SECOND EDITION

Learn Java the Hard Way

Graham Mitchell

Version 20151122.001

Learn Java the Hard Way

Graham Mitchell

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

This version was published on 2016-07-06

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

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

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

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 at conqueringthecommandline.com[1].

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.

[1] http://conqueringthecommandline.com/book/basics

Windows

Installing a Decent Text Editor (Notepad++)

1. Go to notepad-plus-plus.org[2] 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.

[2] http://notepad-plus-plus.org/

Installing the Java Development Kit (JDK)

1. Go to Oracle's Java SE downloads page[3] with your web browser.
2. Click the big "Java" button on the left near the top to download the Java Platform (JDK) 8u92. 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 **8u92** 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.

   You do *not* need to download the "Demos and Samples".
4. Once downloaded, run `jdk-8u92-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_92\` 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_92` 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_92\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_92\bin", "User")
```

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

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

[3] http://www.oracle.com/technetwork/java/javase/downloads/index.html

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 8u92, it should be

```
C:\Program Files (x86)\Java\jdk1.8.0_92\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_92`.

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

```
java -version
```

You should see a response like `java version "1.8.0_92"`.

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 barebones.com[4] 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 Oracle's Java SE downloads page[5] with your web browser.

[4] http://www.barebones.com/products/textwrangler/

[5] http://www.oracle.com/technetwork/java/javase/downloads/index.html

2. Click the big "Java" button on the left near the top to download the Java Platform (JDK) 8u92. 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 **8u92** or any newer version.

   You do *not* need to download the "Demos and Samples".
4. Once downloaded, run `jdk-8u92-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_92`.

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

```
java -version
```

You should see a response like `java version "1.8.0_92"`.

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's Java SE downloads page[6].

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_91`.

[6] http://www.oracle.com/technetwork/java/javase/downloads/index.html

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

```
java -version
```

You should see a response like `openjdk version "1.8.0_91"`.

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 `]` 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 no other brackets in the whole file.
>
> `{` 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

```
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

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

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[7] 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

[7] 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[8] 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:

[8] 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 = 2.718281828459045235 36;
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[9] 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

[9] 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[10] 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:

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

Exercise 6: Mathematical Operations

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

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

[11] "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[12] that can be included if desired.

```
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:

[12] 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.

```
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

```
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!

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

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.

```
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!

```
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. (It will take more than one line of code.)

(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!

```
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.

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

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

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

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

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

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

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

```
1  import java.util.Scanner;
2
3  public class WeaselOrNot {
4      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

```
1  import java.util.Scanner;
2
3  public class AgeMessages {
4      public static void main( String[] args ) {
5          Scanner keyboard = new Scanner(System.in);
6          int age;
7
8          System.out.print( "How old are you? " );
9          age = keyboard.nextInt();
10
11         System.out.println( "You are: " );
12         if ( age < 13 ) {
13             System.out.println( "\ttoo young to create a Facebook account" );
14         }
15         if ( age < 16 ) {
16             System.out.println( "\ttoo young to get a driver's license" );
17         }
18         if ( age < 18 ) {
19             System.out.println( "\ttoo young to get a tattoo" );
20         }
21         if ( age < 21 ) {
22             System.out.println( "\ttoo young to drink alcohol" );
23         }
24         if ( age < 35 ) {
25             System.out.println( "\ttoo young to run for President of the U.S." );
26             System.out.println( "\t\t(How sad!)" );
27         }
28     }
29 }
```

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.