which can help you to see if anything went wrong while reading.

Lines 35 through 39 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 48 through 50 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 43 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. (Mr. Mitchell's real-life students *do* need to do this Study Drill for full credit.)
2. Change the code to find out other things, like the highest temperature in February or whatever suits your fancy.
3. 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.

PickACard.java

```java
1   class Card {
2       int value;
3       String suit;
4       String name;
5
6       public String toString() {
7           return name + " of " + suit;
8       }
9   }
10
11  public class PickACard {
12      public static void main( String[] args ) {
13          Card[] deck = buildDeck();
14          // displayDeck(deck);
15
16          int chosen = (int)(Math.random()*deck.length);
17          Card picked = deck[chosen];
18
19          System.out.println("You picked a " + picked + " out of the deck,");
20          System.out.println("worth " + picked.value + " points in Blackjack.");
21      }
22
23      public static Card[] buildDeck() {
24          String[] suits = { "clubs", "diamonds", "hearts", "spades" };
25          String[] names = { "ZERO", "ONE", "two", "three", "four", "five", "six",
26              "seven", "eight", "nine", "ten", "Jack", "Queen", "King", "Ace" };
27
28          int i = 0;
29          Card[] deck = new Card[52];
30
31          for ( String s: suits ) {
32              for ( int v = 2; v <= 14 ; v++ ) {
33                  Card c = new Card();
```

```
34                    c.suit = s;
35                    c.name = names[v];
36                    if ( v == 14 )
37                        c.value = 11;
38                    else if ( v > 10 )
39                        c.value = 10;
40                    else
41                        c.value = v;
42
43                    deck[i] = c;
44                    i++;
45                }
46            }
47        return deck;
48    }
49
50    public static void displayDeck( Card[] deck ) {
51        for ( Card c : deck )
52            System.out.println(c.value + "\t" + c);
53    }
54 }
```

What You Should See

```
You picked a Ace of clubs out of the deck,
worth 11 points in Blackjack.
```

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 ("two" or "Jack").

On lines 12 though 21 you can see the main(), which is really short. Line 13 declares an array of cards and initializes it using the return value of the buildDeck() function.

Line 14 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 16 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 16 chooses a random index into the array or that line 16 chooses a slot of the array randomly.

Then on line 17 we declare a new Card variable called *picked* and give it a value from the randomly-chosen slot of the array.

Line 19 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 19 will print on the screen the result of running the picked card's toString() method.

By contrast, line 20 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 51 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 52 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 24 through 26 we create two arrays of Strings. The first one (on line 24) 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 use 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 28 we create *i*, which will keep track of which index needs to have a Card put into it next. That is, *i* will always hold the index of the *next available* slot in the array.

Line 29 defines our array of 52 cards (indexed 0 through 51).

Line 31 is another foreach loop. The variable *s* is going to be set equal to "clubs", then "diamonds", then "hearts" and then finally "spades".

Line 32 begins 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 33 defines a Card named *c*. On line 34 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 35 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 36 through 41 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 since that has already been done, we need to make sure that no card gets a value of 12, 13 or 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 14 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 13 of main().

✏️ Study Drills

1. Add a function called `shuffleDeck()`. It should take in an array of cards as a parameter and return nothing (`void`). One way to shuffle[26] 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.

Footnotes:

[26]In an earlier edition of this book, I got email from a mathmatician complaining that my method for shuffling cards isn't very good. In particular, it is not only very inefficient, but is "biased". That means some outcomes are more likely than others. This is a legitimate criticism! If you found the shuffling code easy to write and want to improve it, consider re-writing the `shuffleDeck()` function to use something unbiased like the Fisher-Yates shuffle.

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 story 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.

- https://learnjavathehardway.org/txt/text-adventure-rooms.txt[27]

Then, you had better get started typing. This is a long one, but I think it will be worth it.

