

# **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 2016-08-14



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

© 2015 - 2016 Graham Mitchell

## Also By **Graham Mitchell**

Learn Java the Hard Way

# Contents

Acknowledgements . . . . .	i
Preface: Learning by Doing . . . . .	ii
Introduction: Object-Oriented Java . . . . .	iii
Exercise 0: The Setup . . . . .	1
Exercise 1: Working With Objects . . . . .	17
Exercise 2: Creating Your Own Single Objects . . . . .	23
Exercise 3: Defining Objects in Separate Files . . . . .	26
Exercise 4: Fields in an Object . . . . .	30
Exercise 5: Programming Paradigms . . . . .	32
Exercise 6: Accessing Fields in Methods . . . . .	39
Exercise 7: Encapsulation and Automated Testing . . . . .	42
Exercise 8: Failure to Encapsulate . . . . .	47
Exercise 9: Private Fields and Constructors . . . . .	52
Exercise 10: Automated Testing with Arrays . . . . .	56
Exercise 11: Public vs Private vs Unspecified . . . . .	59
Exercise 12: Reviewing Constructors . . . . .	62
Exercise 13: Default Values for Fields . . . . .	65
Exercise 14: toString and this . . . . .	68
Exercise 15: Noughts and Crosses / Extreme Testing . . . . .	73
Exercise 16: Introduction to ArrayLists . . . . .	84

## CONTENTS

Exercise 17: Word Counter Using an ArrayList . . . . .	89
Exercise 18: Primitive Variables in Memory . . . . .	92
Exercise 19: Reference Variables in Memory . . . . .	96
Exercise 20: Lists of Primitive Values . . . . .	102
Exercise 21: Generics vs. Casts . . . . .	106
Exercise 22: Object-Oriented Design and Efficiency . . . . .	112
Exercise 23: Writing a Silly Class with Generics . . . . .	118
Exercise 24: Writing a Useful Class with Generics . . . . .	123
Exercise 25: Sorting and Complexity . . . . .	133
Exercise 26: Sorting Speeds - Primitives vs Objects . . . . .	143
Exercise 27: Static Variables and Static Methods . . . . .	149
Exercise 28: Popular Static Methods in Java . . . . .	153
Exercise 29: Importing Static Class Members . . . . .	157
Exercise 30: References as Parameters . . . . .	160
Exercise 31: Java Strings Are Immutable . . . . .	165
Exercise 32: Parameters vs Properties . . . . .	169
Exercise 33: Basic Inheritance . . . . .	172
Exercise 34: How Fields Are Inherited . . . . .	177
Exercise 35: “Useful” Inheritance - A Game Board . . . . .	180
Exercise 36: A Game Called Breakthrough . . . . .	191
Exercise 37: Two Kinds of Equality . . . . .	200
Exercise 38: Implementing Interfaces . . . . .	204
Exercise 39: The Comparable Interface . . . . .	208
Exercise 40: List and Map . . . . .	215
Exercise 41: Implementing Several Interfaces . . . . .	220

## CONTENTS

Exercise 42: DropGame and Assertions . . . . .	225
Exercise 43: Abstract Classes and Final Methods . . . . .	232
Exercise 44: Packages . . . . .	236
Exercise 45: Creating a JAR File . . . . .	240
Exercise 46: A Simple Graphical Window . . . . .	244
Exercise 47: An Interactive Window . . . . .	249
Exercise 48: Using (F)XML to Define Your Interface . . . . .	254
Exercise 49: Canvas Basics . . . . .	259
Exercise 50: Getting Mouse Input . . . . .	263
Exercise 51: More Complex Mouse Interaction . . . . .	267
Exercise 52: Animation . . . . .	272
Exercise 53: Handling Keypress Events . . . . .	277
Exercise 54: Graphical Noughts and Crosses . . . . .	283
Exercise 55: Graphical Drop Game . . . . .	290
Next Steps . . . . .	297

# Acknowledgements

To God, who gives me the ability to create.

To Deanna, who believed I could finish this even when I wasn't sure.

To the hundreds of people who purchased my first book; this second book wouldn't exist without you!

And to Jesus, who teaches me grace and humility every day. May I depend on Him more and more each day.

# 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](#)<sup>1</sup> 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.

---

<sup>1</sup><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

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

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

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

Unless otherwise stated, all content is copyright 2015-2016 Graham Mitchell.

# 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](http://conqueringthecommandline.com)<sup>2</sup>.

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.

---

<sup>2</sup><http://conqueringthecommandline.com/book/basics>


# Windows

## Installing a Decent Text Editor (Notepad++)

1. Go to [notepad-plus-plus.org](http://notepad-plus-plus.org)<sup>3</sup> 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.

---

<sup>3</sup><http://notepad-plus-plus.org/>

## Installing the Java Development Kit (JDK)

1. Go to [Oracle's Java SE downloads page](http://www.oracle.com/technetwork/java/javase/downloads/index.html)<sup>4</sup> 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.

---

<sup>4</sup><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” interchangeably. 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](http://barebones.com)<sup>5</sup> 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 Applications 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](http://www.oracle.com/technetwork/java/javase/downloads/index.html)<sup>6</sup> with your web browser.

---

<sup>5</sup><http://www.barebones.com/products/textwrangler/>

<sup>6</sup><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” interchangeably. 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](http://www.oracle.com/technetwork/java/javase/downloads/index.html)<sup>7</sup>.

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”.

---

<sup>7</sup><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” interchangeably. 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`

---


```
1  import java.io.File;
2  import java.util.ArrayList;
3  import java.util.Random;
4  import java.util.Scanner;
5
6  public class WorkingWithObjects {
7      public static void main( String[] args ) throws Exception {
8          File f = new File("datafiles/phonetic-alphabet.txt");
9
10         if ( f.exists() == false ) {
11             System.out.println( f.getName() + " not found in this folder. :(");
12             System.exit(1);
13         }
14
```

```
15     ArrayList<String> words = new ArrayList<String>();
16     Scanner alpha = new Scanner(f);
17
18     System.out.print("Reading words from \"" + f.getPath() + "\"... ");
19     while ( alpha.hasNext() ) {
20         String w = alpha.next();
21         words.add(w);
22     }
23     alpha.close();
24     System.out.print("done.\n\t");
25
26     Random rng = new Random();
27     rng.setSeed(12345);
28     // rng.setSeed(23213);
29
30     for ( int n=0; n<3; n++ ) {
31         int i = rng.nextInt( words.size() );
32         String s = words.get(i);
33         System.out.print( s.toLowerCase() + " " );
34     }
35     System.out.println();
36 }
37 }
```

---

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:

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

Or maybe:

```
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

---

```
1  class Cow {
2      public void moo() {
3          System.out.println("Cow says moo.");
4      }
5  }
6
7  class Pig {
8      public void oink() {
9          System.out.println("Pig says oink.");
10     }
11 }
12
13 class Duck {
14     public void quack() {
15         System.out.println("Duck says quack.");
16     }
17 }
18
19 public class OldMacDonald {
20     public static void main( String[] args ) {
21
22         Cow maudine = new Cow();
23         Cow pauline = new Cow();
24         maudine.moo();
25         pauline.moo();
26
27         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 `Old MacDonald` 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

---

```
1 public class OldMacCow {
2     public void moo() {
3         System.out.println("Cow still says moo.");
4     }
5 }
```

---

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

---

```
1 public class OldMacDuck {
2     public void quack() {
3         System.out.println("Duck still says quack.");
4     }
5 }
```

---

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

```
1 public class OldMacDriver {  
2     public static void main( String[] args ) {  
3         OldMacCow maudine = new OldMacCow();  
4         OldMacCow pauline = new OldMacCow();  
5         maudine.moos();  
6         pauline.moos();  
7  
8         OldMacDuck ferdinand = new OldMacDuck();  
9         ferdinand.quack();  
10    }  
11 }
```

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

---

```
1 public class TVActor {  
2     String name;  
3     String role;  
4 }
```

---

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

TVActorDriver.java

---

```
1 public class TVActorDriver {  
2     public static void main( String[] args ) {  
3         TVActor a = new TVActor();  
4         a.name = "Thomas Middleditch";  
5         a.role = "Richard Hendricks";  
6  
7         TVActor b = new TVActor();  
8         b.name = "Martin Starr";  
9         b.role = "Bertram Gilfoyle";  
10  
11        TVActor c = new TVActor();  
12        c.name = "Kumail Nanjiani";  
13        c.role = "Dinesh Chugtai";  
14  
15        System.out.println( a.name + " played " + a.role );  
16        System.out.println( b.name + " played " + b.role );  
17        System.out.println( c.name + " played " + c.role );  
18    }  
19 }
```

---

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

---

```
1  import java.util.Scanner;
2
3  public class StringFunMonolith {
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         // reverse it
12         String rev = "";
13         for ( int i=msg.length()-1; i>=0; i-- )
14             rev += msg.substring(i,i+1);
15
16         // camel-case it
17         String lower = rev.toLowerCase();
18         String[] words = lower.split(" ");
19         String result = "";
20         for ( String w : words )
21             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: ^C

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

---

```
1  import java.util.Scanner;
2
3  public class StringFunFunctional {
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         // reverse it
12         msg = reverse(msg);
13
14         // camel-case it
15         msg = camelCase(msg);
16
17         // display it
18         System.out.println(msg);
19     }
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

---

```
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: ^C

(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

---

```
1 public class PhraseRepeater {
2
3     String phrase;
4     int repeats;
5
6     public void setValues( String p, int r ) {
7         phrase = p;
8         repeats = r;
9     }
10
11    public String getRepeatedPhrase() {
12        String result = "";
13        for ( int i=0; i<repeats; i++ )
14            result += phrase;
15        return result;
16    }
17 }
```

---

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 interchangeably 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

```
1 import java.util.Scanner;
2
3 public class PhraseRepeaterDriver {
4     public static void main( String[] args ) {
5         Scanner keyboard = new Scanner(System.in);
6
7         System.out.print("Enter a message: ");
8         String msg = keyboard.nextLine();
9         System.out.print("Number of times: ");
10        int n = keyboard.nextInt();
11
12        PhraseRepeater pr = new PhraseRepeater();
13        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

---

```
1 public class SquareRootFinder {
2
3     double n;
4     int iterations;
5
6     public void setNumber( double number ) {
7         n = number;
8         iterations = 7;
9         if ( n < 10 )
10             iterations++;
11     }
12
13     public double getRoot() {
14         double x = n/4;
15         for ( int i=0; i<iterations; i++ ) {
16             x = (x+(n/x))/2.0;
17         }
18         return x;
19     }
20 }
```

---

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*.

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

```
1  import java.util.Scanner;
2
3  public class SquareRootDriver {
4      public static void main( String[] args ) {
5          Scanner keyboard = new Scanner(System.in);
6          double n;
7
8          SquareRootFinder sqrt = new SquareRootFinder();
9
10         do {
11             System.out.print("Enter a number (or <=0 to quit): ");
12             n = keyboard.nextDouble();
13
14             if ( n > 0 ) {
15                 sqrt.setNumber(n);
16                 System.out.println( sqrt.getRoot() );
17             }
18         } while ( n > 0 );
19     }
20 }
```

## 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`

---

```

1  public class SquareRootTester {
2      public static void main( String[] args ) {
3
4          SquareRootFinder sqrt = new SquareRootFinder();
5
6          double max = 0, maxN = 0;
7          double fakeroot, realroot, diff;
8
9          System.out.print("Testing square root algorithm... ");
10         for ( double n = 0; n<=2000; n += 0.01 ) {
11             sqrt.setNumber(n);
12             fakeroot = sqrt.getRoot();
13             realroot = Math.sqrt(n);
14             diff = Math.abs( fakeroot - realroot );
15             if ( diff > max ) {
16                 max = diff;
17                 maxN = n;
18             }
19         }
20
21         if ( max > 0.000001 ) {
22             System.out.println("FAIL");
23             System.out.println("Worst difference was " + max + " for " + maxN );
24         }

```

```
25         else
26             System.out.println("PASS");
27     }
28 }
```

---

## 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

---

```
1 public class SphereCalc {
2     double radius;
3
4     public void setRadius( double r ) {
5         radius = r;
6     }
7
8     public double getRadius() {
9         return radius;
10    }
11
12    public double getSurfaceArea() {
13        return 4*Math.PI*radius*radius;
14    }
15
16    public double getVolume() {
17        return 4*Math.PI*Math.pow(radius,3) / 3.0;
18    }
19 }
```

---

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

```
1 public class SphereCalcTester {
2     public static void main( String[] args ) {
3
4         SphereCalc c = new SphereCalc();
5
6         c.setRadius(5);
7         if ( isNear(c.getSurfaceArea(), 314.159265359) )
8             System.out.println("PASS: surfaceArea for " + c.getRadius());
9         else
10            System.out.println("FAIL: surfaceArea not what was expected!");
11         if ( isNear(c.getVolume(), 523.598775598) )
12             System.out.println("PASS: volume for " + c.getRadius());
13         else
14            System.out.println("FAIL: volume not what was expected!");
15
16         c.setRadius(0.1);
17         if ( isNear(c.getSurfaceArea(), 0.125663706) )
18             System.out.println("PASS: surfaceArea for " + c.getRadius());
19         else
20            System.out.println("FAIL: surfaceArea not what was expected!");
21         if ( isNear(c.getVolume(), 4.18879E-3) )
22             System.out.println("PASS: volume for " + c.getRadius());
23         else
24            System.out.println("FAIL: volume not what was expected!");
25
26
27     }
28
29     public static boolean isNear( double a, double b ) {
30         return Math.abs(a-b) < 1E-9;
31     }
32 }
```

## 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  $1.0 \times 10^{-9}$ ). 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  $4.18879 \times 10^{-3}$  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`

```

1 public class SphereCalc2 {
2     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() {

```

```
11         return radius;
12     }
13
14     public double getSurfaceArea() {
15         return area;
16     }
17
18     public double getVolume() {
19         return volume;
20     }
21 }
```

---

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?

```
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:

```
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

---

```

1 public class SphereCalc4 {
2     private double radius, area, volume;
3
4     public SphereCalc4( 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 void setRadius( double r ) {
11        radius = r;
12        area = 4*Math.PI*r*r;
13        volume = 4*Math.PI*Math.pow(r,3) / 3.0;
14    }
15
16    public double getRadius()      { return radius; }
17    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`

---

```
1 public class BetterTesting {
2     public static void main( String[] args ) {
3
4         double[] inputs = {
5             5,
6             0.1,
7             3.3,
8             20000,
9             8
10        };
11        double[] areas = {
12            314.159265359,
13            0.125663706,
14            136.84777599,
15            5026548245.743669104,
16            804.247719319
17        };
18        double[] volumes = {
19            523.598775598,
```

```
20         4.18879E-3,
21         150.532553589,
22         3.3510321638291125E13,
23         2144.660584851
24     };
25     int passed = 0;
26     double r, a, v, A, V;
27
28     SphereCalc4 c = new SphereCalc4(0);
29     for ( int i=0; i<inputs.length; i++ ) {
30         r = inputs[i];
31         a = areas[i];
32         v = volumes[i];
33
34         c.setRadius(r);
35         A = c.getSurfaceArea();
36         V = c.getVolume();
37         if ( isNear(A, a) )
38             passed++;
39         else {
40             System.out.print("FAIL: surfaceArea for radius " + r );
41             System.out.println("-- Expected " + a + ", got " + A);
42         }
43         if ( isNear(V, v) )
44             passed++;
45         else {
46             System.out.print("FAIL: volume for radius " + r );
47             System.out.println("-- Expected " + v + ", got " + V);
48         }
49     }
50
51     if ( passed == 2*inputs.length )
52         System.out.println("PASS: All tests passed.");
53 }
54
55 public static boolean isNear( double a, double b ) {
56     return Math.abs(a-b) < 1E-9;
57 }
58 }
```

---

## 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

---

```
1 public class FieldAccess {
2
3     public String first;
4     private String last;
5     String nick;
6
7     public FieldAccess() {
8         first = last = nick = "";
9     }
10
11    public FieldAccess( String f, String l, String n ) {
12        first = f;
13        last = l;
14        nick = n;
15    }
16
17    public void setFirst( String s ) {
18        first = s;
19    }
20
21    public void setLast( String s ) {
22        last = s;
23    }
24
25    public void setNick( String s ) {
26        nick = s;
27    }
28
29    public String getFirst() { return first; }
30    public String getLast() { return last; }
31    public String getNick() { return nick; }
```

```
32
33     public String getFullName() {
34         return first + " \" + nick + "\" " + last;
35     }
36 }
```

---

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

---

```
1  public class FieldAccessDriver {
2      public static void main( String[] args ) {
3          FieldAccess j = new FieldAccess("Robert", "Parker", "Butch");
4          System.out.println(j.getFullName());
5
6          j.setLast("Elliott");
7          j.setFirst("Samuel");
8          j.setNick("Sam");
9          System.out.println(j.getFullName());
10
11         j.first = "Avery";
12         // j.last = "Markham";
13         System.out.println(j.nick);
14     }
15 }
```

---

## 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 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

---

```
1 public class Rectangle {
2     private int length, width;
3
4     public Rectangle() {
5         length = width = 0;
6     }
7
8     public Rectangle( int l, int w ) {
9         length = l;
10        width = w;
11    }
12
13    public int getArea() {
14        return length*width;
15    }
16 }
```

---

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.

# Exercise 13: Default Values for Fields

It is a good idea to make sure that variables have values before using them. In fact, in driver code, Java won't even compile the file if you attempt to use a variable before initializing it.

Inside classes, however, Java does some hidden magic behind the scenes to make this a little easier. Fields have *default* values that will be used even if you never set them to anything.

Dog.java

---

```
1 public class Dog
2 {
3     private String name, breed;
4     private int age;
5     private double weight;
6
7     public Dog() {}
8
9     public Dog( String n, String b, int a, double w ) {
10         name = n;
11         breed = b;
12         age = a;
13         weight = w;
14     }
15
16     public String getName() { return name; }
17     public String getBreed() { return breed; }
18     public int getAge() { return age; }
19     public double getWeight() { return weight; }
20
21     public String getEverything() {
22         return name + " is a " + breed + ", " + age + " years old, "
23             + weight + " kg.";
24     }
25 }
```

---

This object is hopefully pretty straightforward. Several fields, two constructors (one default constructor and one with parameters), a bunch of getters, and a “get everything” method that returns a nice String suitable for printing.

The only “weird” thing is on line 7: the default constructor is present but empty. There’s no code between those curly braces!

This compiles, though. Now for the driver, which should also be pretty boring. Try to predict what the output will be before you run it, though.

#### DogDriver.java

