Learn Object-Oriented Java the Hard Way

Graham Mitchell

Learn Object-Oriented Java the Hard Way

Graham Mitchell

This book is for sale at http://leanpub.com/javahard2

This version was published on 2016-10-12

Also By Graham Mitchell

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: Object-Oriented Java

Java is an object-oriented programming language. My first book covered all the basic syntax of the Java language, but avoided all the object-oriented parts. This book covers the rest.

If you have never programmed before in *any* language, this book is probably not for you. You need some experience in a similar language before you will be able to make it through this book. If you already know the basics of Java or another language like C, C++, C# or Javascript, you will be okay. If you only know a very different language like Python or Ruby then you'll have a little catching up to do.

If you get lost trying to follow the code in exercise 1, then you should probably go back and work through a simpler book before trying this one.

What You Will Learn

- How to install the Java compiler and a text editor
- How to work with Java objects and create your own classes
- Fields and instance variables
- Methods and Parameters
- Constructors
- Reference Variables vs Primitives
- Generics and Casting
- Inheritance
- Interfaces
- Abstract Classes and Methods
- Packages
- How to create JAR files
- Graphical User Interfaces in JavaFX
- Mouse and Keyboard Input in GUIs
- Testing and Efficiency
- Algorithmic Complexity and Big-O Notation
- ArrayLists
- Sorting

…and more!

In the final chapter you'll write a graphical version of a popular checker-dropping game and be able to package that up to send to others.

All the examples in this book will work in version 1.8 of Java or any newer version. If you omit the last few chapters on JavaFX, most of the code will work in Java version 1.6 or later.

What You Will *Not* Learn

- How to compile and run Java programs in a terminal
- The basics of Java
- How to make an Android app
- Specifics of different "versions" of Java
- Javascript

Create, compile & run

If you have written some Java before but you have always used an IDE, you should learn how to write your code in a simple text editor and how to compile your code from a terminal. My first book has an entire chapter[1] on it which is free to read online, so work through that first if you need to.

No basics

If you don't already know how to create variables and write if statements, loops and functions in Java, then you should learn that before trying to work through this book.

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.

Except for the last few graphics chapters, I will also not cover anything that was only recently added to Java. This book is for learning the basics of object-oriented 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.

[1]https://learnjavathehardway.org/book/ex01.html

I hope to write more books after this one. 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[2].

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.

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

Windows

Installing a Decent Text Editor (Notepad++)

1. Go to notepad-plus-plus.org[3] 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.

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

Installing the Java Development Kit (JDK)

1. Go to Oracle's Java SE downloads page[4] with your web browser.
2. Click the big "Java" button on the left near the top to download the Java Platform (JDK) 8u102. 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 **8u102** 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-8u102-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_102\` 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_102` 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_102\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:

   <<(code/set-environ-var-8u102.txt)

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

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

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

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

Then type or paste the folder location from above. If you installed the x86 (32-bit) version of JDK version 8u102, it should be

```
C:\Program Files (x86)\Java\jdk1.8.0_102\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_102`.

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

```
java -version
```

You should see a response like `java version "1.8.0_102"`.

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 javahard2
```

`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 "javahard2" if you want to. You will only need to create this folder once per computer.

```
cd javahard2
```

Change into the `javahard2` 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 "javahard2" 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 "javahard2" 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[5] 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[6] with your web browser.

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

[6] 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) 8u102. 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 **`8u102`** or any newer version.

   You do *not* need to download the “Demos and Samples”.
4. Once downloaded, run `jdk-8u102-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_102`.

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

```
java -version
```

You should see a response like `java version "1.8.0_102"`.

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 javahard2
```

`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 "javahard2" if you want to. You will only need to create this folder once per computer.

```
cd javahard2
```

Change into the `javahard2` 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 "javahard2" 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 "javahard2" 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 openjfx
```

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[7].

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".

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

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`.

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 javahard2
```

`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 "javahard2" if you want to. You will only need to create this folder once per computer.

```
cd javahard2
```

Change into the `javahard2` 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 “javahard2” 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 “javahard2” 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: Working With Objects

There's no getting away from it, Java is an object-oriented language. In the original "Learn Java the Hard Way", I tried to avoid the object-oriented parts of Java as much as possible, but some of them still snuck in!

In this chapter we will look at some common patterns Java uses when creating and working with objects, and I'll also have a brief reminder of how to compile and execute Java programs from a command-prompt or terminal window.

Type in the following code into a single file called `WorkingWithObjects.java` and put it into a folder you can get to from the terminal window.

If some of this is unfamiliar, don't worry about it. We're just going to be looking at patterns, and the details aren't that important in this assignment. In particular, you might not have ever used `ArrayList` or `Random`, and that's perfectly fine.

If this code is *extremely* overwhelming, then you might have a problem. Maybe you don't know what an `if` statement is, or `System.out.println`, or you've never used a `for` loop. In that case, this book is probably going to be too difficult for you. You should go back and work through an easier book first and then come back here once you're quite comfortable with the basics of Java.

Anyway, type up the code below and then I'll remind you how to compile it from the terminal. Remember that you shouldn't be using any IDE for these exercises. Also remember not to type in the line numbers in front of each line; those are just there to make it easier to talk about the code later.

WorkingWithObjects.java

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

public class WorkingWithObjects {
    public static void main( String[] args ) throws Exception {
        File f = new File("datafiles/phonetic-alphabet.txt");

        if ( f.exists() == false ) {
            System.out.println( f.getName() + " not found in this folder. :(");
            System.exit(1);
        }

        ArrayList<String> words = new ArrayList<String>();
        Scanner alpha = new Scanner(f);

        System.out.print("Reading words from \"" + f.getPath() + "\"... ");
        while ( alpha.hasNext() ) {
            String w = alpha.next();
            words.add(w);
        }
        alpha.close();
        System.out.print("done.\n\t");

        Random rng = new Random();
        rng.setSeed(12345);
        // rng.setSeed(23213);

        for ( int n=0; n<3; n++ ) {
            int i = rng.nextInt( words.size() );
            String s = words.get(i);
            System.out.print( s.toLowerCase() + " " );
        }
        System.out.println();
    }
}
```

Once you've got the code typed in I'm going to assume that you saved the file `WorkingWithObjects.java` into a folder called `javahard2`. If you saved it into a different folder, substitute the name below.

Open up a terminal window and change into the `javahard2` folder:

```
cd javahard2
ls
```

(If you're stuck on a much older version of Windows that doesn't have Powershell, then you will have to type `dir`. Everybody else gets to type `ls`, though.)

Hopefully you'll see an output listing that includes:

```
WorkingWithObjects.java
```

That means you're in the right place. Then you'll compile the file using the Java compiler, which is called `javac`:

```
javac WorkingWithObjects.java
```

If this command gives an error about `javac` itself, then you skipped Exercise 0! Go back and make sure the JDK is installed and in the PATH, then come back!