TextAdventureFinal.java

```java
1  import java.util.Scanner;
2
3  class Room {
4      int roomNumber;
5      String roomName;
6      String description;
7      int numExits;
8      String[] exits = new String[10];
9      int[] destinations = new int[10];
10 }
11
12 public class TextAdventureFinal {
13     public static void main( String[] args ) {
14         Scanner keyboard = new Scanner(System.in);
15
16         // initialize rooms from file
17         Room[] rooms = loadRoomsFromFile("text-adventure-rooms.txt");
18
```

[27] https://learnjavathehardway.org/txt/text-adventure-rooms.txt

```
19              // showAllRooms(rooms); // for debugging
20
21              // Okay, so let's play the game!
22              int currentRoom = 0;
23              String ans;
24              while ( currentRoom >= 0 ) {
25                  Room cur = rooms[currentRoom];
26                  System.out.print( cur.description );
27                  System.out.print("> ");
28                  ans = keyboard.nextLine();
29
30                  // See if what they typed matches any of our exit names
31                  boolean found = false;
32                  for ( int i=0; i<cur.numExits; i++ ) {
33                      if ( cur.exits[i].equals(ans) ) {
34                          // if so, change our next room to that exit's room number
35                          currentRoom = cur.destinations[i];
36                          found = true;
37                      }
38                  }
39                  if ( ! found )
40                      System.out.println("Sorry, I don't understand.");
41              }
42          }
43
44      public static Room[] loadRoomsFromFile( String filename ) {
45          Scanner file = null;
46          try {
47              file = new Scanner(new java.io.File(filename));
48          }
49          catch ( java.io.IOException e ) {
50              System.err.println("Can't open '" + filename + "' for reading.");
51              System.exit(1);
52          }
53
54          int numRooms = file.nextInt();
55          Room[] rooms = new Room[numRooms];
56
57          // initialize rooms from file
58          int roomNum = 0;
59          while ( file.hasNext() ) {
60              Room r = getRoom(file);
```

```
61                  if ( r.roomNumber != roomNum ) {
62                      System.err.print("Just read room # " + r.roomNumber);
63                      System.err.println(", but " + roomNum + " was expected.");
64                      System.exit(2);
65                  }
66              rooms[roomNum] = r;
67              roomNum++;
68          }
69          file.close();
70
71          return rooms;
72      }
73
74      public static void showAllRooms( Room[] rooms ) {
75          for ( Room r : rooms ) {
76              String exitString = "";
77              for ( int i=0; i<r.numExits; i++ )
78                  exitString += "\t" + r.exits[i] + " (" + r.destinations[i] + ")";
79              System.out.println( r.roomNumber + ") " + r.roomName );
80              System.out.println( exitString );
81          }
82      }
83
84      public static Room getRoom( Scanner f ) {
85          // any rooms left in the file?
86          if ( ! f.hasNextInt() )
87              return null;
88
89          Room r = new Room();
90          String line;
91
92          // read in the room # for error-checking later
93          r.roomNumber = f.nextInt();
94          f.nextLine();   // skip "\n" after room #
95
96          r.roomName = f.nextLine();
97
98          // read in the room's description
99          r.description = "";
100         while ( true ) {
101             line = f.nextLine();
102             if ( line.equals("%%") )
```

```
103                    break;
104                r.description += line + "\n";
105            }
106
107            // finally, we'll read in the exits
108            int i = 0;
109            while ( true ) {
110                line = f.nextLine();
111                if ( line.equals("%%") )
112                    break;
113                String[] parts = line.split(":");
114                r.exits[i] = parts[0];
115                r.destinations[i] = Integer.parseInt(parts[1]);
116                i++;
117            }
118            r.numExits = i;
119
120            // should be done; return the Room
121            return r;
122        }
123 }
```