```
1 public class DogDriver {  
2     public static void main( String[] args ) {  
3         Dog a = new Dog();  
4  
5         Dog pal = new Dog("Lassie", "Rough Collie", 3, 26);  
6         Dog spike = new Dog("Yeller", "Mastador", 5, 43);  
7  
8         System.out.println( pal.getName() + " is a " + pal.getBreed() + ".");  
9         System.out.println( spike.getName() + " is a " + spike.getBreed() + ".");  
10  
11        System.out.println( pal.getEverything() );  
12        System.out.println( a.getEverything() );  
13    }  
14 }
```

So this driver creates two Dog objects on lines 5 and 6, and calls various methods on lines 8 through 11.

The interesting lines, in my opinion, are line 3 and line 12.

Line 3 instantiates a Dog object using the (empty) default constructor. Then line 12 calls a method that will return values for all the fields of that object. This compiles, but when did those fields get initialized?!?

Line 3 calls the default constructor, which does nothing and then returns a reference to the new object. And the `getEverything()` method behaves as if *name*, *breed*, *age* and *weight* have values! What values are used, then?

## What You Should See

```
Lassie is a Rough Collie.  
Yeller is a Mastador.  
Lassie is a Rough Collie, 3 years old, 26.0 kg.  
null is a null, 0 years old, 0.0 kg.
```

The answer is that fields in Java have *default* values. (And *only* fields do. Other types of variables in Java do *not* have default values, which is why the compiler will complain if you attempt to use a variable before initializing it.)

A field which is an `int` has a default value of `0`. A `double` will default to `0.0`. `booleans` have a default value of `false`. Any object defaults to `null`, which is the reference that means “this object isn’t referring to anything yet.” Strings are objects, so in this case the fields *name* and *breed* are both set to `null`.

Note: the Strings do *not* contain the **String** “`null`”; they contain a `null` reference. We’ll learn more about references and `null` in later chapters.

It’s a good idea to always explicitly put values in all your fields, but it is nice to know that Java will put in sensible defaults if you forget.



## Study Drills

1. In the runner, instantiate one more `Dog` object with values and make a `println()` statement that calls all four “getter” methods. Make the output of that line look identical to if you had just called `getEverything()`.

# Exercise 14: toString and this

Today's exercise is hopefully a little interesting. There aren't many new concepts, and hopefully they're not too complicated.

We're going to be looking at a built-in reference variable that objects have called `this` and a built-in method that all objects inherit called `toString()`. Before typing up this code, you might want to skim back over the `getEverything()` method in the previous exercise (`Dog.java`) and also look at the fields back in `TVActorDriver.java` from Exercise 4.

`Tweet.java`

---

```
1 public class Tweet {
2     private String created_at;
3     private int favorite_count;
4     private boolean favorited;
5     private long id;
6     private int retweet_count;
7     private boolean retweeted;
8     private String text;
9
10    public Tweet( String created_at, int favorite_count, boolean favorited,
11                long id, int retweet_count, boolean retweeted, String text )
12    {
13        this.created_at = created_at;
14        this.favorite_count = favorite_count;
15        this.favorited = favorited;
16        this.id = id;
17        this.retweet_count = retweet_count;
18        this.retweeted = retweeted;
19        this.text = text;
20    }
21
22    public String toString() {
23        String out = "";
24        out += text + "\n";
25        if ( retweet_count > 0 ) {
26            out += "Retweets: " + retweet_count + " ";
27        }
28        if ( this.favorite_count > 0 ) {
```

```

29         out += "Favorites: " + this.favorite_count + "\n";
30     }
31     else {
32         out += "\n";
33     }
34     out += created_at + "\n";
35     if ( this.retweeted ) {
36         out += "[^v]";
37     }
38     else {
39         out += "[ ]";
40     }
41     if ( favorited ) {
42         out += "[*]";
43     }
44     else {
45         out += "[ ]";
46     }
47     return out;
48 }
49
50 }

```

Twitter is a company that enables people to post very short messages online for others to read. The short messages are called “tweets” and are limited to 140 characters. However, the code that makes Twitter run is much more complicated! In the above code, I have defined a Java version of a Tweet object containing *some* of the fields present in an actual “tweet” as defined by Twitter. (If you want to see the rest of the fields, you’ll have to browse [Twitter’s API Overview](https://dev.twitter.com/overview/api/tweets)<sup>8</sup>. Note that Twitter is written in Ruby and Scala and not in Java, so some of the variable types aren’t exactly the same.)

The boring new thing in this object is that the method that returns a nice-looking String suitable for printing is called `toString()` instead of `getEverything()`. There’s a reason for that which we’ll see shortly.

You’ll notice something weird on line 13, though.

The parameter constructor that begins on line 10 has *seven* parameters passed in to it. In previous objects, I have just given the parameters short names – usually just a single letter. But in this object, there are two parameters that start with an ‘f’ and two more that begin with an ‘r’. I was worried that trying to shorten them would be confusing.

Because the parameters have the same names as the fields, they “shadow” the fields. This means the names of the parameters hide the names of the fields because they are defined closer to line 13.

---

<sup>8</sup><https://dev.twitter.com/overview/api/tweets>



But Java gives us a way to access the shadowed fields, using the keyword `this`. In Java, `this` always contains a reference to the object you are in. (It is okay if that is a little confusing. We'll talk a lot more about references later.)

So, `this.created_at` refers to the variable named `created_at` that belongs to the entire current object. Whereas in this scope – on lines 12 through 19 – the variable `created_at` refers to the parameter and not the field.

```
this.created_at = created_at;
```

...means to store a copy of the value in the parameter into the instance variable.

Lines 14 through 19 do the same thing. They store a copy of the parameter's value into the field of the same name.

Notice that in the `toString()` method there aren't any parameters. Therefore the only variable named `retweet_count` in this scope is the field. So on line 25, I can just access it by its name like I've been doing in previous exercises.

However, it is *always* legal to access fields using `this.` as shown on line 28. Some programmers do it always. I usually only do it when I have to.

The `toString()` method starts with an empty String named `out` on line 23 and then builds it up one piece at a time using `if` statements. By the bottom of the method we've got a String that is four lines long and it gets returned to the `main()` for printing.

`TweetDriver.java`

---

```

1 public class TweetDriver {
2     public static void main( String[] args ) {
3         Tweet t = new Tweet( "Thu Feb 19 20:29:00 +0000 2105", 8, true,
4             568507566168223744L, 2, false,
5             "You can now buy 'Learn Java the Hard Way' using bitcoin! Probably!"
6             + "\nThanks, @stripe ! #ljthw"
7         );
8
9         System.out.println( t.toString() );
10        System.out.println("\n-----\n");
11        System.out.println( t );
12    }
13 }
14
15 // https://twitter.com/grahammitchell/status/568507566168223744

```

---

The driver is pretty boring. It is maybe a little interesting that we take 5 lines of code to instantiate a single `Tweet` object because it has so many parameters.

It is perfectly fine to do it over several lines like this. I like to indent the lines inbetween and end with a close paren and semicolon on a line by itself as seen on line 7. It would also be fine if you just put everything on one really long line. That's discouraged, though, since that way you'd have to scroll left and right to read everything and bugs can sneak in that way.

At the beginning of line 4, there's a pretty long number with an "L" at the end of it. That is not a mistake. That value is going to be passed into the parameter called `id`, which is a "long". A "long", which is exactly like an `int` but it takes up more space and can hold a *much* bigger number. Don't worry too much about it for now; we'll talk about it in a future exercise.

We need the "L" here on line 4, though, since `ints` can only hold values up to two billion or so. That number is more than 500 quadrillion! Numbers in Java with no decimal point and no "L" at the end are assumed to be integers. Without tagging it with an "L", the Java compiler would try to package that huge thing into an integer and fail and then bail out with an error.

## What You Should See

```
You can now buy 'Learn Java the Hard Way' using bitcoin! Probably!  
Thanks, @stripe ! #ljthw  
Retweets: 2 Favorites: 8  
Thu Feb 19 20:29:00 +0000 2105  
[ ][*]
```

-----

```
You can now buy 'Learn Java the Hard Way' using bitcoin! Probably!  
Thanks, @stripe ! #ljthw  
Retweets: 2 Favorites: 8  
Thu Feb 19 20:29:00 +0000 2105  
[ ][*]
```

Finally let's talk about why the method is called `toString()` instead of something else. As you can see on line 9 of the driver, when you print out the `String` returned by the `toString()` method, everything works out nicely.

On line 11, however, we are attempting to print the entire object itself! Normally this would display a weird jumble of letters, but since our object defined a public method called `toString()`, the Java compiler will print out what `toString()` returns instead of the object itself.

This is pretty handy!

Okay, that's enough for this exercise!



## Study Drills

1. Go line-by-line through the `toString()` method, putting `this.` in front of every field that didn't have it, and *removing* `this.` from the front of every field that *did* have it.
2. Remove the "L" from the end of the long number on line 4 of the driver, then try to compile it. What error message do you get? Answer in a comment.
3. Replace the "L", but make it lower-case this time. Does it compile now? Answer in a comment.
4. How does the output of the driver change when you change the name of the `toString()` method? Call it something else in the object and on line 9 of the driver, but leave line 11 of the driver alone. What gets printed?

# Exercise 15: Noughts and Crosses / Extreme Testing

The code in this exercise is quite long! That's because you now know enough to code some interesting things. The first thing we're going to type up is an object that can be used to play the children's game "Noughts and Crosses", which is also called "Tic Tac Toe".

NoughtsCrossesObject.java

```
1 public class NoughtsCrossesObject
2 {
3     // Instance Variables
4     private String[][] board;
5     private int turns;
6
7     // Constructor
8     public NoughtsCrossesObject()
9     {
10         board = new String[3][3];
11         turns = 0;
12
13         for ( int r=0; r<3; r++ )
14             for ( int c=0; c<3; c++ )
15                 board[r][c] = " ";
16     }
17
18     // Accessor Methods
19     public boolean isWinner( String p ) {
20         // top row
21         if ( winCheck(p, 0,0, 0,1, 0,2) ) return true;
22         // middle row
23         if ( winCheck(p, 1,0, 1,1, 1,2) ) return true;
24         // bottom row
25         if ( winCheck(p, 2,0, 2,1, 2,2) ) return true;
26         // left column
27         if ( winCheck(p, 0,0, 1,0, 2,0) ) return true;
28         // middle column
29         if ( winCheck(p, 0,1, 1,1, 2,1) ) return true;
30         // right column
```

```
31         if ( winCheck(p, 0,2, 1,2, 2,2) ) return true;
32         // diagonal top-left to bottom-right
33         if ( winCheck(p, 0,0, 1,1, 2,2) ) return true;
34         // diagonal bottom-left to top-right
35         if ( winCheck(p, 2,0, 1,1, 0,2) ) return true;
36
37         return false;
38     }
39
40     private boolean winCheck(String p, int a,int b, int c,int d, int e,int f) {
41         return board[a][b].equals(board[c][d])
42             && board[a][b].equals(board[e][f]);
43     }
44
45     public boolean isFull() {
46         if ( turns == 9 )
47             return true;
48         else
49             return false;
50         // return turns == 9;
51     }
52
53     public boolean isCat() {
54         return isFull() && !isWinner("X") && !isWinner("O");
55     }
56
57     public boolean isValid( int r, int c ) {
58         if ( 0 <= r && r <= 2 && 0 <= c && c <= 2 )
59             return true;
60         else
61             return false;
62     }
63
64     public int numTurns() {
65         return turns;
66     }
67
68     public String playerAt( int r, int c ) {
69         if ( isValid(r,c) )
70             return board[r][c];
71         else
72             return "@";
```

```

73     }
74
75     public boolean isTaken(int r, int c) {
76         String p = playerAt(r,c);
77         if ( p.equals(" ") )
78             return false;
79         else
80             return true;
81     }
82
83     public String toString() {
84         String out = "";
85         out += "  0  " + board[0][0] + "|" + board[0][1] + "|" + board[0][2] + "\n";
86         out += "  --+--+ " + "\n";
87         out += "  1  " + board[1][0] + "|" + board[1][1] + "|" + board[1][2] + "\n";
88         out += "  --+--+ " + "\n";
89         out += "  2  " + board[2][0] + "|" + board[2][1] + "|" + board[2][2] + "\n";
90         out += "  0 1 2  " + "\n";
91         return out;
92     }
93
94     // Modifier / Mutator Method
95     public void playMove( String p, int r, int c ) {
96         board[r][c] = p;
97         turns++;
98     }
99 }

```

---

The first instance variable (the one called *board*) is a two-dimensional array of Strings. That means the array has a bunch of strings in it arranged in rows and columns.

There's a constructor that instantiates the two-dimensional array with three rows and three columns, sets *turns* to zero, and then uses a nested loop to put a blank space into every spot in the array.

The *isWinner()* method returns true if player *p* has won and false if that player has not yet won. It accomplishes its task using a "helper method" called *winCheck()*. Notice that *winCheck()* is a **private** method, which means that we can call it from inside this object but it cannot be called from the driver or anywhere else.

*winCheck()* checks if row *a* and column *b* contains the same letter as the letter in row *c* and column *d*. Then it confirms that the letter in [*a*,*b*] is *also* the same as the letter in [*e*,*f*]. If all those things are true, then a true value is returned from the method.

*isFull()* returns true when all nine squares of the board have been used up. You can do this with

an if statement and else statement as shown, or you could do it in one line of code (which is commented out).

isCat() returns true when the board is full but neither player has won.

isValid() returns true if the row value and column value are both in range (0-2). I could have also written this on one line.

numTurns() just returns a copy of the value in the instance variable, and playerAt() will return the "X" or "O" contained in the requested row/column. However, it will *only* do this if the request is valid. Otherwise it will return an "@" symbol instead.

isTaken() returns true when the requested location has already been played. You can do this method in a single line, too, though it was awkward enough that I didn't like it.

The toString() method returns a picture of the board. If the board had been really large I would have figured out a way to do this using loops, but for a 3x3 grid it is really much easier to just list all the squares like this.

Finally the playMove() method is the only method that *changes* the object. It stores the move for player *p* into the specified row and column. It also increments the number of moves.

Currently there's no error-checking in playMove(); it will happily place a move on top of an existing move. The driver will have to account for this.

Now let's type up the driver so we can test our object!

NoughtsCrossesGame.java

---

```

1  import java.util.Scanner;
2
3  public class NoughtsCrossesGame {
4      public static void main( String[] args ) {
5          Scanner keyboard = new Scanner(System.in);
6
7          String p = "X";
8          NoughtsCrossesObject ttt = new NoughtsCrossesObject();
9          int r, c;
10
11         while ( ! ( ttt.isWinner("X") || ttt.isWinner("O") || ttt.isFull() ) ) {
12             System.out.println(ttt);
13             System.out.print("'" + p + "', choose your location (row column): ");
14             r = keyboard.nextInt();
15             c = keyboard.nextInt();
16
17             while ( ! ttt.isValid(r,c) || ttt.isTaken(r,c) ) {
18                 if ( ttt.isValid(r,c) == false )
19                     System.out.println("Not a valid location. Try again.");

```

```

20         else if ( ttt.isTaken(r,c) )
21             System.out.println("Location already full. Try again.");
22
23         System.out.print( "Choose your location (row column): " );
24         r = keyboard.nextInt();
25         c = keyboard.nextInt();
26     }
27     ttt.playMove( p, r, c );
28
29     if ( p.equals("X") )
30         p = "O";
31     else
32         p = "X";
33 }
34
35 System.out.println(ttt);
36
37 if ( ttt.isWinner("X") )
38     System.out.println("X is the winner!");
39 else if ( ttt.isWinner("O") )
40     System.out.println("O is the winner!");
41 else if ( ttt.isCat() )
42     System.out.println("The game is a tie.");
43 }
44 }

```

---

You can see that on line 8 we have instantiated our object, and then we get to use all the provided methods to play the game.

Line 12 prints the `ttt` object, which will call the `toString()` method. That method will return the current state of the board.

Lines 14 and 15 let the human type in a row and a column. You can do this by just typing two numbers at the prompt with a space between them.

Lines 17 through 26 do error-checking to make sure the player is playing in a legal spot.

Line 27 actually places their move, then lines 29-32 change the player from “X” to “O” or vice-versa. And lines 37 through 42 declare a winner.

So, go ahead and compile and run the driver! Let’s play some Noughts and Crosses!

## What You Should See



```

0  | |
  +-+
1  | |
  +-+
2  | |
   0 1 2

X is the winner!

```

Oh, erm, what?

So, I have made an error. Somewhere in the object. If all the code were in the `main()`, this would be hard to track down. Fortunately since it's an object with methods that you can call one at a time, we can automate testing of it!

What follows is a tester I wrote several years ago so my students could find bugs in *their* tic-tac-toe programs. I'm not going to explain it, and I don't even recommend typing it up. Download or just copy the file that I provided.

I can tell you that this tester plays every possible game of tic-tac-toe that could ever exist and checks the status of the board each time. If the object we are testing reports something different than what the tester expects the result to be, it complains.

If this tester passes your code, then your code is very unlikely to have bugs. This is a nice thing to know.

NoughtsCrossesTester.java

---

```

1  // originally written 2007-09-19
2
3  import java.util.Arrays;
4
5  public class NoughtsCrossesTester
6  {
7      private static NoughtsCrossesObject ttt;
8
9      public static void main( String[] args ) {
10         // constructor
11         System.out.print("Checking constructor.....");
12         System.out.print(".....");
13         ttt = new NoughtsCrossesObject();
14         check( "numTurns()",    ttt.numTurns(),    0 );
15         check( "isWinner('X')", ttt.isWinner("X"), false );
16         check( "isWinner('O')", ttt.isWinner("O"), false );

```

```

17     check( "isCat()",      ttt.isCat(),      false );
18     check( "isFull()",    ttt.isFull(),    false );
19     System.out.println("  [ok]");
20
21     // is Valid
22     int trials = 0;
23     System.out.print("Checking isValid()...");
24     for ( int r = -100; r<=100; r++ )
25         for ( int c = -100; c<=100; c++ ) {
26             check( "isValid("+r+","+c+")", ttt.isValid(r,c),
27                 ( 0 <= r && r < 3 && 0 <= c && c < 3 ) );
28             if ( ++trials % 777 == 0 ) System.out.print(".");
29         }
30
31     System.out.println("  [ok]");
32
33     // play every possible game to make sure it's scoring right
34     byte[] game = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
35     int[] cells = { 0, 1, 2, 4, 8, 16, 32, 64, 128, 256 };
36     int n = 0, i;
37     boolean gameOver = false;
38     String p = "X";
39     int r, c, winner = 3;
40     int xTotal = 0, oTotal = 0;
41     boolean xWin, oWin, gato;
42     trials = 0;
43
44     System.out.print("Checking isWinner()...");
45     while ( game[0] < 10 ) {
46         // play a single game
47         ttt = new NoughtsCrossesObject();
48         n = 0;
49         xTotal = oTotal = 0;
50         for ( byte b : game ) {
51             r = (b-1) / 3;
52             c = (b-1) % 3;
53             ttt.playMove(p,r,c);
54
55             if ( p.equals("X") )
56                 xTotal += cells[b];
57             else
58                 oTotal += cells[b];

```