Assuming you have good 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 a mistake, you will see some error. 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 next to your code:

```
javac WorkingWithObjects.java
ls
```

```
WorkingWithObjects.class
WorkingWithObjects.java
```

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 WorkingWithObjects
```

What You Should See

```
Reading words from "datafiles/phonetic-alphabet.txt"... done.
        juliett uniform foxtrot
```

Okay, now that *that* is working, let's talk about some of the patterns we see in this exercise.

Line 1 imports a "library". `java.io.File` contains the definition for a object/class called `File`. Once we have imported it, we can create `File` objects in our code and call methods defined inside those objects.

The next three lines import three more libraries, each defining an object. Some programming languages call these imported things "modules" instead of libraries. Same thing.

On line 8 we instantiate a `File` object and name it `f`. The `File` object is passed a String parameter containing the name of a file to connect itself to.

> You should have been provided a folder called `datafiles` with several files in it, including a text file named `phonetic-alphabet.txt`. Make sure this folder is *inside* the folder where your Java file is located. Copy or move the folder there if it isn't already.

On line 9, we now have a `File` object named `f` that is somehow connected to the text file on our computers.

On line 15 we created a second object. This is an ArrayList of `Strings` named `words`, and instantiating it didn't use any parameters.

On line 16, we create a third object. This one is a `Scanner` object, and it uses the `File` object from before as its parameter.

Finally, on line 26 we instantiate our fourth object: a `Random` object. Notice the pattern. If you want to instantiate an object called "Bob", you'd write code like this:

```java
Bob b = new Bob();
```

Or maybe:

```java
String s = "Robert";
Bob b = new Bob(s);
```

That's pretty much how Java creates objects. The keyword `new` is always involved, and the name of the class (twice) and some parens.

Now let's look at method calls. A *method* is a chunk of code inside an object that accomplishes a single purpose, and "calling" a method means asking the object to execute the code in that method for you.

On line 10, we call a "method" named `exists()` that is contained inside the `File` object `f`. This method will return a Boolean value (either `true` or `false`) depending on whether or not that file exists. The code that figures out how to do that is contained inside the library `java.io.File` that we imported. Make sense?

Line 11 features another method in the `File` class: `getName()`. It returns a String containing the name of the file associated with the object. If you skip down to line 18 you can also see a method named `getPath()` being called.

Line 19 calls the `hasNext()` method, which is in the `Scanner` class (our `Scanner` object is named *alpha*. It returns `true` if there's text in the file we haven't read yet.

The `next()` method reads a single String from the file and then on line 21 we call the `add()` method of ArrayLists to add that String to the list.

On line 23 we call the `close()` method, which closes the `Scanner` object so that we can't read from its file anymore.

Line 27 calls the `setSeed()` method of the `Random` class, and line 31 calls the `nextInt()` method of the same class. Line 31 also calls the `size()` method of ArrayLists, and line 32 calls `get()` to retrieve a single String out of the list.

Finally line 33 calls a method named `toLowerCase()`, which is part of the `String` class. Could you have figured that out if I hadn't told you? I hope so, because "toLowerCase()" looks like a method call, and the variable *s* is a `String`.

Okay, so that's enough for this exercise. The specific details of how this program works aren't important, but at this point you should have a good sense of how you do the following in Java:

1. Import libraries containing classes or objects
2. Instantiate (or "create") an object
3. Call methods on that object

Enough until next time.

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. Computers are pretty bad at being "random". They can only generate a random *sequence* of values, but that sequence is typically based on a "seed". If you use the same seed, the sequence of random numbers will be the same. (This is useful for debugging.) Change the seed on line `27` to something else (maybe `23213` or whatever). Then run the program again and confirm that although the output is different from before, it doesn't change when you run the program many times.

   Add a comment explaining what seed you picked and what the output was.

Exercise 2: Creating Your Own Single Objects

In the last chapter, we imported a few objects from Java's "standard library": the collection of classes and methods that are pre-built by the creators of the language.

In this chapter, we will create objects of our own, and each one will contain a single method.

Type up the following code and get it to compile. Save it in your 'javahard2' folder with a name of `OldMacDonald.java`.

OldMacDonald.java

```java
class Cow {
    public void moo() {
        System.out.println("Cow says moo.");
    }
}

class Pig {
    public void oink() {
        System.out.println("Pig says oink.");
    }
}

class Duck {
    public void quack() {
        System.out.println("Duck says quack.");
    }
}

public class OldMacDonald {
    public static void main( String[] args ) {

        Cow maudine = new Cow();
        Cow pauline = new Cow();
        maudine.moo();
        pauline.moo();

        Pig snowball = new Pig();
28          snowball.oink();
29          snowball.oink();
30
31          Duck ferdinand = new Duck();
32          ferdinand.quack();
33      }
34  }
```

What You Should See

```
java OldMacDonald
```

```
Cow says moo.
Cow says moo.
Pig says oink.
Pig says oink.
Duck says quack.
```

Lines 1-5 define an object called `Cow`. The definition of the Cow class includes the definition of a method called `moo()`. Note that on line 2 it says "public void moo()", not "public *static* void moo()". Except for main(), you won't be using the keyword *static* very much anymore.

Lines 7 through 11 define a class named `Pig`, containing an `oink()` method. And lines 13 through 17 define a class named `Duck`, which contains a `quack()` method.

Lines 19 to 34 define the class that matches the name of the Java file. Notice that in this file, the class `OldMacDonald` has the keyword *public* in front, but none of the other classes do. In Java, each file may only have one public class in it, and the name of that public class has to match the name of the file.

This class contains the `main()` method in it, which is where the Java Virtual Machine *begins* when executing a file. The `OldMacDonald` class is listed after the other classes in the file, but it would work the same if the classes were in a different order.

When we run this program, execution begins on the first line of the main() method. Any other code in the file will only execute if it gets called from inside `main()`.

Lines 22 and 23 instantiate two `Cow` objects. Lines 24 and 25 call the `moo()` method on behalf of each object. This causes execution to jump up to line 3, run the println() statement inside the method, and return back down below.

On line 27 we create an instance of a `Pig` object and then call its `oink()` method twice. And on line 31 we instantiate a `Duck` object and call its only method on the next line.

Then on line 33 we hit the close curly brace of the `main()` method, which typically means the end of the program.

Do you see? Defining your own objects isn't so hard, and calling their methods is pretty easy, too, once you've instantiated an object.

Study Drills

1. Try moving the entire definition of the `Duck` class below the `OldMacDonald` class. Does the code still compile and work? Answer in a comment.
2. Inside the main() method, instantiate another object and call its method. (It doesn't matter which of the three objects; just pick one.)

Exercise 3: Defining Objects in Separate Files

In the previous exercise, we defined three objects (actually four if you count the one that had main() in it), but they were all implemented in the same file. This is not typically how things are done. Usually Java puts the implementation for each class into its own file, and then there's *another* file that just holds the main() method that instantiates the objects and makes them do their thing. This class is often called the "driver" class, so usually I'll put the word "Driver" in the name of the file.

Type up the following code, and put each class into its own file, named as shown. Save them all in the same folder.

OldMacCow.java

```java
public class OldMacCow {
    public void moo() {
        System.out.println("Cow still says moo.");
    }
}
```

After you've typed in and saved `OldMacCow.java`, you should probably try to compile it to make sure you haven't made any mistakes before you move on.

OldMacDuck.java

```java
public class OldMacDuck {
    public void quack() {
        System.out.println("Duck still says quack.");
    }
}
```

Did you accidentally try to run this file or the first one? Neither one contains a main() method, and so executing it by itself won't work.

OldMacDriver.java

```java
public class OldMacDriver {
    public static void main( String[] args ) {
        OldMacCow maudine = new OldMacCow();
        OldMacCow pauline = new OldMacCow();
        maudine.moo();
        pauline.moo();

        OldMacDuck ferdinand = new OldMacDuck();
        ferdinand.quack();
    }
}
```

Ah, there's the main() method. Once done you can compile these a few ways.

```
javac OldMacCow.java
javac OldMacDuck.java
javac OldMacDriver.java
```

You can compile them one at a time. That works just fine.

```
javac OldMacCow.java OldMacDuck.java OldMacDriver.java
```

Although you have probably only used the Java compiler on one file at a time, it will happily compile as many files as you give it, in order from left to right.

If there's an error, though, you'll have to pay attention to the filename in the error message. For example:

```
OldMacCow.java:2: error: illegal start of type
```

This error message is on line 2 in the file `OldMacCow.java`, whereas the next mistake is on line 6 or earlier in the file `OldMacDuck.java` for this error message:

```
OldMacDuck.java:6: error: reached end of file while parsing
```

So just watch for that.