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". An exit 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 of a record for a single room.

```
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 10 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 17 declares the array of rooms and initializes it from the `loadRoomsFrom-File()` function that I will explain later.

Line 19 has a commented-out call to a `showAllRooms()` function that I use for debugging.

On line 22 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 24 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 25 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 26 prints out the description of the current room, which is stored in the *description* field.

On lines 27 and 28 we print out a little prompt and let the player enter in a String for where they want to go.

Lines 32 through 38 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 `cur.exits.length`.

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 39 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 44 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 54 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 55 defines an array of Room records with the appropriate capacity.

On line 58 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 61), we complain and end the program. (`System.exit()` ends the program, even from inside a function call.)

Line 59 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 60 we read a single room record using the `getRoom()` function I'll explain later.

Lines 61 through 65 are the room number sanity check I already mentioned, and then line 66 just stores this new room into the next available slot in the rooms array. And line 67 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 71 we can return the array of rooms back up to line 17 of `main()`.

Lines 74 through 82 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 86 and 87 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 89 the empty room object is defined. Line 90 creates a String called *line*, which I use for a couple of different things.

Line 93 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 94 calls the `nextLine()` method but doesn't bother to store its return value anywhere because it doesn't read anything worth saving.

Line 96 reads in the room name from the file. We only use this for debugging.

On line 99 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 100 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 102 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 104, it means that we successfully 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 108 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 109 there's another infinite loop. I use a very similar approach to read in all the exits.

Line 110 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 111 and 112 stop the loop.)

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 113 `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 115.

Recall that *i* is the index of the slot in the exits array where we need to store the next value. So line 114 stores `parts[0]` (the name of the exit) into the appropriate slot of the exits array. And line 115

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 116 increments the exit index for the next iteration.

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 118.

And that should be it. All the fields in the room have been given values, and we `return` this `Room` object to line 60 of the `loadRoomsFromFile()` function.

✏ Study Drills

1. Write your own text adventure. If you think it turns out pretty good, email it to me! (Mr. Mitchell's real-life students do not need to do this study drill for full credit.)
2. 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.

Next Steps

You made it!

If you finished all the exercises and did the Study Drills and understood what you were doing, then congratulations! Java isn't an easy language for beginners, so pat yourself on the back or something.

You now know the basics of programming, and more importantly, you have a lot of practice reading code, understanding it, and fixing bugs. These are important tools for programmers. With these skills, you should be able to pick up just about any book on programming and handle it just fine.

However, your journey isn't over yet. Typing in someone else's programs is hard but writing your own programs from scratch is a lot harder. Also, you know the basics of Java but nothing about "Object-Oriented Programming", which is a pretty big deal.

Most of my real-life students practiced and learned for four or five more years *after* working through my assignments before they got jobs in the industry.

Currently, you are lacking in three main areas:

- You need a lot more practice, especially writing your own programs.
- You don't know Object-Oriented Programming (OOP).
- You don't know the standard library very well.

You don't know the standard library.

Professional Java programmers spend a lot of time writing small amounts of code to "glue together" modules written by other people. Many of these are included with Java itself and are available to "import".

Java's standard library is provided as packages of objects, so you'll need to know object-oriented programming before you can get much better at this.

You need a lot more practice.

The students I teach at my public school *do* use the exercises in this book to start learning, but that's not all I make them do. They also have to write a lot of programs from scratch. You can see the list of assignments at

Programming by Doing - https://programmingbydoing.com/

I *highly* recommend working through them. Once you've done that, here are some other resources for practice:

- CodingBat[28] - short, targeted practice problems. These are especially good practice for students preparing for the Advanced Placement (AP) exam in Computer Science.
- Project Euler[29] - a series of challenging mathematical/computer programming problems that will require more than just mathematical insights to solve. Although mathematics will help you arrive at elegant and efficient methods, the use of a computer and programming skills will be required to solve most problems.

You don't know OOP.

Java is an object-oriented language. No matter how good you get at assignments like mine, you can never truly call yourself a **Java** programmer until you know OOP well.

I am writing a sequel to this book that teaches object-oriented programming; I hope to have it finished by April 2016 or at least by September 2016. You don't have to email me to ask if it's done, though; I can *promise* you that I won't keep it a secret. learnjavathehardway.org will *definitely* mention the new book once it's done.

You can also see if it has been published at Leanpub yet at <leanpub.com/javahard2>. However, until it's finished I'll have to recommend other people's books.

- Java Methods[30] by Maria and Gary Litvin - an excellent book designed to prepare students for the APCS exam. Teaches objects and object-oriented programming from the very beginning.
- any other highly-rated Java book on Amazon. Almost all books that teach you Java will cover objects and OOP. If you work through any book after completing mine, I would love to hear your opinions of it!

Thanks so much for using my book to start your journey coding. Please tweet @grahammitchell using the hashtag #LJtHW to let me know!

Happy coding!
– Graham Mitchell

---

[28] http://codingbat.com/java
[29] https://projecteuler.net/
[30] http://www.skylit.com/jm.html