```

59
60         n++;
61         xWin = ( Arrays.binarySearch(winPatterns,xTotal) >= 0 );
62         oWin = ( Arrays.binarySearch(winPatterns,oTotal) >= 0 );
63         gato = ( ! xWin && ! oWin && n == 9 );
64
65         check( "numTurns()",    ttt.numTurns(),    n );
66         check( "isWinner('X')", ttt.isWinner("X"), xWin );
67         check( "isWinner('O')", ttt.isWinner("O"), oWin );
68         check( "isCat()",        ttt.isCat(),        gato );
69         check( "isFull()",       ttt.isFull(),       n == 9 );
70
71         p = ( p.equals("X") ? "O" : "X" );
72
73         if ( xWin || oWin )
74             break;
75     }
76
77     trials++;
78     if ( trials % 7200 == 0 )
79         System.out.print(".");
80
81     next_permutation( game );
82 }
83 System.out.println(" [ok]");
84 System.out.println("\nAll tests passed!\n");
85 }
86
87 private static void check(String property, boolean is, boolean shouldbe) {
88     if ( is != shouldbe ) {
89         System.out.println("\n\tFATAL ERROR: " + property + " returns "
90             + is + ", but should be " + shouldbe);
91         System.out.println(ttt);
92         System.exit(1);
93     }
94 }
95
96 private static void check(String property, int is, int shouldbe) {
97     if ( is != shouldbe ) {
98         System.out.println("\n\tFATAL ERROR: " + property + " returns "
99             + is + ", but should be " + shouldbe);
100        System.out.println(ttt);

```

```

101         System.exit(1);
102     }
103 }
104
105 private static boolean unique( byte[] a ) {
106     if ( a[0] == 10 )
107         return true;
108
109     boolean[] used = new boolean[10];
110
111     for ( int i=0; i<a.length; ++i )
112         if ( used[ a[i] ] )
113             return false;
114         else
115             used[ a[i] ] = true;
116
117     return true;
118 }
119
120 private static void increment( byte[] a ) {
121     a[a.length-1]++;
122     for ( int i=a.length-1; i>0; --i )
123         if ( a[i] >= 10 ) {
124             a[i] = 1;
125             a[i-1]++;
126         }
127 }
128
129 private static void next_permutation( byte[] a ) {
130     do {
131         increment( a );
132     } while ( ! unique(a) );
133 }
134
135 private static int[] winPatterns = {
136     7, 15, 23, 39, 56, 57, 58, 60, 71, 73, 75, 77, 79,
137     84, 85, 86, 87, 89, 92, 93, 94, 103, 105, 107, 116, 117,
138     118, 120, 121, 122, 124, 135, 143, 146, 147, 150, 151, 154, 158,
139     167, 178, 179, 184, 185, 186, 188, 201, 205, 210, 212, 213, 214,
140     220, 233, 242, 244, 263, 271, 273, 275, 277, 279, 281, 283, 285,
141     292, 293, 294, 295, 300, 302, 305, 307, 308, 309, 312, 313, 314,
142     316, 329, 331, 337, 339, 340, 341, 342, 345, 348, 356, 358, 369,

```

```

143         372, 401, 402, 403, 405, 409, 410, 420, 421, 428, 433, 448, 449,
144         450, 452, 456, 457, 458, 460, 464, 465, 466, 468, 480, 481, 482, 484
145     };
146 }

```

---

## What You Should See

```

Checking constructor.....
      FATAL ERROR: isWinner('X') returns true, but should be false
0   | |
   +-+
1   | |
   +-+
2   | |
   0 1 2

```

You may not be able to tell from the output, but the tester starts out by creating a single object (on line 13). Before anyone has made any moves, the method call `ttt.isWinner("X")` is *supposed* to return false, but it says true instead!

Can you spot the error in the object? I'll give you a hint; it is on line 41 in the `winCheck()` method.

Think about it a bit before scrolling down.

*think*

*think*

*think*

*think*

The basic problem is that `winCheck()` never looks at the parameter *p*. It does check that `[a,b] == [c,d] == [e,f]`. But this is true even when the board is filled with spaces! It needs to check that those three squares are equal *and* that they have the same letter in them that *p* does!

You will fix this in the Study Drill. Once you have fixed it, the output should look like so:

## What You Should See

```


```

```

Checking constructor..... [ok]
Checking isValid()..... [ok]
Checking isWinner()..... [ok]

All tests passed!

```

And running the game will now look like this:

```

0  | |
  +-+
1  | |
  +-+
2  | |
   0 1 2

'X', choose your location (row column): 1 1
0  | |
  +-+
1  |X|
  +-+
2  | |
   0 1 2

'O', choose your location (row column):

```



## Study Drills

1. Fix `winCheck()`. Confirm that the tester passes now and that the game plays as expected.

# Exercise 16: Introduction to ArrayLists

Whew. That last exercise was a long one! Let's take a break for a few exercises and talk about something else: ArrayLists.

ArrayLists are Java objects you can import. They work a bit like arrays (they hold many values at once and you give them a number to say which value you want), but they *also* have a ton of built-in methods to make it much easier to do a lot of things you need to do manually with an array.

Let's look at some code! Note the import statement at the top. ArrayLists are part of Java's standard library.

ArrayListIntro.java

---

```
1  import java.util.ArrayList;
2
3  public class ArrayListIntro {
4      public static void main(String[] args) {
5          ArrayList<String> arr = new ArrayList<String>();
6
7          System.out.println( "ArrayList has size(): " + arr.size() );
8
9          arr.add("Ant-Man");
10         arr.add("Hulk");
11         arr.add("Iron Man");
12         arr.add("Thor");
13         arr.add("Wasp");
14
15         System.out.println( "ArrayList has size(): " + arr.size() );
16         System.out.println( "Current contents: " + arr );
17         showList(arr);
18
19         arr.add(3, "Jarvis"); // inserts into slot 3; others slide right
20         showList(arr);
21
22         int i = arr.indexOf("Hulk");
23         System.out.println( arr.get(i) + " is located in slot " + i );
24         arr.remove(i); // removes value in that slot; others slide left
25         showList(arr);
26
27         arr.add(1, "Captain America"); // original slot 1 and other slide right
```

```
28     showList(arr);
29
30     arr.set(1, "Cap"); // replaces the value in slot 1
31     showList(arr);
32
33     arr.remove(5); // removing backward lets us avoid sliding
34     arr.remove(4);
35     arr.remove(2);
36     arr.remove(0);
37     showList(arr);
38
39     arr.add("Hawkeye");
40     arr.add("Quicksilver");
41     arr.add("Scarlet Witch");
42     showList(arr);
43
44     i = arr.indexOf("Hawkeye");
45     String removed = arr.remove(i);
46     arr.add(i-1, removed);
47     showList(arr);
48
49     System.out.println( "Current contents: " + arr );
50 }
51
52 public static void showList( ArrayList<String> a ) {
53     System.out.println("size() is " + a.size() );
54     for ( int i=0; i<a.size(); i++ ) {
55         int len = a.get(i).length() + 4; // two quotes, two spaces
56         System.out.print("|" + centerPad(""+i, len));
57     }
58     System.out.println("|");
59     for ( String s : a )
60         System.out.print("| \"" + s + "\" ");
61     System.out.println("|\\n");
62 }
63
64 public static String centerPad(String s, int width) {
65     int totalSpaces = width - s.length();
66     int leftSide = totalSpaces / 2;
67     int rightSide = totalSpaces - leftSide;
68     String out = "";
69     for ( int i=0; i<leftSide; i++ )
```



```
70         out += " ";
71     out += s;
72     for ( int i=0; i<rightSide; i++ )
73         out += " ";
74     return out;
75 }
76 }
```

---

There's only one file here! You could compile this and run right away!

First up (line 5) we instantiate an ArrayList object. This one is an ArrayList of Strings, which is what the angle brackets are about. You can make an ArrayList of any type of thing. ArrayLists are initially empty.

ArrayLists have a `size()` method which tells you how many items are currently stored in it. Note that this doesn't have anything to do with its *capacity*: they grow and shrink as you add and remove items.

Next up, we add five items to the list using the `add()` method. The first item you add goes into slot 0. The next one into slot 1, etc. After these calls, the list will have a `.size()` of 5.

On line 16 you can see (maybe) that the ArrayList prints nicely on it's own. The fact that it prints well implies that there is a `toString()` method inside the ArrayList class which builds up and returns a nice-looking String for printing!

But I wasn't happy with the default display, because I wanted slot numbers in there, and I wanted them to be centered, so that's what the `showList()` function does.

The `centerPad()` function returns a String of the given width. That String will have the parameter `s` in the center and spaces to the left and right. I know it's overkill, but I wanted it to look nice. Anyway, back to the built-in methods in the ArrayList class.

If you only provide a value to the `add()` method, it puts the new value at the end, resizing if necessary. But if you also pass in a slot number to the `add()` method, it will put the new value in the slot you specify, and it moves all values to the right (larger slot numbers) to make room. We do this on line 19, and looking at the output should make it clear what happens to the other values in the list.

## What You Should See



```

ArrayList has size(): 0
ArrayList has size(): 5
Current contents: [Ant-Man, Hulk, Iron Man, Thor, Wasp]
size() is 5
| 0 | 1 | 2 | 3 | 4 |
| "Ant-Man" | "Hulk" | "Iron Man" | "Thor" | "Wasp" |

size() is 6
| 0 | 1 | 2 | 3 | 4 | 5 |
| "Ant-Man" | "Hulk" | "Iron Man" | "Jarvis" | "Thor" | "Wasp" |

Hulk is located in slot 1
size() is 5
| 0 | 1 | 2 | 3 | 4 |
| "Ant-Man" | "Iron Man" | "Jarvis" | "Thor" | "Wasp" |

size() is 6
| 0 | 1 | 2 | 3 | 4 | 5 |
| "Ant-Man" | "Captain America" | "Iron Man" | "Jarvis" | "Thor" | "Wasp" |

size() is 6
| 0 | 1 | 2 | 3 | 4 | 5 |
| "Ant-Man" | "Cap" | "Iron Man" | "Jarvis" | "Thor" | "Wasp" |

size() is 2
| 0 | 1 |
| "Cap" | "Jarvis" |

size() is 5
| 0 | 1 | 2 | 3 | 4 |
| "Cap" | "Jarvis" | "Hawkeye" | "Quicksilver" | "Scarlet Witch" |

size() is 5
| 0 | 1 | 2 | 3 | 4 |
| "Cap" | "Hawkeye" | "Jarvis" | "Quicksilver" | "Scarlet Witch" |

Current contents: [Cap, Hawkeye, Jarvis, Quicksilver, Scarlet Witch]

```

ArrayLists also have an `indexOf()` method, which will find the value you pass in and return the slot number where it is first found. If it is not found, it returns -1 instead of a slot number.

The `remove()` method removes the value from the indicated slot and then moves all other values to the left (smaller slot numbers) to fill in the hole. Again, the output makes that clear, I hope.

There's also a `set()` method (line 30) which *replaces* an existing value. The rest of the values don't move.

Removing a bunch of values is always tricky because the values are all sliding to the left (to smaller indexes). So a neat trick is to remove the values from biggest to smallest so that none of the slot numbers change as you go.

On lines 44 through 47 my intent is to move "Hawkeye" so that he is in alphabetical order. Fortunately, the `remove()` method doesn't *just* remove the value; it also returns the value that was removed. And so I can add it back to the list in the slot *before* the one it had been in.

Anyway, we'll look at a more *useful* program that uses an ArrayList in the next exercise, but that's how they work! ArrayLists are really handy.



## Study Drills

1. Create a second ArrayList of Strings at the bottom of `main()`. Add three or four values to it, remove at least one value, then display the state of the ArrayList using either the `showList()` function or by just printing it.

# Exercise 17: Word Counter Using an ArrayList

This exercise uses an ArrayList of Strings to count the number of words in an entire book!

WordCounter.java

---

```
1  import java.net.URL;
2  import java.util.ArrayList;
3  import java.util.Scanner;
4
5  public class WordCounter {
6
7      private ArrayList<String> words;
8
9      public WordCounter( String url ) {
10         words = new ArrayList<String>();
11
12         Scanner webInput = null;
13         try {
14             URL u = new URL(url);
15             webInput = new Scanner(u.openStream());
16         }
17         catch ( Exception e ) {
18             System.err.println("Couldn't open the URL '" + url + "'. Sorry.");
19             System.exit(1);
20         }
21
22         while ( webInput.hasNext() )
23             words.add( webInput.next() );
24         webInput.close();
25     }
26
27     public int getCount( String word ) {
28         int count = 0;
29         for ( String s : words ) {
30             if ( s.equals(word) ) {
31                 count++;
32             }
33         }
34     }
35 }
```

```
33     }  
34     return count;  
35 }  
36 }
```

---

The only field in this object is an ArrayList of Strings. It gets instantiated in the constructor, then we wrap a Scanner object around the URL which is passed in.

Assuming the URL was opened successfully we read through the entire file one word at a time, adding each word to the ArrayList – duplicates and all! ArrayLists automatically grow as items are added, so doing this job with a regular array would be quite a bit more code.

The getCount() method returns a count of the specified word. It loops through each word in the list using a for-each loop, so the variable *s* takes on the value of each String in the list one at a time.

And that's it! Not much going on in the object except building the list and counting words.

WordCounterDriver.java

---

```
1 public class WordCounterDriver {  
2     public static void main( String[] args ) {  
3         String book = "http://www.gutenberg.org/cache/epub/159/pg159.txt";  
4  
5         WordCounter wc = new WordCounter(book);  
6  
7         System.out.println( wc.getCount("the") );  
8         System.out.println( wc.getCount("I") );  
9         System.out.println( wc.getCount("a") );  
10    }  
11 }
```

---

The driver is even shorter! It instantiates a WordCounter object and then calls the getCount() method a few times.

The book is *The Island of Doctor Moreau* by H. G. Wells. This book was first published in 1896 and the copyright has now expired; it is in the public domain now and is hosted online by the excellent Project Gutenberg.

## What You Should See

```
2865  
1380  
1046
```

There's not much to say about this exercise. It's a pretty standard use of an `ArrayList`. It is just additional practice with `ArrayLists` and about using an object.

The try-catch block is used for code that might throw an exception. In this case, instantiating a `URL` object might blow up with a “malformed URL exception.” And the `URL` class' `openStream()` method might throw an exception, too, if it is unable to open a stream to that `URL`. So we put both of these statements in a try block.

Assuming both succeed, the code in the catch block is skipped. If either statement throws an exception, the code in the catch block runs. Here, we just complain and quit the program. Sometimes, it is possible to overcome the error and continue running but we don't attempt that here.



## Study Drills

1. The `getCount()` method only counts *exact* matches of the word, including capitalization. Change it so that words will match regardless of their case.

# Exercise 18: Primitive Variables in Memory

Up to this point I've been relatively hand-wavy about variables. Variables are a place you stash values. They have a name (identifier), a type (int or double) and – eventually – a value.

But I think you have been coding long enough now that you are ready for some more detail. Type in the code, and then we'll talk.

PrimitivesInMemory.java

```
1 public class PrimitivesInMemory {  
2     public static void main(String[] args) {  
3  
4         int age = 41;  
5         double btc = 0.11307678;  
6         boolean veracity = true;  
7  
8         short month = 11;  
9         byte day = 31;  
10        long debt = 18153890112907L;  
11        float mass = 78.971F;  
12  
13        System.out.println( btc );  
14        System.out.println( debt );  
15        System.out.println();  
16  
17        String name = "Gosling";  
18    }  
19 }
```

## What You Should See

```
0.11307678  
18153890112907
```





remember that they're roughly plus-or-minus two billion. (The exact number is  $-2^{31}$  up to  $2^{31} - 1$ .)

So when should you use these? I'm going to disagree with most programming books and say that you shouldn't. Just use `int` for everything unless you're pretty sure your calculation is going to get over two billion, then use a `long`. (And if a `long` isn't big enough, then you'll have to import `java.math.BigInteger`.)

Older books and older programmers will say that you should use the smallest size variable that's big enough for your data because it's more "efficient".

But first of all, that isn't necessarily true. On 32-bit and 64-bit processors, working with just 8 bits at a time might actually be slower. Secondly, it smacks of premature optimization. Just write the code, get it to work, and *then* determine if it's fast enough.



It is nearly always easier to make *working* code faster than it is to make broken or unfinished "efficient" code work.

Okay, let's talk about the two floating-point types: `float` and `double`.

A `float` takes up 32 bits. A `double` takes up 64 bits. I usually simplify by saying that they're stored in scientific notation, but technically doubles are stored in the IEEE 754 double-precision binary floating-point format known as "binary64".

In binary64 representation, the most-significant bit is the sign bit. Then the next eleven bits store the exponent and the remaining 52 bits store the "significand". The details are pretty complicated, so look it up if you want more information.

binary64 gives you everything from  $10^{-308}$  to  $10^{308}$  with full 15-17 decimal digits of precision (significant figures, if you like).

In Java, `float` variables are stored as the IEEE 754 single-precision binary floating-point format known as "binary32". `binary32` only uses 8 bits for the exponent and 23 bits for the significand, which means you only get from 6 to 9 significant decimal digits of precision.

Six significant digits is pretty few, so I don't ever use `float` in my own Java programs unless I know for sure I really can't afford to use up 64 bits per variable. Which is basically never.

One thing I skipped over in lines 10 and 11 are the letters at the end of the numbers.

A number in Java is called a "literal value". So `123` is an integer literal, and `12.3` is a double-precision floating-point literal.

Integer literals in Java are `ints` unless you put an "L" at the end, which means it's a `long`. Floating-point literals in Java are `doubles` unless you put an "F" at the end to signify it as a single-precision `float`.

## Reference Types

On line 17 we have created a `String` variable called “name” and put the value “Gosling” into it. Except we didn’t.

Strings in Java aren’t primitive data types, so the value “Gosling” isn’t stored directly into the variable. The string value is actually stored somewhere else in memory (called the “heap”), and then its memory address is stored in the variable instead.

In Java, a variable that stores this kind of memory address is called a “reference”.

This exercise is already too long, though, so more on references in the next exercise!



## Study Drills

1. Create three more variables: another `short`, a `byte`, and a `long`. Put a value into the *long* variable that wouldn’t have fit into a 32-bit integer. Then try to put some values into the *byte* and the *short* that won’t fit. Add a comment about which values didn’t work, then fix the values so that they *do* fit.

# Exercise 19: Reference Variables in Memory

Primitive variable types store the variable's value directly in the variable's memory location. References variables store the value somewhere else, and the reference variable's memory location instead holds the location of that "somewhere else".

Don't worry if you didn't quite follow all that; the rest of the chapter will be spent trying to explain it better.

ReferencesInMemory.java

---

```
1  import java.util.Scanner;
2  import java.util.ArrayList;
3
4  class Example {
5  }
6
7  public class ReferencesInMemory {
8      public static void main(String[] args) {
9
10         int age = 41;
11         String name = "Gosling";
12
13         int[] digits = { 3, 1, 4, 1, 5, 9, 2 };
14         Scanner keyboard = new Scanner(System.in);
15         ArrayList arr = new ArrayList();
16         Integer year = new Integer(1995);
17         Object obj = new Object();
18         Example ex = new Example();
19
20         System.out.println( digits );
21         System.out.println( obj );
22         System.out.println( ex );
23         System.out.println();
24
25         String[] names = { "John", "Paul", "George", "Ringo", "Brian" };
26         System.out.println( names );
27     }
28 }
```

---

## What You Should See

```
[I@1540e19d
java.lang.Object@677327b6
Example@14ae5a5

[Ljava.lang.String;@7f31245a
```

```
int x;
int y = 5;
double z = 1;
String s;
```

After this code finishes, the symbol table might look something like this:

identifier	address	type	initialized?
x	@38000100	int	no
y	@38000096	int	yes
z	@38000088	double	yes
s	@38000080	reference to String	no

The symbol table contains the identifiers and their locations, but does *not* usually hold the values of the variables. Those are stored in a different area of memory called “the stack”.

## The Stack

Each computer program (technically each thread in a multi-threaded program) gets one stack, and the stack is where all of the program’s primitive variables are stored, along with other control-flow bookkeeping information. Generally speaking, addresses in the stack tend to start with bigger numbers and grow down. The details of the stack are a bit complicated, though, so don’t worry too much about them.

Anyway, a piece of the stack for this program might look like this:

@38000100	...096	...092	...088	...084	...080
?????	5		1		?????

You’ll note that all four variables (x, y, z and s) have now been *declared*; their names and types have been set. But only two of them have been *initialized* so far; x and s don’t have values.

Let’s put values into both variables.