```
javac OldMac*.java
```

If the filenames you're trying to compile are similar, you can compile them all at once with something like this. The star/asterisk gets expanded by your terminal into all filenames in the current folder that begin with `OldMac` and which end in `.java`. (This includes the file `OldMacDonald.java` from the previous exercise. Which is fine, compiling doesn't "combine" the files in any way, it just converts each file one at a time into its own bytecode (.class) file.)

```
javac OldMacDriver.java
```

So, what magic is going on here? Only one file name? Well, what happens is that javac starts compiling `OldMacDriver.java`. On line 3 we refer to an object called OldMacCow. There's no object *called* that defined in this file. And there are no import statements to import a class called that, either.

So the Java compiler goes hunting. It knows it needs an object called OldMacCow, which would be implemented in a bytecode file named `OldMacCow.class`. If this file exists in the current folder, then it pulls the definitions from this bytecode file *automatically*! (This is a big deal for C++ programmers.)

And if there's no bytecode file in the current folder, it then looks for a source code file called `OldMacCow.java` that it can compile to *create* that bytecode file. If such a file is in the current folder, it'll just automatically compile it *for* you.

It does this for any objects referenced in the file you're compiling. If it can resolve all the dependencies itself, it'll do so. If not, it'll throw a compiler error about the undefined symbol it couldn't find.

So, to sum up, from here until the end of the book you should probably compile each file as you finish it to make sure there aren't any mistakes. *But* if you're lazy or just confident, it is usually okay to just compile the one file containing the main(), and let the compiler find the rest of the files for you.

What You Should See

As you might suspect, when executing the bytecode, you only need to run the file containing the main() method.

```
java OldMacDriver
```

```
Cow still says moo.
Cow still says moo.
Duck still says quack.
```

You’ll notice that the process of actually *instantiating* the objects or calling their methods isn’t any different. (See lines 3 through 9 in the driver file.) You just make an instance of an object, then call its method, just like before.

Hopefully this process of doing one program that is broken up into multiple files makes sense. Because that’s what we will be doing from here on out in the rest of the book!

(I’m not trying to be difficult; that’s just how object-oriented programming works. Code is broken up into classes/objects each in their own file and those objects are combined to make a working program. I’ll talk more about the reasons behind this in the chapters to come.)

Study Drills

1. Edit the message in the moo() method inside OldMacCow. Save the file with the changes but *do not* compile it! Then edit the message in the quack() method inside OldMacDuck(). Save the file but don’t compile it either. Then confirm that the single command `javac OldMacDriver.java` will compile all three files. Answer in a comment in the driver file how things worked out.

Exercise 4: Fields in an Object

So far we have only looked at methods inside of objects. But most objects have variables inside them, too, called “fields” (or sometimes “instance variables”).

This program will illustrate accessing fields in an object.

Type up this code and save it in its own file, named as indicated.

TVActor.java

```java
public class TVActor {
    String name;
    String role;
}
```

Then type up this one and save it in the same folder as the first file.

TVActorDriver.java

```java
public class TVActorDriver {
    public static void main( String[] args ) {
        TVActor a = new TVActor();
        a.name = "Thomas Middleditch";
        a.role = "Richard Hendricks";

        TVActor b = new TVActor();
        b.name = "Martin Starr";
        b.role = "Bertram Gilfoyle";

        TVActor c = new TVActor();
        c.name = "Kumail Nanjiani";
        c.role = "Dinesh Chugtai";

        System.out.println( a.name + " played " + a.role );
        System.out.println( b.name + " played " + b.role );
        System.out.println( c.name + " played " + c.role );
    }
}
```

Remember that you only *need* to compile the one file containing the main() method, though it is a good idea to test compiling each file as you finish it to make sure it’s correct before moving on.

What You Should See

```
javac TVActorDriver.java
java TVActorDriver
```

```
Thomas Middleditch played Richard Hendricks
Martin Starr played Bertram Gilfoyle
Kumail Nanjiani played Dinesh Chugtai
```

So the class `TVActor` contains two instance variables, and they are both Strings. The first variable is called *name* and the second is called *role*.

They are called "instance" variables because each *instance* (copy) of the object gets its own copies of the variables.

That is, just after line 11 is over, there are three instances of the TVActor class created. `public class TVActor` makes a pattern or recipe or blueprint of sorts, and then line 3 actually *sews together* the clothing or *cooks* the recipe or *builds* the structure when it instantiates the object.

And so the instance named *a* has a copy of the *name* variable and a copy of the *role* variable. We can put values into *a*'s copies of these variables as shown on lines 4 and 5, though we'll see later in the book that this is considered bad style.

Line 7 creates a second instance of the class, with its own copies of the instance variables.

And line 11 creates a third instance of the class, which also has *its* own copies of the variables. So by line 14, there are at least nine objects floating around in memory: three `TVActor` objects and six `String` objects (two per TVActor).

Study Drills

1. Add a third instance variable to the TVActor class, either a String, an int, or a double. Name it something suitable, then add code to the driver class to put values for each instance of the TVActor object.

   Also add code to print out the new field.

Exercise 5: Programming Paradigms

Before I get too far into the weeds of object-oriented programming (OOP), it might be useful to see the difference between OOP-style code and doing the same program in other programming paradigms.

I created a short program that does four things:

1. Allow the human to enter a message.
2. Reverse the order of the characters in the message.
3. "Camel-case" each word. That is, convert "Hello how are you" to "HelloHowAreYou".
4. Display the result.

First, here's the program using as much of a simple, monolithic style as Java will allow. You don't have to type this program in unless you really want to.

StringFunMonolith.java

```java
import java.util.Scanner;

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

        // input it
        System.out.print("Enter a message: ");
        String msg = keyboard.nextLine();

        // reverse it
        String rev = "";
        for ( int i=msg.length()-1; i>=0; i-- )
            rev += msg.substring(i,i+1);

        // camel-case it
        String lower = rev.toLowerCase();
        String[] words = lower.split(" ");
        String result = "";
        for ( String w : words )
            result += w.substring(0,1).toUpperCase() + w.substring(1);
22
23          // display it
24          System.out.println(result);
25      }
26  }
```

What You Should See

```
Enter a message: Hello how are you
UoyEraWohOlleh
```

So, lines 8-9 input the message, lines 12-14 reverse it, lines 17-21 camel-case it, and line 24 displays it. Don't worry too much if you don't understand the details of the camel-case part.

Next, I have coded the same program in a "functional" style. Functional style uses only functions with a few inputs and only one output each. The functions don't share information with each other except through their inputs and outputs.

Again, there's no sense typing up this version unless you want the practice. (It does work, though.)

StringFunFunctional.java

```java
import java.util.Scanner;

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

        // input it
        System.out.print("Enter a message: ");
        String msg = keyboard.nextLine();

        // reverse it
        msg = reverse(msg);

        // camel-case it
        msg = camelCase(msg);

        // display it
        System.out.println(msg);
    }
20
21      public static String reverse( String s ) {
22          String rev = "";
23          for ( int i=s.length()-1; i>=0; i-- )
24              rev += s.substring(i,i+1);
25
26          return rev;
27      }
28
29      public static String camelCase( String s ) {
30          String[] words = s.toLowerCase().split(" ");
31          String result = "";
32          for ( String w : words )
33              result += w.substring(0,1).toUpperCase() + w.substring(1);
34
35          return result;
36      }
37  }
```

Lines 1-9 are the same as the previous version, because it's kind of hard to get input from the human in Java any other way.

But you can see on line 12, the message (in the variable *msg*) is passed in to a function called `reverse`, and the result is put back into *msg*, overwriting the previous value. This is a bit more understandable.

And lines 21-27 are the `reverse()` function itself. It's the same code as lines 12-14 of the previous assignment, but there's a little extra setup to name the function and name the parameter and also an extra line to "return" the final result. Notice, though, that we get to call the input *s* instead of having to care that it's really called *msg* elsewhere. It's a bit nice to be able to call that variable whatever we want without caring what happens in other parts of the program.

Line 15 is the `camelCase` function call, and lines 29 through 36 are the function definition. Notice that on line 29 we were free to call the parameter *s* without caring about other parts of the program.

Is the variable in `main()` really called `s`?
Doesn't matter.
Is some other function already using a variable called `s`?
Doesn't matter.
What variable is the return value going into?
It doesn't matter. We can call it *result* or *rev* or whatever suits us.

Another nice thing about a functional style of programming is that since each function receives an input and returns an output, functions can be chained very compactly.

Here is the same functional version, but with the functions all nested inside each other.

StringFunFunctionalShort.java