```
x = 12;
s = "hello";
```

The symbol table has only changed a little bit because we haven’t declared any new variables.

identifier	address	type	initialized?
x	@38000100	int	yes
y	@38000096	int	yes
z	@38000088	double	yes
s	@38000080	reference to String	yes

But the values on the stack should be different:

@38000100	...096	...092	...088	...084	...080
12	5		1		@02001000

So the value 12 is now stored in the memory location for *x*. But the memory location for *s* does *not* contain the string literal "hello"; it contains a *reference* to a different memory location.

The string literal value "hello" is stored in another special section of memory called the "heap".

## The Heap

Memory in the heap is typically managed by the operating system itself, and values are stored wherever it is convenient. All a program's heap values might be stored near each other, or they might be scattered. There's no way to know!

But it's okay, because we have a reference!

@02001000

---

String object, length() 5, contents: hello

So in this example, the String object is located in the heap at the memory address @02001000. The specific structure of the object is defined by the creators of Java.

Make sense? Okay, then let's (finally) look back at the code for this exercise. Skim through the code again before going on.

After the code finishes, the symbol table might look something like this:

identifier	address	type	initialized?
age	@42000100	int	yes
name	@42000092	reference to String	yes
digits	@42000086	reference to int[]	yes
keyboard	@42000078	reference to Scanner	yes
arr	@42000070	reference to ArrayList	yes
year	@42000062	reference to Integer	yes
obj	@42000056	reference to Object	yes
ex	@42000048	reference to Example	yes
names	@42000040	reference to String[]	yes

We have nine variables: one primitive and eight references to objects of various types.

The stack might look like so. (I'm abbreviating a bit so it'll fit on the page.)

'100	'092	'086	'078	'070	'062	'056	'048	'040
41	@408	@eb8	@e40	@2a8	@ec8	@b68	@450	@6f0

'100	'092	'086	'078	'070	'062	'056	'048	'040
------	------	------	------	------	------	------	------	------



In my previous examples, I listed the references as “regular” (base-10) numbers, but programmers typically write memory address in hexadecimal (base-16) form. Java does this, too, so my made-up references above are in base-16.

So the one primitive value (*age*) is stored directly in the stack, the rest of the values on the stack are references. Notice that they’re scattered all around in memory.

For example, the symbol table tells us that the *shallow* value of the variable *keyboard* is stored on the stack at memory location @42000078. Then we find that the actual *Scanner* object is allocated in the heap at memory location @e40. The *deep* value of the variable *keyboard* is the *Scanner* object itself.

The variable *age* only has a shallow value, and that value is on the stack. The variable *keyboard* has a shallow value *and* a deep value. The shallow value is the reference and the deep value is the object itself.

This might seem like a subtle difference, but it is important.

You’ll notice that arrays behave a lot like objects. (Because in Java, arrays *are* objects!) On line 13 in the code we declared and initialized an array of integers called *digits*. But on line 20 when we attempted to print out the shallow value of the array, it only displayed the reference plus a little extra information.

```
[I@1540e19d
```

The first character of the output is ‘[’, which means that this is a reference to an array. The second character is ‘I’, which means that it’s a reference to an array of primitive integers. Then there’s an ‘@’ sign followed by the reference/memory address itself (in hexadecimal).

(The reference will be different in your own output, of course. The memory address in the heap is different!)

The remaining output of our program is similar. It gives the type followed by the reference. Running the program several times might cause the references to change, but it might not.

The final line of output is what you see for an array of *String* objects.

Well, I think that’s quite enough for this exercise! We will talk more about this distinction between objects and references and primitive values in future exercises.

**Note:** In some other programming languages (notably C and C++), it is possible to put an object’s value directly on the stack. In these languages, you can create an object on the stack or you can dynamically allocate it in the heap and store a *reference* to it on the stack like Java does.

In Java, you do not have this option. It is *impossible* to allocate an object directly on the stack. It is *impossible* to create an object without that object being stored in the heap.



## Study Drills

1. Add `println()` statements to display the remaining variables in the program. Why is their output different? Answer in a comment. (Hint: it has to do with the `toString()` method.)



# Exercise 20: Lists of Primitive Values

In the previous two exercises, you *finally* learned the difference between primitive data types (like `int` and `double`) and reference/object types in Java.

So now I can let you in on a secret: `ArrayLists` can *only* hold references. Type in the code, then we will talk about the details.

`ListsOfPrimitives.java`

---

```
1  import java.util.ArrayList;
2
3  public class ListsOfPrimitives {
4      public static void main(String[] args) {
5          ArrayList<String> hats = new ArrayList<String>();
6
7          hats.add("fez");
8          hats.add("bowler");
9          hats.add("beanie");
10         hats.add("western");
11         hats.add("fedora");
12
13         System.out.println( hats );
14
15         String jumble = "";
16         for ( String s : hats ) {
17             jumble += s;
18         }
19         System.out.println("All together now: " + jumble);
20
21         // ArrayList<int> bins = new ArrayList<int>();
22         ArrayList<Integer> bins = new ArrayList<Integer>();
23
24         bins.add(new Integer(1));
25         bins.add(new Integer(3));
26         bins.add(new Integer(3));
27         bins.add(new Integer(1));
28
29         bins.add(1);
30         bins.add(4);
```

```

31     bins.add(6);
32     bins.add(4);
33     bins.add(1);
34
35     System.out.println( bins );
36
37     int total = 0;
38     for ( Integer N : bins ) {
39         int n = N.intValue();
40         total += n;
41     }
42     System.out.println("The total is " + total);
43
44     total = 0;
45     for ( int n : bins ) {
46         total += n;
47     }
48     System.out.println("The total is still " + total);
49
50     ArrayList<Character> letters = new ArrayList<Character>();
51     letters.add('z'); // auto-boxes char
52
53     ArrayList<Double> weights = new ArrayList<Double>();
54     weights.add(0.14); // auto-boxes double
55
56     ArrayList<Boolean> dealt = new ArrayList<Boolean>();
57     while ( dealt.size() < 52 )
58         dealt.add(false); // auto-boxes boolean
59 }
60 }

```

## What You Should See

```

[fez, bowler, beanie, western, fedora]
All together now: fezbowlerbeaniewesternfedora
[1, 3, 3, 1, 1, 4, 6, 4, 1]
The total is 24
The total is still 24

```

This exercise begins by creating an `ArrayList` of `String` objects. The list starts out empty, but then we quickly add five values.

It is easy to pretend that after those lines of code, the `ArrayList` contains values like so:

0	1	2	3	4
fez	bowler	beanie	western	fedora

And in fact the `ArrayList` class' `toString()` method makes it look like it, too.

But in reality, `ArrayLists` only hold *references*. So the real picture is more like this:

0	1	2	3	4
@d70	@a60	@868	@db0	@338

Each slot in the `ArrayList` object contains a *reference*, not a string literal. The string *values* are actually stored at the referenced memory locations.

Java goes to a lot of trouble to hide the fact that `Strings` are references sometimes, so you can write a lot of working code without needing to know about them.

The problem comes when you would like to create an `ArrayList` that holds primitive integers. The commented-out code on line 21 shows what we *want* to write. What we must write instead is on line 22... because `ArrayLists` can only hold references.

`int` is a primitive type. `Integer` is an object that contains a single `int`. Java calls this a “wrapper” class, because it’s a very thin layer that wraps around a primitive type.

We want to add four `ints` to the `ArrayList`. But we must *wrap* each `int` with an `Integer` object and add *that* to the `ArrayList` instead, because `ArrayLists` can only hold references.

This is what is happening on lines 24 through 27. When Java was first released, this was the only way to do it. But when version 1.5 of Java came out (2004), “autoboxing” was added, which means that Java now automatically converts `ints` to `Integer` objects for you in most situations.

So now that Java has autoboxing, we can just add the `ints` directly to the `ArrayList` as shown on lines 29 through 33. Java automatically wraps each `int` in an `Integer` object before adding the reference to that `Integer` object to the List.

Similarly, we used to have to call the `intValue()` method to get the `ints` *out* of the `Integer` object. An example of this is on line 39.

But Java also now has “unboxing”, so something like lines 45 and 46 now works. This is nice, but it is probably good to keep in mind: getting primitive values into and out of an `ArrayList` causes some extra work behind the scenes.

If you find that a program with an `ArrayList` of `Integers` is running too slowly, ask yourself if just a native array of `ints` might work instead.

There are wrapper classes for all of the primitive types in Java: `Character`, `Integer`, `Short`, `Byte`, `Long`, `Double`, `Float`, and `Boolean`. The exercise finishes up with some examples of these.



## Study Drills

1. There is a little more inside the `Integer` wrapper class than just a single `int` value. There are fields like `Integer.MAX_VALUE` and `Integer.MIN_VALUE`. There are also some useful methods like `Integer.parseInt()`, which can convert a `String` to an `int`. Add some `println()` statements to display the value of `Long.MAX_VALUE` and `Integer.MAX_VALUE`, and use `Integer.parseInt()` to convert a `String` with some digits in it into a proper primitive `int`.
2. Add some lines of code to manually unbox the `char` from `letters.get(0)` and the `double` from `weights.get(0)`.

# Exercise 21: Generics vs. Casts

When we have been making ArrayLists so far, we have been using a feature that was added to Java in version 1.5: generics. Generics allow you to create structures where the *type* of an object is variable.

This exercise will look at the old way of dealing with this in Java and then the modern way that takes advantage of generics.

GenericsVsCasts.java

---

```
1  import java.util.ArrayList;
2
3  public class GenericsVsCasts {
4
5      @SuppressWarnings("unchecked")
6      public static void main( String[] args ) {
7          ArrayList objarr = new ArrayList();
8
9          objarr.add("one");
10         objarr.add("two");
11
12         Object aa = objarr.get(0);
13         Object bb = objarr.get(1);
14
15         // String qq = objarr.get(0); // won't work
16         // String rr = objarr.get(1);
17
18         String ss = (String)objarr.get(0);
19         String tt = (String)objarr.get(1);
20
21         System.out.println(objarr);
22
23         // -----
24         ArrayList<String> strarr = new ArrayList<>();
25         strarr.add("one");
26         strarr.add("two");
27
28         String s = strarr.get(0);
29         String t = strarr.get(1);
```

```
30
31     System.out.println(strarr);
32
33     Object a = strarr.get(0); // this also works, because Strings are Objects
34     Object b = strarr.get(1);
35
36     // -----
37     Integer i = new Integer(123);
38     objarr.add(i);
39
40     System.out.println(objarr); // ["one", "two", 123] !!!
41
42     // String uu = (String)objarr.get(2); // runtime ClassCastException
43
44     // strarr.add(i); // won't compile
45 }
46 }
```

---

Line 5 is what is called an “annotation”. In this case it is a note to the compiler to not show a warning about “unchecked or unsafe operations” in the `main()` method of this class. The code would compile with or without this annotation, but the warning annoys me.

It is warning me that dealing with `ArrayLists` the old way isn’t safe now that generics exist. Which is true! I promise I don’t do this in real code.

## What You Should See

```
[one, two]
[one, two]
[one, two, 123]
```

I should start by explaining what `Object` is. In Java, *every* object inherits from a special class named `Object`.

We will cover more on inheritance in a later exercise, but the basic idea is that *everything* in Java that isn’t a primitive type is an `Object`.

- Strings are Objects.
- Scanners are Objects.
- Integers are Objects (but not `ints`; those are primitive).

- any class *you* make is an Object

So a variable of type `Object` (like *aa* on line 12) holds a reference, just like every other non-primitive variable in Java. But *aa* can hold a reference to *any* type of object.

## Without Generics

The first line of code in `main()` creates an `ArrayList` called *objarr*, but *doesn't* tell it what type of references it is supposed to hold. So this `ArrayList` can hold references to Objects. *Any* Object.

On lines 9 and 10 we can add two references to Strings to the list, because *objarr* can hold references to Objects and Strings are Objects.

On lines 12 and 13 we retrieve those two references and store copies into our two Object variables called *aa* and *bb*.

Let's pause for a second and look at the symbol table and the `ArrayList` so far:

identifier	address	type	initialized?
objarr	@12a020b0	reference to ArrayList of Objects	yes
aa	@12a020a8	reference to Object	yes
bb	@12a020a0	reference to Object	yes

The `ArrayList` *objarr*:

0	1
@a58	@980

In this made-up example, the String literal "one" is stored on the heap at reference @a58, and the String literal "two" is at reference @980.

Okay, now the stack:

...20b0	...20a8	...20a0
@f98	@a58	@980

So we can see that the variable *aa* (of type `Object`) holds the reference @a58. And the Object variable *bb* holds @980.

@a58 is the reference to the String "one". And @980 is the reference to the String "two". But the code on lines 15 and 16 would *not* compile. Even though the reference stored in `objarr.get(0)` is a reference to a String, that information has been lost. The compiler can only be sure that *objarr* contains references to Objects, and not all Objects are Strings.

So in the Bad Old Days before Java had generics, we had to use a "cast" to get around the compiler's

lack of information. This is demonstrated on lines 18 and 19.

What comes out of the list is a reference to an Object, but we know that is *actually* not just any Object, but a reference to a *String*. We know this because we put the Strings there in the first place!

So `(String)` is a “cast”. (The parens are significant and required.) It tells the compiler “I know you think that reference is just an Object, but I can assure you that it is *actually* a reference to a *String*. Let it go.”

This code would compile even if they *weren't* references to Strings. But if we mistakenly attempt to cast a reference to the wrong type, the cast will “blow up” when the program is running. (Specifically it will throw a `ClassCastException`. You’ll experiment with this during the Study Drills.)

## With Generics

On line 24 we create a second `ArrayList`. But thanks to generics we can specify that it’s an `ArrayList` of references to *Strings*.

(You can put `String` inside the second set of angle brackets if you want, but it is optional. Though *I* often put it there because I forget that you can leave it out.)

After lines 25 and 26, the symbol table doesn’t look much different:

identifier	address	type	initialized?
objarr	@12a020b0	reference to ArrayList of Objects	yes
aa	@12a020a8	reference to Object	yes
bb	@12a020a0	reference to Object	yes
strarr	@12a02098	reference to ArrayList of Strings	yes

And the `ArrayList strarr`:

0	1
@a58	@980

Just like before, the list holds references to the two `String` literals. But this time, the compiler *knows* that they are references to Strings and not just Objects.





If you are very observant, you might have noticed that the addresses of the Strings didn't change in the second list. Even though these addresses are made-up, this is not a copy-paste error.

The Java compiler makes a list of all the String literals in your code and puts them in a special part of the heap called the string pool. (Technically it's the "runtime constant pool".) If the same literal occurs more than once in your code, only one copy is created and all the references to it are shared.

So even though the literal "one" appears *twice* in the code for this exercise, only *one* copy is stored in the heap. The *same* reference is added to both lists.

Anyway, since the compiler can be sure that references to *Strings* are going to come out of *strarr*, lines 28 and 29 work fine, and no cast is needed.

Also! Since Strings are Objects, lines 33 and 34 also work fine. The variable *a* can hold a reference to *any* Object, including the String reference that is retrieved from the ArrayList.



All Strings are Objects, but not all Objects are Strings.

## Shenanigans

Okay, so remember how our original ArrayList (*objarr*) can hold references to any Object? So on line 37 I have created another type of Object.

And line 38 compiles just fine. *objarr* can hold references to Objects, and *i* is a reference to an Integer. So now our list contains two different types of Objects, which is almost certainly a bad idea.

Line 42 would compile. The cast would try to convert an Object reference (which is secretly an Integer reference) to a String reference. It'll compile, but the program would throw a ClassCastException at runtime.

Line 44, on the other hand, won't compile. The compiler knows – thanks to generics! – that *strarr* can only hold references to Strings, and *i* isn't one of those.

I hate getting errors when I compile my code. But I would *much* rather get compiler errors than have my code blow up at runtime.



## Study Drills

1. Uncomment line 42 and confirm that it compiles but throws a runtime exception.
2. Add a line of code toward the bottom that uses a cast to retrieve the Integer object out of *objarr*.
3. Write an if statement that compares *s* with *ss*. (Use `==` instead of `equals()`). Is the if statement true or false? Are you surprised? What do you think is happening? Answer in a comment.

# Exercise 22: Object-Oriented Design and Efficiency

English words are hard to spell. The language has been influenced by so many other languages with different rules. So when I was a kid, sometimes teachers would say a little “rule” for spelling words with I and E in them:

I before E, except after C.

This exercise is a little program to test that rule, and we’ll use it to talk about object-oriented design, efficiency and not doing work until you have to.

IBeforeEChecker.java

---

```
1  import java.util.Scanner;
2  import java.util.ArrayList;
3  import java.io.InputStream;
4
5  public class IBeforeEChecker {
6      private InputStream source;
7      private int matchCount;
8      private boolean computed;
9      private ArrayList<String> wordList;
10
11     public IBeforeEChecker( InputStream source ) {
12         matchCount = 0;
13         computed = false;
14         wordList = new ArrayList<String>();
15         this.source = source;
16     }
17
18     private boolean fitsRule( String word ) {
19         if ( word.contains("ie") && ! word.contains("cie") ) {
20             return true;
21         }
22         else if ( word.contains("cei") ) {
23             return true;
24         }
```

```
25         return false;
26     }
27
28     private void doCalculations() {
29         Scanner input = new Scanner(source);
30         while ( input.hasNext() ) {
31             String word = input.next();
32             int ieLoc = word.indexOf("ie");
33             int eiLoc = word.indexOf("ei");
34             if ( ! wordList.contains(word) && (ieLoc >= 0 || eiLoc >= 0) ) {
35                 wordList.add(word);
36             }
37         }
38         input.close();
39
40         for ( String word : wordList ) {
41             if ( fitsRule(word) )
42                 matchCount++;
43         }
44         computed = true;
45     }
46
47     public int getMatches() {
48         if ( ! computed )
49             doCalculations();
50         return matchCount;
51     }
52
53     public int getExceptions() {
54         if ( ! computed )
55             doCalculations();
56         return getWordCount() - matchCount;
57     }
58
59     public int getWordCount() {
60         if ( ! computed )
61             doCalculations();
62         return wordList.size();
63     }
64
65     public ArrayList<String> getWordList() {
66         if ( ! computed )
```

```

67         doCalculations();
68         return wordList;
69     }
70 }

```

---

I put together a long file of sample English text by combining

- “The Island of Doctor Moreau” by H.G. Wells (1896)
- “Les Misérables” by Victor Hugo (1887), and
- “Ulysses” by James Joyce (1922)

...into one really big file. I picked these novels because they are long, easily available thanks to the excellent [Project Gutenberg](https://www.gutenberg.org/)<sup>10</sup>, and – most importantly – all are now in the public domain.

You will probably get slightly different results with different source material.

I have gone back to creating objects and drivers, so you’ll also need to type in the following file for everything to work.

IBeforeEDriver.java

---

```

1  import java.io.FileInputStream;
2
3  public class IBeforeEDriver {
4      public static void main( String[] args ) throws Exception {
5          FileInputStream file = new FileInputStream("datafiles/en-sample.txt");
6          IBeforeEChecker check = new IBeforeEChecker( file );
7          System.out.println( check.getWordCount() + " ie/ei words found." );
8          System.out.print( "\t" + check.getMatches() + " match the rule, " );
9          System.out.print( check.getExceptions() + " do not: " );
10         System.out.println(100.0*check.getMatches()/check.getWordCount() + "%");
11     }
12 }

```

---

## What You Should See

```

3417 ie/ei words found.
      2675 match the rule, 742 do not: 78.28504536142815%

```

---

<sup>10</sup><https://www.gutenberg.org/>

For this sample, the rule works 78% of the time.

Now let's talk about how the code works and *why* it is arranged the way it is.

I think the driver is hopefully pretty clear. I decided the driver should have to pass in an `InputStream` object through the constructor, because you can get an `InputStream` object from a local text file (using `FileInputStream`) or from a URL (using `java.net.URL`'s `openStream()` method).

That way the `IBeforeEChecker` object doesn't need to know or care where the data is coming from. This makes the object more versatile, and easier for others to use. You need to think about these things when designing objects.

The driver instantiates a `IBeforeEChecker` object on line 6 of the runner. I wanted this to finish quickly, so that the object doesn't spend the time to calculate the statistics until a method is called that actually requires that information.

Also notice that this driver calls `getWordCount()` twice. It calls `getMatches()` twice, and it calls `getExceptions()` once. When I was writing the object, I wasn't sure if all the methods would be called or just a few, and I also didn't know what order the methods would be called in.

So here were my goals for creating the object.

- The `IBeforeEChecker` object shouldn't do a ton of work until it's needed.
- It shouldn't duplicate work. Any calculations should only happen once.
- The driver shouldn't have to know to call some magic "do the work" function; this should be handled inside the object.
- The methods should return the right value no matter what order they are called in.
- The methods should return the right value even if only one of them is ever called.
- Calling the methods more than once shouldn't cause work to be repeated.

With those in mind, let's look back at the code for the `IBeforeEChecker` object.

We begin the object with four private variables; these are our instance variables / fields that will be accessible in every method in the class.

The *source* field holds a reference to the `InputStream` object that's passed in to us from the user of this object. Later we'll connect a `Scanner` object to this `InputStream` to read all the words.

The *matchCount* field will hold the number of words that fit the "I before E" rule. It starts out at zero since we haven't done any calculations yet.

The *computed* field is a boolean that will be `false` until we actually read all the words and do the calculations. This variable is going to let us not do the work more than once.

Finally, *wordList* is a reference to an `ArrayList` of `Strings`. All our instance variables are given suitable starting values inside the constructor (lines 12 through 15), which is always a good idea.

## Private Helper Methods

Starting on line 18, we define a private method! `fitsRule()` is going to be a “helper method” that we will use to make our code easier to read and understand, but since it is private the driver won’t be able to call it directly. Only *we* will be able to call it while writing our other methods.

`fitsRule()` takes in a `String` parameter and then returns `true` if that word contains “ie” but not “cie”, or if it contains “cei”.

Starting on line 28 we define a second private helper method called `doCalculations()`. This is a dumb name, but fitting.

This helper method attaches an `Scanner` to the `InputStream` that the user gave us, and then scans through it one word at a time. For each word, we check if the word contains the letters “ie” or “ei”. If so, it gets added to the `ArrayList`. (But only once! Adding duplicate words would skew the statistics.)

After reading all the words we close/detach the `Scanner` from the `InputStream`.

Then, for each ie/ei word in `wordList`, we use our other helper method to see if the word fits the spelling rule. If so, we increment the counter.

Finally we change the `computed` field to `true`. We’ll see why this is important soon.

## Public Methods

Our first public method is `getMatches()`, which returns the number of words that match the spelling rule. It *could* just return `matchCount`, but that variable is only valid if the calculations have been done!

So, we check the `computed` variable. If it is not true that means none of the other methods have run `doCalculations()` yet. So we first run the helper method, which does all the work and *sets computed to true*.

Then it’s safe to return `matchCount`. Cool, huh?

`getExceptions()` is very similar. Remember that we don’t know what order these methods are going to be called in, so this method *also* needs to check the `computed` field. Once we’re sure the work is done, we do a common trick.

We call the `getWordCount()` method to get the total number of ie/ei words then subtract the number of matches to get the number of exceptions. And we return that.

There is also a public method called `getWordList()`, which returns a reference to the `ArrayList` of ie/ei words. The driver code doesn’t currently use this method, but when I was writing the object I figured that somebody might want to be able to get this list.

I think it’s important to restate: all the instance variables / fields are *private*, so the user of my object can’t access them. If I were to make javadoc documentation from this object file, the private variables wouldn’t even be mentioned.

Also, the private helper methods are just to make *my* job easier. The person using this object doesn't know about them and doesn't have to mess with them. The constructor and the *public* methods are all they need and they Just Work.

Designing your code in an object-oriented way is a bit more work. But then you can make a module like this that turns tricky code into something easy to use. And this is why people bother.



## Study Drills

1. Modify the driver code so that it prints out the list of ie/ei words.
2. Modify the driver code so that it can read from a URL instead of a local file.



# Exercise 23: Writing a Silly Class with Generics

A couple of exercises ago we looked at how generics in Java allow us to *use* an ArrayList without having to put casts every time we retrieve a value out of it.

So this time we're going to see how to use a generic type in a class *we* are creating. There are four files in this exercise. The first two you should probably skip typing in, though you'll need to copy mine because the driver expects them to be there.

The third one (PointSwapper) is where the magic happens, and you should definitely type it yourself.

PointSwapperInt.java

---

```
1 public class PointSwapperInt {
2     private int a, b;
3
4     public PointSwapperInt( int a, int b ) {
5         this.a = a;
6         this.b = b;
7     }
8
9     public void swap() {
10         int temp = a;
11         a = b;
12         b = temp;
13     }
14
15     public int getFirst() { return a; }
16     public int getLast() { return b; }
17
18     public String toString() {
19         return "(" + a + ", " + b + ")";
20     }
21 }
```

---

I'm not sure why anyone would want such a thing, but let's pretend that for some reason you want a class that represents a point (like x & y) *and* that you want to be able to easily make x and y trade places.

I don't know; just work with me here.

So at first we decide the two values should be integers. So our private instance variables / fields are integers. The constructor receives two integers as parameters.

We have a `swap()` method, and the temporary variable inside the method is an integer as well.

Our `getFirst()` method *returns* an integer. `getLast()` also returns an integer.

All these aren't integers for any good reason; they are *only* integers because our point is made of integers.

Okay, so now let's pretend that we also want to be able to make a swappable point out of floating-point values. Here's *that* code!

PointSwapperDouble.java

---

```
1 public class PointSwapperDouble {
2     private double a, b;
3
4     public PointSwapperDouble( double a, double b ) {
5         this.a = a;
6         this.b = b;
7     }
8
9     public void swap() {
10        double temp = a;
11        a = b;
12        b = temp;
13    }
14
15    public double getFirst() { return a; }
16    public double getLast() { return b; }
17
18    public String toString() {
19        return "(" + a + ", " + b + ")";
20    }
21 }
```

---

Notice how similar the code is! Except all the integers are doubles instead. It's a shame how much code has to be duplicated when the only difference is the *type* of the coordinates in the point.

So that's where generics come to the rescue. Here's an example of the same code, but with the type made *generic*.

## PointSwapper.java

---

```
1 public class PointSwapper<Type> {
2     private Type a, b;
3
4     public PointSwapper( Type a, Type b ) {
5         this.a = a;
6         this.b = b;
7     }
8
9     public void swap() {
10        Type temp = a;
11        a = b;
12        b = temp;
13    }
14
15    public Type getFirst() { return a; }
16    public Type getLast() { return b; }
17
18    public String toString() {
19        return "(" + a + ", " + b + ")";
20    }
21 }
```

---

So notice that after the `public class` line there are angle brackets with a word in them. I have chosen to put the word “Type” in there, but it’s just an identifier. You could have put anything in there and it would work as long as you’re consistent with it. (Most Java programmers would prefer that it start with a capital letter, though.) Using just `<T>` is pretty common, and so is `<E>` (for “element”).

You’ll see that every time I would have said `int` or `double`, I just put the name of the type parameter (also called a “type variable”) instead: `Type`. Easy!

Okay, so here’s the driver which runs all three objects.

## PointSwapperDriver.java

---

```
1 public class PointSwapperDriver {
2     public static void main( String[] args ) {
3         PointSwapperInt pi = new PointSwapperInt(3, 5);
4
5         System.out.println( "before: " + pi );
6         pi.swap();
7         System.out.println( "after: " + pi );
8         pi.swap();
9     }
10 }
```

---

```
9      System.out.println( "after after: " + pi );
10
11      PointSwapperDouble pd = new PointSwapperDouble(1.1, 4.4);
12
13      System.out.println( "\nbefore: " + pd );
14      pd.swap();
15      System.out.println( "after: " + pd );
16      pd.swap();
17      System.out.println( "after after: " + pd );
18
19      PointSwapper<Integer> pg1 = new PointSwapper<>(2, 6);
20      PointSwapper<Double> pg2 = new PointSwapper<>(1.3, 5.7);
21
22      System.out.println( "\nbefore: " + pg1 );
23      pg1.swap();
24      System.out.println( "after: " + pg1 );
25      pg1.swap();
26      System.out.println( "after after: " + pg1 );
27
28      System.out.println( "\nbefore: " + pg2 );
29      pg2.swap();
30      System.out.println( "after: " + pg2 );
31      pg2.swap();
32      System.out.println( "after after: " + pg2 );
33  }
34 }
```

---

The first few lines of the driver just create a `PointSwapperInt` object and test it out. Then a `PointSwapperDouble` object is instantiated and tested.

The fun stuff begins on line 19. Two `PointSwapper` objects are created; the first one passes in the `Integer` wrapper class as the type parameter and the second one passes in the `Double` class. (Note that generics only work with objects; you couldn't just pass in `<int>`.)

This is pretty nice, because we only had to write a single class, and it will work with any type! Generics!

## What You Should See



```
before: (3, 5)
after: (5, 3)
after after: (3, 5)

before: (1.1, 4.4)
after: (4.4, 1.1)
after after: (1.1, 4.4)

before: (2, 6)
after: (6, 2)
after after: (2, 6)

before: (1.3, 5.7)
after: (5.7, 1.3)
after after: (1.3, 5.7)
```

Unfortunately, generics are not as easy to work with in Java as they are in other languages like C#. We'll get a taste of this in the next exercise when we try to implement a *useful* class using generics.



## Study Drills

1. Add a comparison method to the PointSwapper class so the user of the class (the driver) can pass in two `Type` values as parameters. Have the new method return `true` if the values passed in are the same as *a* and *b*. Also add some code to the driver to test your new method.

# Exercise 24: Writing a Useful Class with Generics

In the previous exercise I created a silly, silly class that used generics.

So in this exercise we are going to write a *real*, useful class that uses generics. Most of the code is pretty straightforward except for some weirdness where Java won't let you instantiate an array with a generic type.

This class works a lot like Java's built-in ArrayList class. It works so much like it, in fact, that you could use this class instead of `java.util.ArrayList` for any of the exercises so far.

JavaHardArrayList.java

---

```
1 public class JavaHardArrayList<E> {
2     private E[] arr;
3     private int used;
4
5     @SuppressWarnings("unchecked")
6     public JavaHardArrayList() {
7         used = 0;
8         // arr = new E[10]; // doesn't compile "error: generic array creation"
9         arr = (E[])new Object[10];
10    }
11
12    public int size() {
13        return used;
14    }
15
16    public boolean add(E obj) {
17        if ( used == arr.length )
18            grow();
19        arr[used] = obj;
20        used++;
21        return true;
22    }
23
24    public void add(int index, E obj) {
25        if ( used == arr.length )
26            grow();
```

```
27         // move the other values over to make space
28         for ( int i=used; i>index; i-- )
29             arr[i] = arr[i-1];
30         arr[index] = obj;
31         used++;
32     }
33
34     public E get(int index) {
35         if ( index < used ) {
36             return arr[index];
37         }
38         return null;
39     }
40
41     public E set(int index, E obj) {
42         if ( index < used ) {
43             E old = arr[index]; // make a copy since set() returns old value
44             arr[index] = obj;
45             return old;
46         }
47         return null;
48     }
49
50     public E remove(int index) {
51         if ( index < used ) {
52             E temp = arr[index];
53             // slide everything back one slot
54             for ( int i=index; i<used; i++ )
55                 arr[i] = arr[i+1];
56             used--;
57             return temp;
58         }
59         return null;
60     }
61
62     public boolean contains(E obj) {
63         for ( int i=0; i<used; i++ )
64             if ( arr[i].equals(obj) )
65                 return true;
66         return false;
67     }
68
```

```

69     public String toString() {
70         String out = "";
71         if ( used > 0 )
72             out += arr[0];
73         for ( int i=1; i<used; i++ )
74             out += ", " + arr[i];
75         return "[" + out + "]";
76     }
77
78     private void grow() {
79         int newCapacity = 1 + 2*arr.length;
80         @SuppressWarnings("unchecked")
81         // E[] arr2 = new E[newCapacity]; // "error: generic array creation" :(
82         E[] arr2 = (E[])new Object[newCapacity];
83         for ( int i=0; i<arr.length; i++ )
84             arr2[i] = arr[i];
85         // move the reference so that arr now refers to the new larger array
86         arr = arr2;
87     }
88
89 }

```

---

You'll note that I was *almost* able to just put `<E>` after the name of the class and then put `E` everywhere that I would have put `Integer` or `Double`.

The only exceptions are lines 8/9 and 81/82. I *wanted* to just create an array of `E` objects, but that fails to compile with a “generic array creation” error.

So instead I had to instantiate an array of `Objects` and then cast the reference into an array of `E` instead. Which is weird, but apparently what you must do in Java. Also I had to stick the `SuppressWarnings` annotation in there to avoid the “unchecked or unsafe operations” warning when compiling.

But before I get too far into explaining the methods, here is the driver, which is just `ListsOfPrimitives.java` modified to use this `ArrayList` class instead of the one from the Java Standard Library.



## JavaHardArrayListDriver.java

---

```
1 public class JavaHardArrayListDriver {
2     public static void main(String[] args) {
3         JavaHardArrayList<String> hats = new JavaHardArrayList<>();
4
5         hats.add("fez");
6         hats.add("bowler");
7         hats.add("beanie");
8         hats.add("western");
9         hats.add("fedora");
10
11        System.out.println( hats );
12
13        String jumble = "";
14        for ( int i=0; i<hats.size(); i++ ) {
15            jumble += hats.get(i);
16        }
17        System.out.println("All together now: " + jumble);
18
19        JavaHardArrayList<Integer> bins = new JavaHardArrayList<>();
20
21        bins.add(new Integer(1));
22        bins.add(new Integer(3));
23        bins.add(new Integer(3));
24        bins.add(new Integer(1));
25
26        bins.add(1);
27        bins.add(4);
28        bins.add(6);
29        bins.add(4);
30        bins.add(1);
31
32        System.out.println( bins );
33
34        int total = 0;
35        for ( int i=0; i<bins.size(); i++ ) {
36            total += bins.get(i);
37        }
38        System.out.println("The total is " + total);
39
40        JavaHardArrayList<Character> letters = new JavaHardArrayList<>();
41        letters.add('z'); // auto-boxes char
```

```
42
43     JavaHardArrayList<Double> weights = new JavaHardArrayList<>();
44     weights.add(0.14); // auto-boxes double
45
46     JavaHardArrayList<Boolean> dealt = new JavaHardArrayList<>();
47     while ( dealt.size() < 52 )
48         dealt.add(false); // auto-boxes boolean
49 }
50 }
```

---

(I guess also took out a couple of lines of code just to make it a smidge shorter.)

## What You Should See