```
 1 import java.util.Scanner;
 2 
 3 public class StringFunFunctionalShort {
 4     public static void main( String[] args ) {
 5         Scanner keyboard = new Scanner(System.in);
 6 
 7         System.out.print("Enter a message: ");
 8         System.out.println(camelCase(reverse(keyboard.nextLine())));
 9     }
10 
11     public static String reverse( String s ) {
12         String rev = "";
13         for ( int i=s.length()-1; i>=0; i-- )
14             rev += s.substring(i,i+1);
15 
16         return rev;
17     }
18 
19     public static String camelCase( String s ) {
20         String[] words = s.toLowerCase().split(" ");
21         String result = "";
22         for ( String w : words )
23             result += w.substring(0,1).toUpperCase() + w.substring(1);
24 
25         return result;
26     }
27 }
```

You read line 8 from the inside out. The inner-most thing happens first: `keyboard.nextLine()` is called. Once it's done, it returns a String, which we pass immediately to `reverse()`, then `camelCase()`, then `println()`.

Some programs are more difficult in a functional style, but when it works it's really nice and clean-looking.

> By the way, formulas in a spreadsheet program like Microsoft's *Excel*, Apple's *Numbers* or LibreOffice's *Calc* are programmed in a functional style. This is a tricky way to code, as you know if you've ever struggled to get one right!
>
> Graphics-processing shaders (like in OpenGL or Direct3D) are usually written in a functional style, too, and that makes them well-suited for parallel processing.

Okay, finally let's do this same little program in an object-oriented style. We'll use two files as usual: one containing our class/object, and one with the driver. These are the ones you should type in.

StringFunObject.java

```java
 1 public class StringFunObject {
 2 
 3     String message;
 4 
 5     public void setMessage( String s ) {
 6         message = s;
 7     }
 8 
 9     public String getMessage() {
10         return message;
11     }
12 
13     public void reverse() {
14         String rev = "";
15         for ( int i=message.length()-1; i>=0; i-- )
16             rev += message.substring(i,i+1);
17 
18         message = rev;
19     }
20 
21     public void camelCase() {
22         String[] words = message.toLowerCase().split(" ");
23         String result = "";
24         for ( String w : words )
25             result += w.substring(0,1).toUpperCase() + w.substring(1);
26 
27         message = result;
28     }
29 }
```

There's something new in this one. On line 3 there's an instance variable / field, just like you learned about in the previous exercise.

Notice that on line 6, there's a method that stores a copy of the parameter *s* into a variable named *message*. Where is this *message* declared? It's the field. We'll look more at this in later chapters, so don't worry too much about it for now.

Just remember for this code, any time that "message" is referenced, it's the instance variable. All the other variables are "local", which means they only exist inside the method in which they are defined.

So, here's the code for the driver. Type this one in, too.

StringFunOODriver.java

```
 1  import java.util.Scanner;
 2
 3  public class StringFunOODriver {
 4      public static void main( String[] args ) {
 5          Scanner keyboard = new Scanner(System.in);
 6
 7          // input it
 8          System.out.print("Enter a message: ");
 9          String msg = keyboard.nextLine();
10
11          StringFunObject sfo = new StringFunObject();
12          sfo.setMessage(msg);
13          sfo.reverse();
14          sfo.camelCase();
15
16          // display it
17          System.out.println( sfo.getMessage() );
18      }
19  }
```

What You Should See (Reminder)

```
Enter a message: Hello how are you
UoyEraWohOlleh
```

(There's the output again so you don't have to scroll up for it.)

This is very typical object-oriented code. Line 11 declares and instantiates an object. Line 12 calls the `setMessage()` method of that object, and passes the message into it as a parameter. Then the next few changes happen *inside* the object: the message gets reversed, then the message gets camel-cased.

Finally on line 17 of the driver we call the `getMessage()` method of the object, and it returns to us the modified String for printing.

Maybe you don't like the object-oriented style. Maybe you think the monolithic version is better, or maybe the functional version.

You know what? I agree with you. Object-oriented programming isn't a very good fit for a tiny program like this. OOP works best when the programs are large and complicated (like 10,000 lines of code or more).

Whenever *I* write a program to help me automate something annoying, I almost never code it in an object-oriented style if it's going to be only 500 lines of code or less.

I write it in a monolithic style if it's just going to be 10-50 lines long. I use functions when it's 50-500 lines long, and I start out object-oriented if it's going to be much bigger than that.

Unfortunately the rest of this book is going to be a little weird. I'm going to use the object-oriented style even for tiny 20-line programs. It'll be gross. You might not like it. You might think "This program would be *so much simpler* if he would just…."

But I can't teach you object-oriented programming using only nice huge 1,000 line perfect examples where OOP makes sense. (Well, I could, but this book would be about 800 pages longer and I wouldn't have finished writing it yet!) Instead I have to teach you OOP using small silly example programs where the OOP feels weird and forced *but* the programs are small enough to understand.

Once you're done with this book, you're free to code for the rest of your life in a non-object-oriented way. But, if someone dumps a 20,000 (or two-million!) line program on you that uses OOP just to have any *hope* of preventing bugs, then you'll have the tools to make sense of it.

Deal?

Study Drills

1. Using the `reverse()` method as a guide, add a method to the object-oriented version to remove half of the letters from the message. (It can be the first half, the last half, every other letter or whatever scheme you like.) Then add a call for that method to the driver.

Exercise 6: Accessing Fields in Methods

In this exercise, we are going to look in closer detail at the concept of an object having fields and methods that access those variables. This was introduced in the OOP-version of the previous exercise.

Here is the source code for the object, which will take a phrase and a number and produce a String with the specified number of copies of that message.

PhraseRepeater.java

```java
public class PhraseRepeater {

    String phrase;
    int repeats;

    public void setValues( String p, int r  ) {
        phrase = p;
        repeats = r;
    }

    public String getRepeatedPhrase() {
        String result = "";
        for ( int i=0; i<repeats; i++ )
            result += phrase;
        return result;
    }
}
```

This class has two fields / instance variables, a `String` named *phrase* and an `int` named *repeats*. Remember that if we were to instantiate several versions of this object in a driver, each instance of the object would have its own copies of both fields.

> "Instance variables" are variables defined in a class but outside of any method. A "field" is just a generic name for a member of a class. They mean pretty much the same thing, so I will use them interchangably in this book.

In case you didn't remember it from the previous exercise, these instance variables belong to the whole *class*, so all the methods in the object can access them.

Lines 6-9 implement a method called `setValues()`. This method receives two values from the outside world, a `String` we're going to call *p* and an integer we will call *r*.

On line 7 we store a copy of *p*'s value into our instance variable *phrase*, and on the next line we store a copy of *r*'s value into *repeats*. After line 9, the parameter variables *p* and *r* go away; those names don't have any meaning outside of this method. This method is `void`, so it doesn't return any value to the outside world.

Lines 11 through 16 have the implementation of a method called `getRepeatedPhrase()`, which builds up and then returns a copy of the value of a String.

On line 12 we start with a new String *result*, which is initialized to "the empty String" (that's what we call a String value with *no* characters in it).

Line 13 sets up a `for` loop that will execute its body *repeats*-many times. That is, if *repeats* has a 4 in it, the loop will run through four times. There are no curly braces in this loop, so the body of the loop is just a single line: `result += phrase;`, which adds a copy of the value of *phrase* to the end of whatever is already in *result*.

After the loop finishes, *result* now has several copies of the phrase in it. Notice that just like the previous method, this method was able to access the fields.

Finally, on line 15, a copy of the String in the local variable *result* is returned to the outside world. This method returns a String, which is why the first line of the method says `public String getRepeatedPhrase()` instead of `public void getRepeatedPhrase()`.

(Note that the method does *not* return the variable *result* itself; it merely returns a copy of the *value* that was in that variable.)

Okay, here's the driver code:

PhraseRepeaterDriver.java