```
[fez, bowler, beanie, western, fedora]
All together now: fezbowlerbeaniewesternfedora
[1, 3, 3, 1, 1, 4, 6, 4, 1]
The total is 24
```

The goal of this exercise is to see a *real* example of generics used in a useful way. And you've done that already. So you could skip all of this text and just move on to the next exercise if you want.

However, theoretically you have all the background knowledge you need to understand all the code in this exercise. And some of you may be interested in the details, so I'm going to explain it all.

But don't feel bad about skipping ahead if you want. Just because I'm taking the time to write this doesn't mean that I think you need to read it. Really!

## The Idea

Arrays are cool. They're fast and easy to access. But they don't grow! Once an array has been allocated, its size is fixed and can't be changed. But we'll handle that problem later.

We'll start with a fixed-size array of ten elements. There's currently nothing in the ArrayList, but there's an array with ten slots ready to go. And we will also use a separate variable to keep track of how many values are currently stored in the array.

So at first, the picture looks like this:

arr									
0	1	2	3	4	5	6	7	8	9
?	?	?	?	?	?	?	?	?	?
used									

Since *used* begins with the value zero, we can imagine that *used* is “pointing” at the next available slot in the array. That is, I’m using the variable to count *how many* values are already in the array *and also* to refer to the next available index.

This works out since array indexes (technically “indices”) start with zero instead of one.

Actually, I’m lying. An array of references like *arr* is **not** filled with question marks. Arrays in Java are automatically filled with ‘default’ values when they are created, and in the case of arrays of references they are filled with `null`, which is the value a reference has when it doesn’t (yet) refer to anything. So here is a more accurate picture of *arr*:

arr: @2b8									
0	1	2	3	4	5	6	7	8	9
null	null	null	null	null	null	null	null	null	null
used									

Also! Remember that an array variable is a reference in the symbol table. So I have also added the address that the variable *arr* refers to. You can see that – for now – the reference variable *arr* holds the address @2b8. Okay, let’s proceed.

Let’s pretend now that the user of this class calls the `add()` method with some value they want to add to the list.

*used* is zero, and `arr.length` is ten, so the `if` statement at the beginning of the `add()` method is false. Thus we store the value they gave us into index *used* a.k.a. index 0. Then *used* gets incremented to 1. Now the picture looks like this:

arr: @2b8									
0	1	2	3	4	5	6	7	8	9
@e50	null	null	null	null	null	null	null	null	null
used									

The first slot of the array now holds a reference to whatever they gave us, and *used* is “pointing” to the next available slot in the array.

So that’s the basic idea. Now let us look at each of the methods, one at a time. If you want you can skip down to `grow()`, which is quite interesting.

## Method Details

We start with two private instance variables / fields: *arr* and *used*. *arr* is an array of references to *E* objects, whatever type that ends up being. And *used* is an integer that keeps track of how many slots in the array are currently filled. It also “points” to the next free slot in the array.

The **constructor** is next, and it just sets *used* to 0 and allocates an array of ten *E* objects in the weird way that Java makes us do it.

The `size()` method is supposed to return the count of items currently stored in the list, and that’s stored in the *used* field. So it’s as easy as returning a copy of the value.

`ArrayLists` (and all `Lists`, actually) have *two* `add()` methods. The first adds the new item at “the end” of the `List`. This method is also supposed to return `true` if the `add` “succeeds”. (Some kinds of `Lists` in Java might fail to add items, but in this case we’re just going to always return `true`.)

If the array is “full”, then we have to magically “grow” the array to make it bigger. (More on that later!) Then either we weren’t full to begin with or we *now* have enough space, so we put the reference to the new item in the next available slot in the array and then increment our counter. And always return `true`.

The second `add()` method receives a specific index where the new item should be added. Just like before, we first check if the array is full and magically grow if needed.

Then, make a “hole” for the new item. Start at the last filled slot and copy each value into the location to its right. (`arr[i-1]` is the slot to the left of `arr[i]`, so each value moves right.)

We loop from the end down to *index*, which is where they want the new value to be added. So after that loop finishes, all the values have been moved over, and slot *index* is ready to receive the new value. We put in it there, and increment *used* and return `true`.

(I probably should have checked to make sure the new *index* isn’t bigger than the current size of the list. Oops! I’ll have you do that in the Study Drill.)

The `get()` method is easier; it just has to return the value in the specified *index*. We check to make sure the requested index isn’t “out of bounds”, and return the value.

When the requested index *is* out of bounds then nothing gets returned yet so the `return null` happens instead. If I were cooler I would have figured out a way to throw an “`IndexOutOfBoundsException`” exception like the real `ArrayList` class does. But this code is already complicated enough.

The `set()` method replaces the value at the given index, and then returns whatever value was previously in that slot. Again we check to make sure the value they’re replacing isn’t out of bounds. Then we make a copy of the existing value in that slot, overwrite it, then return the old value.

The `remove()` method is like the inverse of the `add()` method; it moves all the values back one slot to the left so there’s not a gap in the array. And since a value has been removed, it decrements *used*. This method also returns a copy of the value that got removed, since that how Java’s `ArrayList` class works.

The `contains()` method is pretty simple; we just look through each slot one at a time. Notice that since we're dealing with objects, we have to compare using `equals()`. If we find the specified object, we return `true`, and if the loop finishes then we must not have found it and we can return `false`.

Also notice that the loop only counts up to *used*, not `arr.length`. The array probably isn't full, so looking at slots with nothing in them is trouble.

The real `ArrayList` class has a nice `toString()` method that makes it easy to print. It creates a `String` with all the values, in order, with commas between and then wraps the whole thing in square brackets.

## The magical `grow()` method

Finally we have reached the magical `grow()` method which makes the array larger. Even though arrays in Java can't be resized. So let's look at a "full" array, where `used == arr.length`.

arr: @2b8									
0	1	2	3	4	5	6	7	8	9
@e50	@660	@540	@b70	@c08	@fe0	@4a8	@3b0	@580	@960
									used

`arr.length` is 10, and *used* is 10, so the array is full.

In the code, we create a second, larger array with a capacity of 21. (In the diagram I'm only going to make the new capacity 15 so it will fit on the page.)

Then we copy *all* of the values from the original smaller array into the new larger array. It would look like this:

arr2: @4c0														
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
@e50	@660	@540	@b70	@c08	@fe0	@4a8	@3b0	@580	@960	null	null	null	null	null

Now for the magic. Remember that in the symbol table, an array variable just holds a reference. If we assume that at the moment our symbol table looks like this:

symbol table		
identifier	address	type
arr	@17400100	reference to E[]
used	@17400096	int
newCapacity	@17400092	int
arr2	@17400084	reference to E[]

symbol table		
identifier	address	type

And – more importantly – if the stack currently looks like this:

stack				
17400100	...096	...092	...088	...084
@2b8	10	14		@4c0

Here's the magic line of code, which looks very boring:

```
arr = arr2;
```

Did you see it?!? We just copied the shallow value of `arr2` – its reference – into `arr`, **overwriting** its previous value! So now the stack looks like this instead:

stack				
17400100	...096	...092	...088	...084
@4c0	10	14		@4c0

At this point, both `arr` and `arr2` hold the **same** reference: the reference to the larger array! And *no* variable holds the old reference to the previous capacity-10 array (@2b8)!

After the `grow()` method finishes running, its local variables (`newCapacity` and `arr2`) go out of scope and their entries are removed from the symbol table and their values are popped off the stack. They now look like this:

symbol table		
identifier	address	type
arr	@17400100	reference to E[]
used	@17400096	int

stack	
17400100	...096
@4c0	10

Which means that the array *arr* now looks like so:

arr: @4c0														
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
@e50	@660	@540	@b70	@c08	@fe0	@4a8	@3b0	@580	@960	null	null	null	null	null

(Well, really it would have 21 slots, but that won't fit on the page.)

Pretty cool trick, huh? Since array variables are really just references we could “swap out” a reference from a smaller array with a reference to a bigger one. The symbol table didn't even have to change because *arr* is still the same variable; it just has a different reference stored in it.

If you have previously programmed in a language like C or C++ you might be wondering what happened to the memory allocated for the old smaller array! In these languages, dynamically-allocated memory in the heap must be manually “freed” or “deleted” to give it back to the OS when you are done with it, and if you forget you have a “memory leak.” But in Java you don't have to do this. Once a chunk of dynamically-allocated memory no longer has any references to it, that memory gets automatically freed by something called the “garbage collector”. There are trade-offs, but garbage-collected languages are pretty nice.

Okay, so that's everything.

There are quite a few more methods you would have to implement to duplicate the “real” `ArrayList` class. You would also want to inherit from `AbstractList` and implement a half-dozen interfaces – especially the `List` interface! Oh, and throw `Exceptions` instead of just returning `null`.

But we haven't covered those things yet. Also, you don't *have* to since Java provides a perfectly good `ArrayList` class and we can just use that!



## Study Drills

1. Modify the second `add()` method, so that it will only add the new item if the requested *index* is **less than or equal to** *used*. Otherwise it should do nothing. (Or, if you want, you can figure out how to get it to throw an `IndexOutOfBoundsException` exception.)

# Exercise 25: Sorting and Complexity

In this exercise we will do another example that uses generics. But mostly we are going to talk about sorting and about what programmers mean when they say that one piece of code is more “efficient” than another.

This class receives an array of some kind of number (Integers or Doubles or really anything Comparable) and can say if they are in order. It can also *put* them in order (“sort” them) in two different ways.

I apologize in advance for the first line of the class. Yes, that all has to be there.

ThingySorter.java

---

```
1 public class ThingySorter<E extends Comparable<E>> {
2
3     private E[] arr;
4
5     public ThingySorter( E[] arr ) {
6         this.arr = arr;
7     }
8
9     private void swap( int i, int j ) {
10         E temp = arr[i];
11         arr[i] = arr[j];
12         arr[j] = temp;
13     }
14
15     public void exchangeSort() {
16         for ( int i=0; i<arr.length-1; i++ )
17             for ( int j=i+1; j<arr.length; j++ )
18                 if ( arr[i].compareTo(arr[j]) > 0 )
19                     swap(i,j);
20     }
21
22     public void quickSort() {
23         quickSortHelper(0, arr.length-1);
24     }
25
26     private void quickSortHelper( int lo, int hi ) {
27         if ( lo < hi ) {
```



```

28         int pivotloc = partition(lo, hi);
29         quickSortHelper(lo, pivotloc-1);
30         quickSortHelper(pivotloc+1, hi);
31     }
32 }
33
34 private int partition( int lo, int hi ) {
35     int pi = (lo+hi)/2;
36     E pivot = arr[pi];
37     swap(pi,hi);
38     int cur = lo;
39     for ( int i=lo; i<hi; i++ ) {
40         if ( arr[i].compareTo(pivot) < 0 ) {
41             swap(i,cur);
42             cur++;
43         }
44     }
45     swap(cur,hi);
46     return cur;
47 }
48
49 public boolean isSorted() {
50     for ( int i=0; i<arr.length-1; i++ ) {
51         if ( arr[i].compareTo(arr[i+1]) > 0 )
52             return false;
53     }
54     return true;
55 }
56 }

```

---

Hm. That first line.

When I was first creating this assignment, what I had *hoped* to be able to write for the first line of the class was:

```
public class ThingySorter<E> {
```

But there's a problem with that. On line 18 (and 40) I take an object of type E and call `compareTo()` on it. And not all objects in Java *have* a `compareTo()` method!

So then I wanted to write:

```
public class ThingySorter<E implements Comparable> {
```

...which ought to mean “let *E* be any type that implements the Comparable interface.” But that doesn’t work either. (More on interfaces in a future exercise.) It turns out that even though Comparable is an interface and not a superclass, you would write:

```
public class ThingySorter<E extends Comparable> {
```

But that doesn’t quite work either because Comparable itself needs a generic type (parameterized type). So you must write:

```
public class ThingySorter<E extends Comparable<E>> {
```

Which basically means: the ThingySorter class needs a type parameter. We’ll call the parameter *E*. But the type supplied must be one that implements the Comparable interface, and it must be comparable with *other* objects of type *E*. Okay?

Good enough.

(If you really wanted to be fancy you would write:

```
public class ThingySorter<E extends Comparable<? super E>> {
```

...which means that the type must be comparable with other objects of type *E* or *any of E’s subclasses*. Again, don’t worry too much about that because we will be talking more about subclasses, superclasses, inheritance, interfaces and implementing in future exercises.)

ThingySorterDriver.java

---

```
1 import java.util.Arrays;
2 import java.util.Random;
3
4 public class ThingySorterDriver {
5     public static void main( String[] args ) {
6
7         long start, stop;
8         Integer[] inums = {68, 65, 5, 56, 83, 17, 4, 9, 90, 78, 42, 10, 91, 34};
9         Double[] dnums = {12.1, 45.7, 43.9, 7.0, 1.5, 13.7, 19.6, 49.1, 29.7};
10
11         ThingySorter<Integer> sort1 = new ThingySorter<>(inums);
12         System.out.println( Arrays.toString(inums) );
13         System.out.println( "Sorted? " + sort1.isSorted() );
14         sort1.exchangeSort();
```

```
15      System.out.println( Arrays.toString(inums) );
16      System.out.println( "Sorted? " + sort1.isSorted() + "\n" );
17
18      ThingySorter<Double> sort2 = new ThingySorter<>(dnums);
19      System.out.println( Arrays.toString(dnums) );
20      System.out.println( "Sorted? " + sort2.isSorted() );
21      sort2.exchangeSort();
22      System.out.println( Arrays.toString(dnums) );
23      System.out.println( "Sorted? " + sort2.isSorted() );
24
25      Random gen = new Random();
26      Integer[] inums2 = new Integer[50000];
27      Integer[] inums2copy = new Integer[50000];
28      for ( int i=0; i<inums2.length; i++ ) {
29          inums2[i] = gen.nextInt(1000);
30          inums2copy[i] = inums2[i];
31      }
32
33      ThingySorter<Integer> sort2a = new ThingySorter<>(inums2);
34      System.out.print( "\n" + inums2.length + " Integers sorted? " );
35      System.out.println( sort2a.isSorted() );
36      System.out.print( "Sorting..." );
37      start = System.nanoTime();
38      sort2a.exchangeSort();
39      stop = System.nanoTime();
40      System.out.println( "done." );
41      System.out.print( inums2.length + " Integers sorted? " );
42      System.out.println( sort2a.isSorted() );
43      System.out.print( "Exchange sorting them took " );
44      System.out.println( (stop-start)/1000/1000.0 + " milliseconds." );
45
46      ThingySorter<Integer> sort2b = new ThingySorter<>(inums2copy);
47      System.out.print( "\n" + inums2copy.length + " Integers sorted? " );
48      System.out.println( sort2b.isSorted() );
49      System.out.print( "Sorting..." );
50      start = System.nanoTime();
51      sort2b.quickSort();
52      stop = System.nanoTime();
53      System.out.println( "done." );
54      System.out.print( inums2copy.length + " Integers sorted? " );
55      System.out.println( sort2b.isSorted() );
56      System.out.print( "Quick sorting them took " );
```

```
57         System.out.println( (stop-start)/1000/1000.0 + " milliseconds." );
58     }
59 }
```

---

The ThingySorter can put values in order and it can do it using two different techniques. An “exchange sort” has *much* simpler code – only four lines! The so-called “quick sort” is a lot more complicated: it requires three different methods and one of the methods is recursive!

(A recursive method is a method that runs *itself* with a slightly smaller version of the original task in order to get things done.)

So why would we bother writing the quick sort code? You’ve probably already seen the results, so you probably already know.

## What You Should See

```
[68, 65, 5, 56, 83, 17, 4, 9, 90, 78, 42, 10, 91, 34]
Sorted? false
[4, 5, 9, 10, 17, 34, 42, 56, 65, 68, 78, 83, 90, 91]
Sorted? true

[12.1, 45.7, 43.9, 7.0, 1.5, 13.7, 19.6, 49.1, 29.7]
Sorted? false
[1.5, 7.0, 12.1, 13.7, 19.6, 29.7, 43.9, 45.7, 49.1]
Sorted? true

50000 Integers sorted? false
Sorting...done.
50000 Integers sorted? true
Exchange sorting them took 6650.047 milliseconds.