```java
import java.util.Scanner;

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

        System.out.print("Enter a message: ");
        String msg = keyboard.nextLine();
        System.out.print("Number of times: ");
        int n = keyboard.nextInt();

        PhraseRepeater pr = new PhraseRepeater();
        pr.setValues(msg, n);
14          System.out.println( pr.getRepeatedPhrase() );
15      }
16  }
```

What You Should See

```
Enter a message: Boots and cats.
Number of times: 4
Boots and cats.Boots and cats.Boots and cats.Boots and cats.
```

The first ten lines of the driver are pretty straightforward if you've been coding in Java for a bit: they allow the human to enter in some values which get stored into local variables *msg* and *n*.

Then on line 12 we instantiate a single copy of the `PhraseRepeater` object and store a reference to it in the variable *pr*.

Then on line 13 we call that object's `setValues()` method, passing in copies of our local variables. The driver does not know that the instance variables inside the object are named *phrase* and *repeats*. The driver does not care what they are named.

The driver does not know that the parameters to the `setValues()` method will be called *p* and *r*. It makes no difference to the driver. Only the object cares what those variables are called.

This sort of not-caring is one of the reasons that object-oriented programming makes it easier to write very large complicated programs and debug them.

Anyway, the last useful line of the driver program is line 14, which calls the `getRepeatedPhrase()` method. That method returns a String, and the String that gets returned is fed to `println()` for... printing.

Notice that the `setValues()` method *changes* something inside the object. The fields in that object are different after the method call. Thus methods that change the internal state of an object in some way are often called "modifier methods" or "mutator methods".

On the other hand, the `getRepeatedPhrase()` method does *not* change anything about the internals of the object; both instance variables are used but they are not modified. But the method does return a value that lets you know something about the internal state of the object. Methods like this are often called "accessor methods" because they allow the driver to access the fields in some way.

Study Drills

1. On line 13 of the driver, change the order of the parameters in the method call. Does it compile? What happens and why? (Answer in a comment.)

Exercise 7: Encapsulation and Automated Testing

The OOP part of this exercise isn't any more difficult than the last exercise. Two fields are changed by a mutator method and accessed (but not changed) by an accessor method.

But in the driver… oh, you'll see.

SquareRootFinder.java

```java
public class SquareRootFinder {

    double n;
    int iterations;

    public void setNumber( double number ) {
        n = number;
        iterations = 7;
        if ( n < 10 )
            iterations++;
    }

    public double getRoot() {
        if ( n <  0 ) return Double.NaN;
        if ( n == 0 ) return 0;
        double x = n/4;
        for ( int i=0; i<iterations; i++ ) {
            x = (x+(n/x))/2.0;
        }
        return x;
    }
}
```

So, there's nothing new to see here. The `SquareRootFinder` class has two instance variables (*n* and *iterations*). The `setNumber()` method allows the user of the class to pass in a value that will be copied into the field *n*.

I guess I should mention that `NaN` stands for "not a number". It's a special value that you sometimes get in Java when you try to do something undefined like divide zero by zero or take the square root of a negative.

Then the `getRoot()` method does some complicated calculations to compute an estimate of the square root of that number. Does it work? Yes. *How* does it work? That's the thing about programming. Sometimes you won't know.

Go ahead and type in the driver code, then we'll continue this thought.

SquareRootDriver.java

```java
import java.util.Scanner;

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

        SquareRootFinder sqrt = new SquareRootFinder();

        do {
            System.out.print("Enter a number (or <=0 to quit): ");
            n = keyboard.nextDouble();

            if ( n > 0 ) {
                sqrt.setNumber(n);
                System.out.println( sqrt.getRoot() );
            }
        } while ( n > 0 );
    }
}
```

What You Should See

```
Enter a number (or <=0 to quit): 7
2.6457513110645907
Enter a number (or <=0 to quit): 81
9.0
Enter a number (or <=0 to quit): -99
```

I know that the square root of 4 is 2. (`2.0` when it's a `double`.) I know that the square root of 2 is 1.414-something. I check it on my calculator on my phone and it gives me "1.4142135624", which fits with what the driver gives me. I can type in a couple of other numbers and check them by hand and then say "uh, close enough", but how do I know?

Why does the `getRoot()` method start out *x* with n/4? Why is *iterations* set to 7 inside `setNumber()`? Why not 5 or 6 or 70? Why don't we let the user of the class pass in a value for *iterations* like we do for *n*?

The answer to all these questions is that sometimes "the user of the class" isn't the same person as "the creator of the class" and sometimes the user doesn't have the training and has better things to do or would probably mess these decisions up anyway.

For most programming tasks, there is more than one person involved. It is often better to let a single person say "here is an object, you use it in this way: put in a number here and the answer will come out here." This is a form of information hiding called "encapsulation", and it is one of the important concepts in object-oriented programming.

In encapsulation, an object has fields and forces the user of the object to use the methods provided instead of messing with the variables directly. In this example, `SquareRootFinder` allows the user of the class to pass in a value for *n* through the `setNumber()` method but does *not* allow them to pass in a value for *iterations*.

Make sense?

Okay, so how does the person who created the class know what value for *iterations* is "right"? I tested it. Like, a lot. Like, not just "type in a few numbers on the calculator and compare", but like so:

SquareRootTester.java

```java
public class SquareRootTester {
    public static void main( String[] args ) {

        SquareRootFinder sqrt = new SquareRootFinder();

        double max = 0, maxN = 0;
        double fakeroot, realroot, diff;

        System.out.print("Testing square root algorithm... ");
        for ( double n = 0; n<=2000; n += 0.01 ) {
            sqrt.setNumber(n);
            fakeroot = sqrt.getRoot();
            realroot = Math.sqrt(n);
            diff = Math.abs( fakeroot - realroot );
            if ( diff > max ) {
                max = diff;
                maxN = n;
            }
        }

        if ( max > 0.000001 ) {
            System.out.println("FAIL");
            System.out.println("Worst difference was " + max + " for " + maxN );
        }
        else
            System.out.println("PASS");
    }
}
```

What You Would See If You Ran the Tester

```
Testing square root algorithm... PASS
```

In my tester program, I compare the output of the `getRoot()` method with the "real" square root as computed by Java's built-in `Math.sqrt()`. I test every number from 0 to 2000, in increments of 0.01. For each number, I find the absolute difference between "my" square root and the "real" square root, and if the worst difference is more than 0.000001, then I throw an error.

Running this program over and over allowed me to test out different things. The variable x is my initial estimate of the square root; I had initially set x to n, but that starts to get too inaccurate as n gets bigger. Starting x at `n/2` is better, but then very small values of n get inaccurate.

The best compromise I found was to start with a guess of `n/4`, which gets me close enough within seven iterations for every number in the range. *Except* for values of n between 0 and 1 (where $\sqrt{n} > n$). I eventually gave up and just decided to give small numbers one extra iteration to compensate for my poor initial estimate in those cases.

Often (for well-designed / well-managed software, anyway) the person who creates a class or someone else on the team will design a "test suite" for that class. For example, SQLite (a database that can be embedded into other software) is famous for being *very* well tested. Quoting from "How SQLite Is Tested":

> The reliability and robustness of SQLite is achieved in part by thorough and careful testing.
>
> As of version 3.8.10, the SQLite library consists of approximately 94.2 KSLOC of C code. (KSLOC means thousands of "Source Lines Of Code" or, in other words, lines of code excluding blank lines and comments.) By comparison, the project has 971 times as much test code and test scripts - 91515.5 KSLOC.

Whenever they fix bugs or make improvements, the maintainers of SQLite run the full test suite to make sure they didn't accidentally break anything else! This is a good idea, and the kind of modular design that OOP forces on you makes testing like this possible.

Study Drills

1. In the tester, instead of the computing the *maximum* error in the range, compute the *total* error (the sum of all the errors). What is the sum? How does it change when the number of iterations is increased to 8? Answer in a comment.

Exercise 8: Failure to Encapsulate

In the previous exercise, we looked at extreme testing, and how encapsulation makes that possible. In this one, we'll see some of the tradeoffs that can be made with fields and methods.

SphereCalc.java

```java
public class SphereCalc {
    double radius;

    public void setRadius( double r ) {
        radius = r;
    }

    public double getRadius() {
        return radius;
    }

    public double getSurfaceArea() {
        return 4*Math.PI*radius*radius;
    }

    public double getVolume() {
        return 4*Math.PI*Math.pow(radius,3) / 3.0;
    }
}
```

This object is very similar to the ones in the last couple of exercises. A single instance variable this time, one mutator method (`setRadius()`) and three accessor methods. (The surface area of a sphere is $4\pi r^2$, and the volume of a sphere is $\frac{4}{3}\pi r^3$.)

SphereCalcTester.java

```java
public class SphereCalcTester {
    public static void main( String[] args ) {

        SphereCalc c = new SphereCalc();

        c.setRadius(5);
        if ( isNear(c.getSurfaceArea(), 314.159265359) )
            System.out.println("PASS: surfaceArea for " + c.getRadius());
        else
            System.out.println("FAIL: surfaceArea not what was expected!");
        if ( isNear(c.getVolume(), 523.598775598) )
            System.out.println("PASS: volume for " + c.getRadius());
        else
            System.out.println("FAIL: volume not what was expected!");

        c.setRadius(0.1);
        if ( isNear(c.getSurfaceArea(), 0.125663706) )
            System.out.println("PASS: surfaceArea for " + c.getRadius());
        else
            System.out.println("FAIL: surfaceArea not what was expected!");
        if ( isNear(c.getVolume(), 4.18879E-3) )
            System.out.println("PASS: volume for " + c.getRadius());
        else
            System.out.println("FAIL: volume not what was expected!");


    }

    public static boolean isNear( double a, double b ) {
        return Math.abs(a-b) < 1E-9;
    }
}
```

What You Should See

```
PASS: surfaceArea for 5.0
PASS: volume for 5.0
```

```
PASS: surfaceArea for 0.1
PASS: volume for 0.1
```

This is clearly a *tester* and not just a simple driver program; I have tests that are passing or failing. Writing this was pretty annoying, but I wanted to show you the idea without making it too crazy, so I had to use my calculator with a couple of test cases to see what they ought to be. In a future exercise we'll see a much better way to do a lot of tests like this without repeating so much code, but it's too complicated for now.

Line 4 instantiates a SphereCalc object, then line 6 sets its radius to `5`.

Starting down on line 29, there's a little helper function I wrote. It receives two `doubles` and returns `true` if the absolute value of their difference is very small (smaller than 1.0×10^{-9}). It's best to avoid using just == on two floating-point values since sometimes repeating decimals or slight differences in rounding will make two values that *ought* to be the same slightly different.

(Instead of `isNear()` I probably could have called the function `isVeryCloseToEqual()` but I didn't feel like typing that more than once.)

So lines 7 through 14 just call the methods from `SphereCalc` and make sure they return numbers close enough to the expected values. If so, we print out "PASS" and if not we print out "FAIL" and a little bit of detail. Normally you'd want to print out more information with the failure (like which radius failed and what the expected value was and what you got instead), but I didn't want to clutter up the code.

Oh, and in case you've never seen it before, an `E` inside a floating-point number means "times ten to the". On line 21, `4.18879E-3` means 4.18879×10^{-3}) A.K.A. `0.00418879`.

Okay, so now let's look at an slightly different way of splitting up the work in the `SphereCalc` object. (You'll need to type this one in, too, if you're going to do the Study Drill.)

SphereCalc2.java

```java
public class SphereCalc2 {
    double radius, area, volume;

    public void setRadius( double r ) {
        radius = r;
        area = 4*Math.PI*r*r;
        volume = 4*Math.PI*Math.pow(r,3) / 3.0;
    }

    public double getRadius() {
        return radius;
    }

    public double getSurfaceArea() {
        return area;
    }

    public double getVolume() {
        return volume;
    }
}
```

`SphereCalc2` has three fields instead of just one. And inside the `setRadius()` mutator method, it doesn't *just* set the radius, it also goes ahead and computes the surface area and volume, too.

There's a trade-off here. Each instance of a `SphereCalc2` object would take up slightly more memory than each `SphereCalc` object, because of the extra fields, and creating a instance of a `SphereCalc2` object would take slightly longer than instantiating a `SphereCalc` object because it does more calculations up front.

However, if you had a `SphereCalc` object and you called `getVolume()` over and over again in a loop or something, it would have to do that calculation over and over. Whereas a `SphereCalc2` object has already *done* the calculation and just gets to return that single value over and over.

Which approach is better? You'd have to run tests and see how your object is being used to find out.

`SphereCalc2` has one serious problem, however. Well, it's more like a *vulnerability* than a problem. When someone is using a `SphereCalc2` object and they want to change the radius, we *expect* them to use the provided `setRadius()` method. We *hope* that's what they will do.

But as you might recall from `TVActorDriver.java` way back in Exercise 4, a driver class can access instance variables directly. At least, the way we've been writing them up to this point.

What's to prevent someone from writing code like this?

```java
SphereCalc2 sph = new SphereCalc2();
sph.setRadius(5);
sph.radius = 7;   // OH NOES!
System.out.println( sph.getVolume() );
```

Now, it probably wouldn't look so evil. It might be like on line 16 on the tester. Instead of writing:

```java
c.setRadius(0.1);  // <-- why write this...
c.radius = 0.1;    // <-- when it's SO much easier to write this?
```

It's more efficient, right? Who wants to call a method when you can just put a value in a variable?!? Not this guy!

Anyway, hopefully that illustrates the "problem". For the solution, you'll have to come back in the next exercise.

Study Drills

1. Modify the tester to use SphereCalc2 objects instead of SphereCalc objects, then add code to change the *radius* variable directly instead of calling `setRadius()`. Confirm the tests now fail even though the radius is right. (This is bad.)

Exercise 9: Private Fields and Constructors

Assuming you did the last exercise, you have seen that some classes won't work properly if you change their fields directly instead of going through their methods. In this exercise, you'll learn how to put a *stop* to that.

You'll also learn about something that'll make it easier for others to *use* your classes and make it safer, too.

SphereCalc3.java