50000 Integers sorted? false
Sorting...done.
50000 Integers sorted? true
Quick sorting them took 18.701 milliseconds.
```

The exchange sort method – at least on my computer – takes more than three hundred (300!) times longer to sort the exact same list of 50,000 integers! And the gap widens the more values you try to sort: on my computer, doubling the list to 100,000 integers made the exchange sort eleven hundred (1100) times slower than the quick sort!!

This is due to the “algorithmic complexity” of the two sort methods. But we’ll talk more about that in a bit. First let me explain the code in the class/object and in the driver.

The constructor is pretty boring, since we only have a single instance variable / field. They pass in an array of “Elements” and so we make a copy of it. Remember that array variables hold *references*, though, so there’s not actually a copy of the array values being made. Our field just holds the same reference to the array on the heap as what they passed in. (This will be important later.)

The `swap()` method is just a little private helper method to make the rest of our code easier to read; we will send in two indexes corresponding to the slots of the array to exchange. Then we make the values trade places, so that the value which *was* in slot *i* ends up in slot *j* and vice-versa.

Notice that our temporary variable has the same type as whatever is in the array: E.

The `exchangeSort()` method is short and sweet. We make *i* go from the front of the array to the back, then *j* starts with the slot after *i* and also goes to the back. If the value in slot *i* is larger than the value in slot *j*, then the values are out of order and should swap places. Done.

You should notice that these are *nested* loops, and each one counts up to the end. That means if *N* is the number of elements in the array, then the `if` statement inside both loops gets checked roughly  $N \times N$  times, or  $N^2$ .

The `quickSort()` method is recursive, so it needs parameters passed in to it that it can use to keep track of which part of the array it is working on at the moment. But the driver class doesn’t know that (and shouldn’t *need* to know that), so this public method just fills in the blanks and calls the recursive version of the method.

Don’t worry if you don’t completely follow the details of the recursive quicksort at this time. It’s a tricky bit of code and getting all the details right took *me* a while.

The basic idea is easy, though. Let’s pretend you are the recursive quicksort method. You are asked to make sure that all the values in the array between *lo* and *hi* are in the correct order.

You pick a “pivot” value, and then shuffle the values around so that everything smaller than the pivot is moved to the first half of the array, and everything bigger than the pivot is moved to the second half of the array. Then, you ask the recursive quicksort method to sort the first half. And finally you get the recursive quicksort method to sort the second half for you.

Here’s a brief example, sorting nine values:

arr								
0	1	2	3	4	5	6	7	8
2	1	8	9	4	7	5	6	3
lo								
							hi	

At first, the array isn’t sorted, *lo* is “pointing” at the first slot in the array, and *hi* is pointing at the last slot in the array. In our code, we choose the pivot halfway between *lo* and *hi*. Choosing a pivot

value in a better way can really affect the speed of the quicksort, but I just wanted to do something easy that would work fine.

So in this case it'll choose the pivot value 4, which happens to be in slot 4. Before the `partition()` method goes, here's the original array again. I have highlighted the pivot value it will choose.

arr									
0	1	2	3	4	5	6	7	8	
2	1	8	9	4	7	5	6	3	
lo								hi	

The call `partition(0,8)` moves all the values smaller than 4 to the front of the array, and all the values bigger than 4 to the back. After it finishes, the array will look like this:

arr									
0	1	2	3	4	5	6	7	8	
2	1	3	4	8	7	5	6	9	
lo				p				hi	

Then the quicksort helper will go again: twice. It will partition slots 0-2 and then partition slots 4-8.

The code for the quicksort is definitely more complicated, but it does a lot less comparing overall. Everything in the array just got compared with the pivot value, but the values now in the front (2, 1, 3) didn't get compared with the values in the back (8, 7, 5, 6, 9)... and *they never will*.

In fact, each time the `partition()` method runs, it cuts the number of comparisons remaining in half. So instead of  $N \times N$  comparisons, it ends up doing  $N$  times the logarithm (base 2) of  $N$  comparisons.

The log (base 2) of  $N$  is basically the exponent you would have to raise 2 to so that you get  $N$ . For example, if  $N$  is 8, then  $\log_2(8)$  is 3, because  $2^3 = 8$ . And if  $N$  is 100, then  $\log_2(100)$  is between 6 and 7, because  $2^6 = 64$  and  $2^7$  is 128. (You can compute the exact value if you have a calculator, because  $\log_2(N)$  equals  $\log_{10}(N) / \log_{10}(2)$ .)

We'll talk more about the implications of this in just a bit, but first I want to finish explaining the rest of the code in today's exercise.

The `isSorted()` method is comparatively easy: it just loops through the array looking for a value that is larger than the one in the slot to its right. If we find even one value like this, the array is *not* sorted.

Okay, let's talk about the driver for just a bit.

The driver begins by creating two arrays of Objects, in random order. It instantiates a `ThingySorter` object and passes the first array into the constructor.

Arrays don't print nicely like ArrayLists do, so there's a helper method inside `java.util.Arrays` that will produce a nice-looking string from an array. We use this method to help us display the array in original order.

We confirm that the array is not sorted using the `isSorted()` method, then sort it, then display it again, and finally confirm the sortedness using `isSorted()` one more time.

Just to prove that `ThingySorter` uses generics properly, we do all the same steps again, but this time with an array of Doubles instead of Integers.

## Benchmarking

We plan to use a timer to see which sorting method is better, so we have to get set up before we start the timer.

There are two common ways to pick a random number in Java. In my previous book I *only* used `Math.random()`. But since we're learning about object-oriented programming, this time I decided to do it using `java.util.Random` from the Standard Library.

You instantiate a `Random` object, then you can call the method `nextInt()`, which will return a random integer from 0 up to (but not including) whatever number you stick in the method call.

So `gen.nextInt(1000)` will return a random integer between 0 and 999.

I create two arrays of Integers, each with a capacity of 50,000. Then I generate 50,000 random numbers and store them into both arrays. Why two copies of the same values?

Remember how in the constructor for `ThingySorter` we passed in a *reference* to an array? And that's the array that gets sorted?

Well, so if we pass in `inums2` into a `ThingySorter` and then sort it, it's not just *arr* that gets sorted, since *arr* and *inums2* are both references to the same physical array in the heap. So we need to make a copy *before* sorting if we want our trial to work on the same numbers.

We instantiate a new `ThingySorter` with the first copy of the 50,000 random numbers. We confirm that it's not sorted. Then we start a timer.

Well, sort of. `System.nanoTime()` returns a really big number. (That's why we have to store the return value into a long; an int isn't big enough.) We don't know much about the number except that it is "the current value of the most precise available system timer, in nanoseconds."

After we put this big number into the variable *start*, we sort the 50,000 integers using the exchange sort. Once the method finishes, we immediately call `System.nanoTime()` again to get another big number.

Subtracting the two numbers gives us the number of nanoseconds between the two calls. Nanoseconds are really small, so before printing out the number I divide it by 1000 to get microseconds, then by 1000 again to get milliseconds. (I throw away decimals the first time but not the second, so we end up with three digits after the decimal.)

Then we do the whole process again with the second copy of the 50,000 random numbers, but this time we sort them using the quicksort. Quicksort is much faster.

## Algorithmic Complexity

So finally let's talk about the algorithmic complexity of the two sorts.

Because the exchange sort ends up comparing  $N$  things with  $N$  things, we say that the time complexity is  $O(n^2)$ . (That's a capital  $O$ , not a zero.)

That's called "Big  $O$  notation", and out loud we would say "order of  $N$  squared" or just "order  $N$  squared".  $O(n^2)$  is also called "quadratic time" and it means that your sorting technique isn't very efficient.

Quadratic time means that if you double the number of values in the array, the amount of comparisons you have to do increases by a factor of four ( $2 \times 2$ ). If you triple the length of the array, the amount of work goes up by a factor of 9 ( $3 \times 3$ ). And so on.

The quicksort partitions things into two "halves" on each pass, and so the number of comparisons ends up being more like  $N \times \log_2(N)$ . Since almost all algorithmic complexity analysis ends up using the log base 2, we just call this  $O(n \log n)$  ("order of  $N \log N$ "). This is commonly referred to as "log linear time" and it's the best you can do when sorting values.

Think about it this way. If the exchange sort is  $O(n^2)$  and  $N$  is 50,000, then it ends up doing like  $50000 \times 50000$  comparisons, which is 2.5 billion!

Quicksort does  $50000 \times \log_2(50000)$ , and the log base 2 of 50,000 is only  $\sim 15.6$ , so it's only doing like 780,000 comparisons!

I don't want to get *too* deeply into Big  $O$  notation, but before I go here's a chart on the most common complexities you tend to see, from best to worst.

Big O		
Big O	name	example
$O(1)$	constant	find first element of an array
$O(\log n)$	logarithmic	binary search of sorted array
$O(n)$	linear	summing array values in an array
$O(n \log n)$	log linear	quicksort
$O(n^2)$	quadratic	exchange sort, bubble sort
$O(2^n)$	exponential	recursive Fibonacci
$O(n!)$	factorial	brute-force traveling salesman

Just a note: constant time means "it takes the same amount of time no matter how big the input is."





## Study Drills

1. Change the driver code so that *both* ThingySorter objects get passed a reference to *inums2*. Do you understand why the second sorter completes way too quickly?
2. Change the length of the array of Integers from 50,000 to different values to confirm that the exchange sort is quadratic and the quicksort is log linear. Make a little table showing the relative times for various values.

# Exercise 26: Sorting Speeds - Primitives vs Objects

In this exercise, we are going to look at the relative speeds of dealing with large numbers of primitives vs large numbers of Objects.

And for the first time in several exercises, there is only one file for you to type in!

SortingSpeeds.java

---

```
1  import java.util.Arrays;
2  import java.util.Random;
3  import java.util.Scanner;
4
5  public class SortingSpeeds {
6      public static void main( String[] args ) {
7          long start, stop, elapsed, slowest = 0, fastest = Long.MAX_VALUE;
8
9          int iterations = 40;
10         int SIZE = 20000;
11         int[]    intArr = new int    [SIZE];
12         Integer[] objArr = new Integer[SIZE];
13
14         Random gen = new Random(555555585);
15         Scanner kb = new Scanner(System.in);
16
17         System.out.println( "1) Primitive ints using selection sort" );
18         System.out.println( "2) Integer objects using selection sort" );
19         System.out.println( "3) Integer objects using Arrays.sort()" );
20         System.out.print( "Which one? " );
21         int choice = kb.nextInt();
22
23         String testDescription = "error";
24
25         for ( int runs=0; runs<iterations; runs++ ) {
26
27             for ( int i=0; i<SIZE; i++ ) {
28                 intArr[i] = gen.nextInt(1000);
29                 objArr[i] = new Integer(intArr[i]);
```

```

30     }
31
32     if ( choice == 1 ) {
33         testDescription = "primitive ints using selection sort";
34         start = System.nanoTime();
35         selectionSortPrimitives(intArr);
36         stop = System.nanoTime();
37     }
38     else if ( choice == 2 ) {
39         testDescription = "Integer objects using selection sort";
40         start = System.nanoTime();
41         selectionSortObjects(objArr);
42         stop = System.nanoTime();
43     }
44     else { // choice == 3, probably
45         testDescription = "Integer objects using Arrays.sort()";
46         start = System.nanoTime();
47         Arrays.sort(objArr);
48         stop = System.nanoTime();
49     }
50     elapsed = (stop-start)/1000000;
51
52     if ( runs > 10 && elapsed < fastest )
53         fastest = elapsed;
54     if ( runs > 10 && elapsed > slowest )
55         slowest = elapsed;
56 }
57
58 System.out.print( "\n" + iterations + " runs sorting " + SIZE + " " );
59 System.out.println( testDescription );
60 System.out.println( "\tFastest time: " + fastest + " milliseconds." );
61 System.out.println( "\tSlowest time: " + slowest + " milliseconds." );
62 }
63
64 public static void selectionSortPrimitives( int[] arr ) {
65     for ( int i=0; i<arr.length-1; i++ ) {
66         int min = i;
67         for ( int j=i+1; j<arr.length; j++ ) {
68             if ( arr[j] < arr[min] )
69                 min = j;
70         }
71         if ( min != i ) {

```

```

72         int temp = arr[i];
73         arr[i] = arr[min];
74         arr[min] = temp;
75     }
76 }
77 }
78
79 public static void selectionSortObjects( Integer[] arr ) {
80     for ( int i=0; i<arr.length-1; i++ ) {
81         int min = i;
82         for ( int j=i+1; j<arr.length; j++ ) {
83             if ( arr[j].compareTo(arr[min]) < 0 )
84                 min = j;
85         }
86         if ( min != i ) {
87             Integer temp = arr[i];
88             arr[i] = arr[min];
89             arr[min] = temp;
90         }
91     }
92 }
93 }

```

---

The basic idea of this code is pretty simple. We create two arrays of 20,000 numbers, but make one of them an array of primitive ints and the other an array of references to Integer objects.

We sort both to see the difference in execution times. We're using the *same*  $O(n^2)$  sort for both arrays, and they both have the same numbers in them. It's just that one contains objects and the other doesn't.

In the previous exercise, we used a quadratic-time sort called the "exchange sort", which is very easy to write. This time, I have opted to use a different quadratic sort called the "selection sort".

The selection sort looks through the unsorted portion of the array to find the smallest value. It then swaps that value with the value in the "first" slot. This does a bit less swapping than the exchange sort but is still pretty bad.

Okay, here are the results on my computer and then I'll talk about the code a bit more.

## What You Should See

First run: primitives.

```
1) Primitive ints using selection sort
2) Integer objects using selection sort
3) Integer objects using Arrays.sort()
Which one? 1

40 runs sorting 20000 primitive ints using selection sort
    Fastest time: 80 milliseconds.
    Slowest time: 86 milliseconds.
```

Second run: objects.

```
1) Primitive ints using selection sort
2) Integer objects using selection sort
3) Integer objects using Arrays.sort()
Which one? 2

40 runs sorting 20000 Integer objects using selection sort
    Fastest time: 144 milliseconds.
    Slowest time: 157 milliseconds.
```

Third time's a charm: objects using the Java standard library's sort function instead of the lame one that I wrote:

```
1) Primitive ints using selection sort
2) Integer objects using selection sort
3) Integer objects using Arrays.sort()
Which one? 3

40 runs sorting 20000 Integer objects using Arrays.sort()
    Fastest time: 2 milliseconds.
    Slowest time: 10 milliseconds.
```

Dang.

Okay, let's talk about the code a bit.

Testing code in this way is called "micro-benchmarking". It's hard to do well, and it's easy to measure the wrong things. Doing it properly is beyond the scope of this book, but at least I will try to avoid some of the more obvious pitfalls, even though it makes the code a bit more complicated.

The first thing we will do is test the same code a bunch of times. I'm only doing it 40 times, since I don't want you to have to wait *minutes* for your tests to complete, but in real life we'd want to do this 10,000 or 100,000 times to have valid results.

Secondly, because of the way the Java runtime works, the "HotSpot Virtual Machine" is still analyzing your code and making optimizations while the code is running. So it is smart to ignore the first dozen runs or so. This is called the "warmup phase." You don't see it until line 52, but I'm not including any results until after the first ten runs.

Thirdly, I'm using a "seed" for the random number generator so that we're using the *same* random numbers to test each approach. Some lists of numbers might already be mostly sorted, so using the same one every time avoids one approach looking better just because it got "lucky" numbers.

After creating all the variables we print a little menu on the screen so the human can tell us which one they want to test.

Then the first loop controls how many times we run the test. Basically the whole program other than setup is inside this loop.

The first chunk inside is another loop that picks 20,000 random numbers from 0 to 999 and stores copies into each array.

Then, depending on which option was chosen, we set an appropriate description for the tests, start a timer, run the specified sort, and then stop the timer.

After a single test runs (and assuming we're no longer in the warmup phase), we check to see if this run was faster than our previous winner or if it was slower than our previously slowest run.

This ends our main outer loop, so after all the tests are complete we just print out the details.

Below that are two functions (also known as "static methods"). They are nearly identical except that one has to use `.compareTo()`, and the temporary swapping variable is an `Integer`.

## On Efficiency

You can see that, all things being equal, Java is a little faster at dealing with native `ints` than `Integer` objects. On my computer, primitives are about "twice as fast". Keep that in the back of your mind when writing code that deals with a WHOLE lot of values.

However! Don't be fooled! The real takeaway is that even using Objects and a crappy quadratic-time sort that I wrote myself, my machine can sort 20,000 values in less a sixth of a second!

And! There's no reason to write my own sort because Java has a perfectly-good library sort that can sort the *same* values in under a hundredth of a second!

What is *really* more efficient is using a better algorithm and not having to write the code yourself!

The built-in sorting method provided by Java's standard library is an  $O(n \log n)$  (aka "log linear") sort. For primitive `ints`, the [documentation says](https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#sort-java.lang.Object:A-)<sup>11</sup> it uses a "stable, adaptive, iterative mergesort that

---

<sup>11</sup><https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#sort-java.lang.Object:A->

requires far fewer than  $n \lg(n)$  comparisons when the input array is partially sorted, while offering the performance of a traditional mergesort when the input array is randomly ordered.... [It] was adapted from Tim Peters's list sort for Python (TimSort)."

Do you know my favorite thing about using the standard library's sort? It is only one line of code, and I don't have to debug it! And it uses a more efficient algorithm than I would have anyway.

So here are my suggestions for writing efficient code:

## Mr. Mitchell's Tips for Writing Efficient Code

1. Don't worry about efficiency. You are probably wrong about what's efficient, anyway.
2. Write the code. Don't worry about whether or not it's efficient.
3. Get the code to work. (Debug it.)
4. Make sure the code does what it is supposed to. (Test it.)
5. Now that it works *correctly*, is the code fast enough? If so, you're done!
6. If it's too slow, "profile" the code to see which parts are slow.
7. Fix the slow parts. Go back to step 4.

It is far easier to make *correct* code faster than it is to get "optimized" but broken code working.

And to quote a famous computer scientist:

"Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%."

– Donald Knuth

This quote is from 1974, when popular computers like the PDP-11 had fewer than one megabyte of memory and ran at clock speeds slower than 15 MHz! If programs for machines back then didn't need obsessive optimization, then ours today don't either!



## Study Drills

1. Add a fourth menu option that uses `Arrays.sort()` to sort `intArr` (the array of primitive integers).
2. If you want, put the menu in a loop so that you can run different tests without having to start the program over. Make it so that the program keeps running until they choose menu option 5: quit. (Mr. Mitchell's real-life students don't need to do this Study Drill to receive credit.)

# Exercise 27: Static Variables and Static Methods

Okay, we've talked about efficiency and testing for a few exercises now, so let's get back to talking about more of the object-oriented language features of Java.

Up for this exercise: static member variables and static methods.

Are you going crazy with anticipation?!? You *finally* get to find out what `static` means, even though you have been typing it into every Java program since the very beginning!

We're back to two files again; the first file is the object, and the second file is the driver that uses our object.

Fish.java

---

```
1 public class Fish {
2     private String name;
3     private int id;
4     private static int count = 0;
5
6     public Fish( String name ) {
7         this.name = name;
8         count++;
9         id = count;
10    }
11
12    public String toString() {
13        return name + ", fish #" + id + " of " + count;
14    }
15
16    public static int numCreated() {
17        return count;
18    }
19 }
```

---

So pretend you are going to code a virtual fish tank. It will have fish swimming around in it, and each fish will have a name. (That's all for now; our fish are pretty boring.)



We want to be able to keep track of the different fish objects in some other part of our code, however, so each fish will also have an *id* that uniquely identifies it. Our goal is so that no matter how many fish objects we instantiate, they will all get a different id.

Notice that in this object, the private variable *count* is labeled as *static*, and so is the public method `numCreated()`. This is important.

FishDriver.java

---

```
1  import java.util.ArrayList;
2
3  public class FishDriver {
4      public static void main( String[] args ) {
5
6          String[] names = { "Wanda", "Dory", "Bruce", "Qwerty", "Blinky", "Lenny",
7                          "Flounder", "Mr. Limpet", "Mrs. Puff", "Moby Dick", "Freddi" };
8
9          ArrayList<Fish> school = new ArrayList<Fish>();
10
11         for ( String n : names ) {
12             Fish f = new Fish(n);
13             school.add(f);
14         }
15
16         for ( Fish f : school ) {
17             System.out.println(f);
18         }
19
20         System.out.println();
21         System.out.println(Fish.numCreated() + " Fish objects were created.");
22     }
23 }
```

---

The driver has an array of fish names so I don't have to write the same code over and over again. And there's an `ArrayList` that will hold references to `Fish` objects.

I start with a loop over each fish name, and for each name I instantiate a `Fish` object with that name and add that new `Fish` to the `ArrayList`.

Then I make another loop, having all the `Fish` "print" themselves using the `Fish` class' `toString()` method. You'll notice that each fish knows its *id* *and* how many total fish there are!

Then on line 21 of the driver, I'm calling the *static* method `numCreated()` to ask the entire `Fish` *class* how many `Fish` objects have been created so far.

## What You Should See

```
Wanda, fish #1 of 11
Dory, fish #2 of 11
Bruce, fish #3 of 11
Qwerty, fish #4 of 11
Blinky, fish #5 of 11
Lenny, fish #6 of 11
Flounder, fish #7 of 11
Mr. Limpet, fish #8 of 11
Mrs. Puff, fish #9 of 11
Moby Dick, fish #10 of 11
Freddi, fish #11 of 11