```
 1 public class SphereCalc3 {
 2     private double radius, area, volume;
 3 
 4     public void setRadius( double r ) {
 5         radius = r;
 6         area = 4*Math.PI*r*r;
 7         volume = 4*Math.PI*Math.pow(r,3) / 3.0;
 8     }
 9 
10     public double getRadius()      { return radius; }
11     public double getSurfaceArea() { return area;   }
12     public double getVolume()      { return volume; }
13 }
```

This is basically SphereCalc2, just shrunk down. When all you're doing in a method is returning the value of a single variable, Java programmers often write the "getter" methods all on one line like I did in lines 10-12.

So the only *interesting* change is at the beginning of line 2: the keyword `private`.

You've been making things `public` since your first Java program ever thanks to "public static void main", so maybe you suspected.

Instance variables are typically made *private*. Almost always, as a matter of fact. (You can designate methods as private instead of public, too, but we won't see an example of that for a while.)

Private means "DON'T TOUCH!" Any private variable can't be accessed in any way outside of the class where it is defined.

Inside the class, private variables work just like the fields we have been using; any method inside the class is free to change or access private variables just the same.

> “Public” and “private” are called “access level modifiers” in Java. When you leave them out (like we’ve been doing with our fields since exercise 4) the default access is called “package-private”, which means that they’re accessible to anything inside the same “package”. All the classes we’ve written so far are all inside the same package, but we won’t do anything about that until close to the end of the book.
>
> So for our purposes, “public” variables and variables with *no* package modifier are equivalent: they can be accessed or changed from *outside* their class. And that’s a bad thing.

Java programmers are typically pretty strict about making fields private. In fact, on the Advanced Placement Computer Science exam, failing to mark an instance variable as private is so serious that it can cost you more than 10% of your score on a question, even if every other part of your solution is perfect!

Anyway, back to the main point. Now that the fields are private, code that uses the `SphereCalc3` class has no choice; they can *only* change the radius through the `setRadius()` method. Attempts to do it directly won’t even compile.

```
SphereCalc3 sph = new SphereCalc3();
sph.radius = 7;   // <-- This won't even compile.
sph.setRadius(7); // This works just fine, of course.
System.out.println( sph.radius ); // still won't compile
```

As you can see, this applies even if you’re not trying to *change* the instance variable. `private` doesn’t just prevent modifying the field, it prevents accessing it, too. That’s why you have to write public “getter” methods for every variable you want accessible.

Some programming languages (like C#) have a slightly different way of dealing with this problem; you can mark variables as read-only so they can’t be *changed* from outside the class (only through methods) but they can still be read. Other languages have a way of making it *look* like you’re accessing a variable directly, but they’re really secretly running a method to set or read the variable.

In Java, however, private fields with setters and getters are the only good solution.

Now, one more potential problem before we move on. It is a little bit annoying to have to always remember to call the setter method before doing anything else. Look at some examples from the past several exercises:

```
SphereCalc sph = new SphereCalc();
sph.setRadius(5);
// now it's safe to use the other methods in SphereCalc
//
SquareRootFinder srf = new SquareRootFinder();
srf.setNumber(n);
// now it's safe to use the other methods in SquareRootFinder
//
PhraseRepeater pr = new PhraseRepeater();
pr.setValues(msg, n);
// now it's safe to use the other methods in PhraseRepeater
//
StringFunObject sfo = new StringFunObject();
sfo.setMessage(msg);
// now it's safe to use the other methods in StringFunObject
```

Not only is this annoying, it's not *safe*. I won't make you do it in the Study Drills, but if you accidentally forgot to call `setNumber()` or `setValues()` the driver would *still compile*, but it wouldn't work properly. And as much as I hate compile-time errors, I hate it a **lot** more when I have code that compiles but doesn't work.

Fortunately, there's a solution! A special sort-of setter method called a "constructor". Here's an example.

SphereCalc4.java

```
public class SphereCalc4 {
    private double radius, area, volume;

    public SphereCalc4( double r ) {
        radius = r;
        area = 4*Math.PI*r*r;
        volume = 4*Math.PI*Math.pow(r,3) / 3.0;
    }

    public void setRadius( double r ) {
        radius = r;
        area = 4*Math.PI*r*r;
        volume = 4*Math.PI*Math.pow(r,3) / 3.0;
    }

    public double getRadius()      { return radius; }
    public double getSurfaceArea() { return area;   }
18      public double getVolume()       { return volume; }
19  }
```

Lines 4 through 8 are the implementation of the constructor. Notice on line 4 that unlike the `setRadius()` setter/mutator method, the constructor is *not* `void`. Constructors have no return type specifier at all; it's just missing.

Also notice that the constructor has the same name as the class itself. This is required. If you do those two things, then instead of having to remember to call some special method to pass in initial values for the instance variables, you get to pass them in **while you're instantiating the object**. Which you would have to do anyway! Like so:

```
SphereCalc4 sc = new SphereCalc4( 5 );
// it's safe right away to use the other methods in SphereCalc
SquareRootFinder srf = new SquareRootFinder(n);
// ditto
PhraseRepeater pr = new PhraseRepeater(msg, n);
StringFunObject sfo = new StringFunObject(msg);
```

This makes a little more work when implementing a class, because you usually have to write a constructor and *also* write your setter methods. But it makes it easier to work with your object and safer, too.

Okay, that's enough for now. We'll see plenty more constructors in the chapters to come.

Study Drills

1. In SphereCalc4, edit the code inside the constructor so that it *calls* `setRadius()` instead of duplicating its code.
2. Save a copy of SphereCalcTester.java as SphereCalcTester4.java and change the objects from SphereCalc2 objects to SphereCalc4 objects. (You'll have to pass in the first radius in the instantiation.) Add several lines of code to confirm that since SphereCalc4 has private fields, you can't access them directly at all.

Exercise 10: Automated Testing with Arrays

Two exercises ago, `SphereCalcTester` did a decent job testing our object, but it took a lot of code for each test, and there was a lot of repeated code. It is important to make testing code as easy as possible to write and to automatically run. Otherwise, you might be tempted to *not* test your code, and that doesn't lead anywhere good.

So here is an example of a tester for `SphereCalc4` that makes it much easier to add additional tests without adding any extra code!

This code uses arrays of doubles to hold the expected inputs and outputs, and they are in the same order in each array so that `areas[0]` holds the expected area output for radius `inputs[0]` and `volumes[0]` holds the corresponding expected volume. Arrays used like this are called "parallel" arrays.

This isn't the *absolute* best way to do this, but it's good enough for now. We'll see an even better testing technique later in the book.

BetterTesting.java

```java
public class BetterTesting {
    public static void main( String[] args ) {

        double[] inputs = {
            5,
            0.1,
            3.3,
            20000,
            8
        };
        double[] areas  = {
            314.159265359,
            0.125663706,
            136.84777599,
            5026548245.743669104,
            804.247719319
        };
        double[] volumes = {
            523.598775598,
            4.18879E-3,
            150.532553589,
            3.3510321638291125E13,
            2144.660584851
        };
        int passed = 0;
        double r, a, v, A, V;

        SphereCalc4 c = new SphereCalc4(0);
        for ( int i=0; i<inputs.length; i++ ) {
            r = inputs[i];
            a = areas[i];
            v = volumes[i];

            c.setRadius(r);
            A = c.getSurfaceArea();
            V = c.getVolume();
            if ( isNear(A, a) )
                passed++;
            else {
                System.out.print("FAIL: surfaceArea for radius " + r );
                System.out.println("-- Expected " + a + ", got " + A);
            }
            if ( isNear(V, v) )
                passed++;
            else {
                System.out.print("FAIL: volume for radius " + r );
                System.out.println("-- Expected " + v + ", got " + V);
            }
        }

        if ( passed == 2*inputs.length )
            System.out.println("PASS: All tests passed.");
    }

    public static boolean isNear( double a, double b ) {
        return Math.abs(a-b) < 1E-9;
    }
}
```

What You Should See

```
PASS: All tests passed.
```

Lines 4 through 24 just contain the values for inputs and outputs. I like to list them one per line like this, but Java doesn't care if you put them all on one line. If you *do* put them all on one line, it'd probably be good for your sanity if you add extra spaces so that the corresponding entries line up, like so:

```
double[] inputs  = {   5,            0.1,            3.3,          // etc
double[] areas   = { 314.159265359, 0.125663706, 136.84777599,  // etc
double[] volumes = { 523.598775598, 4.18879E-3,  150.532553589, // etc
```

On line 28 we just create a single `SphereCalc4` object, which will be reused each time. Then there's a loop through each value in the arrays. NOTE: If you accidentally make the arrays different lengths, then this code might blow up. That's one of the problems with parallel arrays.

On lines 30 through 36 we pull out the expected radius, area and volume and put them into nicely-named but easy-to-type variables, and then tell the object to *use* that radius and get the computed area and volume from our object. So *a* is the expected area, and *A* is the actual area according to our object.

Then we can just use the same `isNear()` function from earlier together with `if` statements to see if what we got matches what was expected. We increment a variable for each "PASS" but don't bother printing anything. If there's a failure, we print an error message.

Once the loop is over, there should be twice as many "passes" as inputs, so we check that and print a summary message if everything is good.

Not too bad, huh? Adding more tests is as easy as just adding more numbers to the arrays, but none of the other code has to change. And it's easy to tell when everything turned out as expected and easy to see specifically what went wrong when something isn't right.

Study Drills

1. Change one of the digits in one of the input or output values and see how the program shows different output.
2. Break one of the formulas in `SphereCalc4` and see how the tester shows something different. How could your tester distinguish between bad test cases (like Study Drill #1) and a wrong formula? Answer in a comment in the tester program.

Exercise 11: Public vs Private vs Unspecified

We looked at making fields private a couple of exercises back, but there were a lot of other things going on, too. So this exercise focuses on just that.

FieldAccess.java

```java
public class FieldAccess {

    public String first;
    private String last;
    String nick;

    public FieldAccess() {
        first = last = nick = "";
    }

    public FieldAccess( String f, String l, String n ) {
        first = f;
        last = l;
        nick = n;
    }

    public void setFirst( String s ) {
        first = s;
    }

    public void setLast( String s ) {
        last = s;
    }

    public void setNick( String s ) {
        nick = s;
    }

    public String getFirst() { return first; }
    public String getLast()  { return last; }
    public String getNick()  { return nick; }

    public String getFullName() {
        return first + " \"" + nick + "\" " + last;
    }
}
```

There are three instance variables (A.K.A. “fields”) in this object. One is public, one is private, and one has an unspecified access level, which means it defaults to something called “package-private”.

You’ll notice that this object also has *two* constructors. The first constructor (lines 7-9) has no parameters, so it’s called the “default” constructor or sometimes the “zero-argument” constructor. The second constructor runs from line 11 through 14 and has three String parameters.

That means whenever the driver instantiates a `FieldAccess` object, it can either do it with no arguments like `new FieldAccess()`, or it must pass in three Strings.

You’ll also see the usual getters and setters for the fields.

FieldAccessDriver.java

```java
public class FieldAccessDriver {
    public static void main( String[] args ) {
        FieldAccess j = new FieldAccess("Robert", "Parker", "Butch");
        System.out.println(j.getFullName());

        j.setLast("Elliott");
        j.setFirst("Samuel");
        j.setNick("Sam");
        System.out.println(j.getFullName());

        j.first = "Avery";
        // j.last = "Markham";
        System.out.println(j.nick);
    }
}
```

What You Should See

```
Robert "Butch" Parker
Samuel "Sam" Elliott
Sam
```

On line 3 of the driver we instantiate a `FieldAccess` object in the expected way: call the constructor and pass in three strings. Line 4 shows that it has been constructed correctly.

Lines 6 through 8 change the fields “properly”: by using the setter (mutator) methods.

On lines 11 through 13 we access all three fields “incorrectly”: directly. The field *first* actually works; it is public, after all. Accessing *last* directly wouldn’t even compile, which is why it’s commented out.

And printing out the *nick* field also works. Remember that a field without an access-level modifier defaults to package-private, which means that any code defined in the same package can touch that variable. We haven’t learned about packages yet (and won’t for a while yet), so all the programs you have written so far are all in the same (unnamed) package.

Hopefully that was a pretty simple exercise, and there weren’t any surprises.

Study Drills

1. Modify lines 11 through 13 in the driver so that they change/access the values using the proper methods.

Exercise 12: Reviewing Constructors

There's nothing really new in this exercise, so if you're totally comfortable with constructors and private instance variables, then feel free to skip this one.

But constructors are very important, so I want to cover them one more time before moving on to a new topic.

Rectangle.java

```java
public class Rectangle {
    private int length, width;

    public Rectangle() {
        length = width = 0;
    }

    public Rectangle( int l, int w ) {
        length = l;
        width = w;
    }

    public int getArea() {
        return length*width;
    }
}
```

This (boring) `Rectangle` class has two private fields. (In the future, I probably won't bother to write "private fields"; calling them "fields" pretty much implies that they will be private. That's just how Java programmers do things. I probably should have written getters and setters for them, but this exercise is just focused on constructors so I left them out.)

There are two constructors. There are three things you should remember about constructors.

1. They have the same name as the class (`Rectangle` in this case).
2. They do not have a return type – not even "void".
3. Their "job" is to make sure all necessary setup has been done. This means initializing all the instance variables, but sometimes other stuff happens, too.

I was going to add "Constructors must be public," but that's not true. They are *usually* public but sometimes you want a constructor but don't want people to be able to call it, so you make one of the constructors private or something.

RectangleDriver.java

```
 1 public class RectangleDriver {
 2     public static void main( String[] args ) {
 3         // Rectangle r = new Rectangle();
 4         // r.length = 10;
 5         // r.width = 5;
 6 
 7         Rectangle r = new Rectangle(10, 5);
 8         System.out.println("The area is " + r.getArea());
 9     }
10 }
```

Lines 3-5 in the driver are commented out, but they show what you would have done to instantiate the object several exercises ago. You must construct the object itself (line 3), and then put values into all the fields.

Now that our fields are private, and now that we have constructors, we can accomplish all this in just a single line in the driver. This is shown on line 7.

The first thing that happens on line 7 is the left-hand side of the equal sign. The compiler creates a `Rectangle` object and names it *r*. At first, it doesn't have an object in it.

Then the right-hand side of the equal sign is done: it calls the parameter constructor, passing in `10` and `5` as parameters. Because the `10` is first, a copy gets put into the first parameter (the `int` *l*). Then a copy of the `5` gets passed into the second parameter (the `int` named *w*).

Secretly behind the scenes just before line 9 our `Rectangle` object is actually instantiated in memory. Then on lines 9 and 10 the constructor copies the values from the parameters into the instance variables.

Once the constructor ends on line 11 one other thing happens behind the scenes that *doesn't happen in regular methods: a reference to the `Rectangle` object is returned back to the driver. (We'll learn more about this in a later exercise.)

This sends us back to line 7 of the driver, where the right-hand side has just completed. So finally the "equal sign" part of the line happens; (a reference to) the object returned from the constructor gets stored into the variable on the left hand side (*r*).

So at this point line 7 is completely done, and the object has been instantiated and the fields have values.

That's a lot of doing for one line of code, eh? That's why constructors are nice, actually. It's a little more complicated than doing all those things manually, but it's nicer for the person using our class and it's safer since it makes *certain* all that setup has been completed before the object gets used.

What You Should See

```
The area is 50
```

Cool?

Study Drills

1. In the driver, add code to instantiate two more `Rectangle` objects, and print out their areas.

Buy the Full Book!

If you really made it this far, you have what it takes to finish the book.

So buy it! One semester of a college course would cost you hundreds of dollars, and hiring a personal tutor would cost even more.

Aren't you worth it?

– Graham Mitchell