11 Fish objects were created.
```

Before I talk too much about the implications of `static`, let's remember how regular (non-static) instance variables and methods work.

In the `Fish` class, the `String name` and the integer `id` **belong** to the `Fish` object. Every time a `Fish` object is instantiated that object gets its own copies of both of those variables.

So if you have created a `Fish` object called *f*, then calling `f.toString()` operates on *f*'s copy of the *name* variable and on *f*'s copy of the *id* variable.

A `static` variable is different. It belongs to the class, but it belongs to the *whole* class. No matter how many instances of `Fish` objects get created, they all share a **single** copy of the variable *count*.

This is why there's a `count++` in the `Fish` class constructor. *count* is initialized to 0 all the way up on line 4, so whenever a new `Fish` is being created, that `Fish` adds to the shared counter and then sets its own *id* field to that same number.

So inside `toString()`, the values of *name* and *id* are unique to each `Fish` object, but the value of *count* is shared by them all. Any `Fish` object could change that variable and *all* other `Fish` objects would then see that new value.

The method `numCreated()` can be declared `static` because it *only* tries to access `static` variables. A `static` method can only access `static` variables. This is because `static` methods and `static` variables exist even when *no* objects from the class have been instantiated yet.

A regular (non-static) method can access either kind of variable, `static` or not.

Because `static` method `numCreated()` exists even when no `Fish` objects have been created, you can call it like we did on line 21 of the driver:

```
int fishCount = Fish.numCreated();
```

Notice that you just use the name of the class. It is also legal (but weird) to call a static method from an *instance* of a class:

```
Fish omari = new Fish("Sticks");  
int count = omari.numCreated();
```

That works just fine.

We didn't *have* to define the `numCreated()` method to be `static`. Just because it only accesses static variables doesn't mean that it *has* to be static. But if we hadn't made it static, then

```
int fc = Fish.numCreated();
```

...wouldn't work.

Okay, that's probably enough for this exercise. In the next one we will look at some popular static methods that are often used in Java.



## Study Drills

1. Make it so that `Fish` objects also have a `String` specifying their color ("red" or "blue"), and make it so that if the current `Fish` was red, then the next one will be blue. You may add a second static variable to help accomplish this, though it is possible to do it without one.

Make sure to add the color to the `toString()` method.

# Exercise 28: Popular Static Methods in Java

Now that you know about static methods and static variables, I'm going to point out several of them in Java. You have already used many of them in previous exercises!

PopularStaticMethods.java

```
1 public class PopularStaticMethods {
2     public static void main( String[] args ) {
3         System.out.println( "println() is a static method!" );
4
5         System.out.print("You don't need to instantiate a Math object ");
6         System.out.println("to call the random() method; it's static!");
7         double a = Math.random();
8         double b = Math.random();
9
10        // Math class
11        System.out.println( Math.PI );
12        System.out.println( Math.abs(-9) ); // absolute value
13        System.out.println( Math.sqrt(2) ); // square root
14        System.out.println( Math.min(a, b) + " is smaller." );
15        System.out.println( Math.max(a, b) + " is larger." );
16        System.out.println( Math.pow(a, b) ); // a raised to the power of b
17        double x = Math.PI * 0.5;
18        System.out.println( Math.sin(x) );
19        System.out.println( Math.cos(x) );
20        System.out.println( Math.tan(x) );
21
22        // String class
23        String digits = String.valueOf( Math.E ); // <-- valueOf() is static
24        int num = Math.min(17, digits.length());
25        String mitchell = digits.substring(0,num); // <-- substring() is not
26        System.out.println("Mr. Mitchell has " + mitchell + " memorized.");
27
28        // Integer class
29        System.out.println("ints take up " + Integer.SIZE + " bits.");
30        System.out.println("\tSmallest possible int: " + Integer.MIN_VALUE);
31        System.out.println("\tLargest possible int: " + Integer.MAX_VALUE);
```

```
32     int model = Integer.parseInt("1070");
33     model += 10;
34     System.out.println( String.valueOf(model) );
35     System.out.println( ""+model );
36 }
37 }
```

---

Java has a built-in class called `Math`. It's in the package `java.lang`, which means you don't have to import anything to use it; everything in `java.lang` is always imported.

Every method in the `Math` class is static, so there is never a need to instantiate a `Math` object.

The `Math` class also contains two *public* static variables, but both variables are also defined to be *final*, which means that their values can't be changed. (So don't go getting any ideas.)

I've always liked the way the Java documentation explains it:

```
public static final double PI
```

```
    The double value that is closer than any other to pi, the ratio of the circumference of a circle to its diameter.
```

"Closer than any other." Nice.

And you actually finally know enough to understand everything in that definition:

The variable is named *PI*. It is a `double` because it's a number and needs decimals. It's *final*, so it has a value already and can't be changed. The variable is *public* which means that it can be accessed directly from outside the `Math` class. And it is *static*, which means that there's only one copy that belongs to the whole class *and* you don't have to create an instance of a `Math` object to be able to get to it.

Whew.

This is why reading the Java documentation is really tough for beginners. There is a *ton* of information in just a single line, and every word means something important and a little complicated.

In fact, that's one of the things I don't like about Java. Doing *anything* in Java has a relatively high cognitive load.

## What You Should See



```

println() is a static method!
You don't need to instantiate a Math object to call the random() method; it's st\
atic!
3.141592653589793
9
1.4142135623730951
0.24228266829807188 is smaller.
0.5041921434592096 is larger.
0.4893055760546649
1.0
6.123233995736766E-17
1.633123935319537E16
Mr. Mitchell has 2.718281828459045 memorized.
ints take up 32 bits.
    Smallest possible int: -2147483648
    Largest possible int: 2147483647
1080
1080

```

(By the way, it's true that I have the first 17 digits of Euler's number memorized.)

The `String` class that you use every time you create a `String` variable also has a few static methods in it. `String.valueOf()` makes a `String` out of a numeric value. This is used for graphics applications, since there is usually only a method to draw a `String` onto a graphics window. So if you want to display a score or something, you've got to convert your integer into a `String` before you can display it.

But most of the methods inside the `String` class are *not* static. It doesn't make sense for `substring()` to be static, because it needs to have a value to make a substring out of!

The `Integer` class also has several useful static methods. I tend to use `Integer.MAX_VALUE` to give a starting value to my variable before I find the minimum value in a list. (You might remember I did that a couple of exercises back.)

`Integer.MAX_VALUE` is the largest value that can be stored into an `int` before overflow happens. One larger and it'll wrap around.

Another really useful one is `Integer.parseInt()`, which is like the inverse of `String.valueOf()`. If you give it a `String` that contains only digits, it will return an `int` with the same value.

Again, graphical programs tend to *only* be able to receive input as a `String`. If you ask the person to enter their age and then they type it in, it'll come in as a `String` and you have to convert it before you can do arithmetic on it.

Before the exercise ends, though, I should confess that I pretty much *never* use `String.valueOf()` to convert a number into a `String`. I always just add it to the empty `String` as shown on the last line

of the program.

It works just as well and I don't have to remember what it's called! Plus it works the same in just about every programming language.

It's been a while since I suggested that you look at the official Java documentation, but now you've learned a lot more and should be able to make sense out of a lot of it.

- [java.lang.Math](#)<sup>12</sup>
- [java.lang.String](#)<sup>13</sup>

Scroll past the introductory text down to the “Method Summary”, and look at the methods.

Notice what type of value each method returns, and whether or not the method is static.

Notice if the method needs any parameters, and what types those parameters need to be.

I think you're starting to get to the point where you are capable of understanding *how* to call 80% of the methods in Java classes that you could import! That's pretty cool.

Just take it slow and look at one part at a time.

Okay, see you in the next exercise, where we'll talk about something cool you can do with static members of a class.



## Study Drills

1. Add some code to call one of the static methods of either the Math class or the String class that isn't already in this exercise.
2. Find the javadocs for the Integer class. (I usually just search the phrase “java.lang.Integer” and then click on the link from Oracle.) Figure out how to use one of the static methods from the Integer class to convert a binary number in a String (like “101010”) into a regular base-10 number. Hint: another word for “base” is “radix”.

---

<sup>12</sup><https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html>

<sup>13</sup><https://docs.oracle.com/javase/8/docs/api/java/lang/String.html>

# Exercise 29: Importing Static Class Members

This is a pretty short exercise just to show you something neat that Java lets you do with static items in a class.

ImportingStatic.java

---

```
1  import static java.lang.Math.*;
2  import static java.lang.System.out;
3
4  public class ImportingStatic {
5      public static void main( String[] args ) {
6          double a = random();
7          double b = random();
8
9          out.println( PI );
10         out.println( abs(-9) );
11         out.println( sqrt(2) );
12         out.println( min(a, b) + " is smaller." );
13         out.println( max(a, b) + " is larger." );
14         out.println( pow(a, b) ); // a raised to the power of b
15         out.println( Math.sin(PI) );
16         out.println( Math.cos(PI) );
17         out.println( Math.tan(PI) );
18     }
19 }
```

---

In some programming languages, you couldn't name something *sqrt*. It's because there's already a built-in square root function and it is always around.

You'd say that the identifier for *sqrt* is in the default global "namespace". If you remember symbol tables from a few exercises ago, the namespace is basically the list of variables that are already in the symbol table before you even write a single line of your own code.

In other languages, once you import the Math library, **then** *sqrt* is in the namespace. But if you didn't import the Math library you could use that name in your program. (Not sure why you would want to, but....)



Java has a compromise approach that I like. The `Math` library is always “imported”, but the function names don’t clutter up the global namespace. I don’t have to worry whether or not I can make a variable called *min* because even though the `Math` library is available all those static methods are safely inside. I have to say `Math.min()` if that’s what I want.

But what if I’m creating a real math-heavy program. Must I really do

```
a = Math.sqrt( Math.pow(b, 2) + Math.pow(c, 2) - 2*b*c * Math.cos(A) );
```

Goodness, no! Java allows you to write a line that means “import all the static methods and variables into the current namespace.”

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

(When dealing with computers, `*` is often a “wildcard” character that means “match anything”, and that how it is used here.)

After importing those static items, you can now get away with writing:

```
a = sqrt( pow(b, 2) + pow(c, 2) - 2*b*c * cos(A) );
```

Which is nicer!

## What You Should See

```
3.141592653589793
9
1.4142135623730951
0.42034187251898336 is smaller.
0.6354469426811356 is larger.
0.5765267432127924
1.2246467991473532E-16
-1.0
-1.2246467991473532E-16
```

See, it really works!

The `System` class contains a static member called “out”, which is a `PrintStream` object featuring our oft-used `println` method!

So you can import *just* that single object as shown, allowing us to write

```
out.println( "No System needed!" );
```

It is considered bad style to import a lot of static items into the namespace, so use it sparingly.

And one more thing before the Study Drills. Did you notice the unusual output at the end of the program? Maybe you didn't notice, but *I* expected both the sin and tan of  $\pi$  to be 0, not  $1.2246 \times 10^{-16}$ !

This is a common problem when working with floating-point numbers on a computer. It's not just Java; even the professional numerical computing environment MATLAB gives  $1.2246\text{E-}16$  as the answer! I don't know how well you remember high-school math, but  $1.22\text{E-}16$  is a very small number close to zero:

```
0.00000000000000001224646799
```

But because this sort of thing is really common, you shouldn't test for equality when using floating point numbers because roundoff errors will make them slightly different.

That is, don't write:

```
double a = 0;
double b = sin(PI);
if ( a == b ) {
    // do something
}
```

...because the `if` statement will almost never be true. Instead, check if the two values are "close enough":

```
double a = 0;
double b = sin(PI);
double error = 1.0E14;
if ( abs(a-b) < error ) {
    // close enough!
}
```

This is much safer.

Okay, that's enough for this exercise. We take a look at parameter passing in the next exercise!



## Study Drills

1. Import one of the static variables from one of the wrapper classes like `Integer`, `Long`, or `Double` and add code to display its value on the screen.

# Exercise 30: References as Parameters

In this exercise, we're going to review references and look at the implications when you pass a reference variable as a parameter to a method.

Before typing in the code, remind yourself that array variables contain a reference to a memory location in the heap. And remember that assigning one array to another copies the *reference*, not the values inside the array.

ReferenceParameters.java

---

```
1  import java.util.Arrays;
2
3  public class ReferenceParameters {
4      public static void main( String[] args ) {
5          int[] a = new int[7];
6          Arrays.fill(a, 2);
7          System.out.println( Arrays.toString(a) );
8
9          int[] b = { 1, 2, 3, 4, 5 };
10         a = b;
11         System.out.println( Arrays.toString(a) );
12
13         int[] c = new int[a.length*2];
14         for ( int i=0; i<a.length; i++ ) {
15             c[i] = a[i];
16         }
17         a = c;
18         System.out.println( Arrays.toString(a) );
19
20         System.out.println("\na is at reference " + a);
21         System.out.println( Arrays.toString(a) );
22         squareValues(a);
23         System.out.println( Arrays.toString(a) );
24
25         System.out.println("\na is at reference " + a);
26         chopArray(a);
27         System.out.println( Arrays.toString(a) );
```

```

28         System.out.println("a is at reference " + a);
29
30         System.out.println("\na is at reference " + a);
31         a = chopArray2(a);
32         System.out.println( Arrays.toString(a) );
33         System.out.println("a is at reference " + a);
34     }
35
36     public static void squareValues( int[] arr ) {
37         System.out.println("Squaring array at reference " + arr);
38         for ( int i=0; i<arr.length; i++ ) {
39             arr[i] = arr[i]*arr[i];
40         }
41     }
42
43     public static void chopArray( int[] arr ) {
44         int[] c = new int[arr.length/2];
45         for ( int i=0; i<c.length; i++ ) {
46             c[i] = arr[i];
47         }
48         arr = c;
49         System.out.println("The array is now at reference " + arr);
50     }
51
52     public static int[] chopArray2( int[] arr ) {
53         int[] c = new int[arr.length/2];
54         for ( int i=0; i<c.length; i++ ) {
55             c[i] = arr[i];
56         }
57         System.out.println("Returning the reference " + c);
58         return c;
59     }
60 }

```

---

So *a* is an array with 7 slots (numbered 0 through 6). Just for fun, we use the static method `Arrays.fill()` to put a 2 into all the slots. Then we use another static method – `Arrays.toString()` – to give us a nice String representation of the array’s contents that is suitable for displaying.

Moving on, *b* is also an array. We used an initializer list, so it has values already, and the `.length` of *b* is 5 (indexed 0 through 4).

Then, on line 10, we overwrite *a* with the *reference* from *b*. The array that had only 2s in it is gone; both *a* and *b* reference the same five-element array on the heap. Displaying *a* confirms this.

In the next little chunk of code we create a third array. *c* is twice as long as *a* (so, 10 slots numbered 0-9). We use a traditional for loop to copy all of *a*'s values into *c*, so that the first five slots in *c* have the numbers 1-5 in them. (The rest are still zeroes, since that what arrays in Java are filled with by default.)

Then in line 17 we overwrite *a* again, this time with a copy of *c*'s reference.

This is kind of cool, since we just sort-of made *a* **larger**. It used to have 5 slots, with 1-5 in them, and now it has the same contents but more capacity! The reference is different, but who cares?

Well... we have to. As you'll soon see.

Starting on line 20 we print out the reference so we can keep track of it. Remember that though arrays in Java are objects (and Objects), there's nothing good in their `toString()` method, so attempting to print an array just shows funny symbols followed by a memory reference. (First there's a `[` to indicate that it's an array, then the capital `I` means it's an array of primitive ints, then an `@` sign and finally the reference / memory address in hexadecimal (a.k.a. base 16).)

## What You Should See

```
[2, 2, 2, 2, 2, 2, 2]
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5, 0, 0, 0, 0, 0]

a is at reference [I@6bc7c054
[1, 2, 3, 4, 5, 0, 0, 0, 0, 0]
Squaring array at reference [I@6bc7c054
[1, 4, 9, 16, 25, 0, 0, 0, 0, 0]

a is at reference [I@6bc7c054
The array is now at reference [I@232204a1
[1, 4, 9, 16, 25, 0, 0, 0, 0, 0]
a is at reference [I@6bc7c054

a is at reference [I@6bc7c054
Returning the reference [I@4aa298b7
[1, 4, 9, 16, 25]
a is at reference [I@4aa298b7
```

Keep an eye on the memory reference value. We print out *a*'s current address on line 20, then its contents on the following line. Then we pass a copy of *a*'s reference into the static method `squareValues()`.

The `squareValues()` method has a parameter named *arr*. You can see that *arr* has the same reference in it that *a* does. They are different variables, but they both hold a copy of the same address. Thus they “refer” to the same physical array in the heap.

The method goes through each slot in the array, changing each value. And so when the method ends and gives control back to `main()`, you can see that the values in *a* are also changed. (Hopefully unsurprisingly, since it’s the same array.)

Okay, second example. Display *a*’s reference, pass a copy of that reference into the static method `chopArray()`.

So inside `chopArray()`, it also has a parameter called *arr*. This is technically a different variable than the one from the previous method, but it also has a copy of the reference value from *a*.

The first thing `chopArray()` does is create a new array called *c* with half the capacity of *arr* (so, `c.length` is 5). Just like we did earlier, this method copies all the values over and **overwrites** *arr* with a copy of the reference to this new, smaller array.

So *arr* no longer refers to the same physical array in the heap as *a* does. We can confirm this by displaying the reference that *arr* now holds; you can see that it’s different.

Then the method ends and returns control back to `main()`. The \$64,000 question is this: what happens to *a*?

Hopefully the answer is not a surprise. Nothing. Nothing happens to *a*.

It’s true that *arr* had a **copy** of the reference from *a*, but destroying that copy doesn’t have anything to do with the original value. It’s still safely stored in *a*.

Some programming languages have a concept of passing in parameters that can be modified by the function or method. In these languages (like C++), you can specify whether a parameter can be changed or not.

Java doesn’t have any of this. **All** parameters in Java are pass-by-copy. If the parameter is an object then it’s making a copy of the reference. If the parameter is a primitive type then it makes a copy of the value. But it is always a copy. There’s no way around this in Java, which is sometimes inconvenient but makes Java programs easier to debug and safer to write most of the time.

Okay, final example. We print out *a*’s reference (in case you somehow forgot what it was) and then pass a copy of it into the static method `chopArray2()`.

The `chopArray2()` method is identical to its similarly-named sibling, but with a tiny difference: it doesn’t bother trying to modify *arr* (since it wouldn’t do any good, anyway). Instead it **returns** the reference to the new, smaller array.

You can see that the method isn’t `void` like the previous one; its return type is `int[]`, which means that it returns a reference to an array of integers.

So where does that returned reference go? Back in `main()`, you can see what we did:

```
a = chopArray2(a);
```

We stored the return value of the method into *a*, overwriting the reference that was there. There's no other way to do this in Java. It's impossible to write a method in Java that can change the value of a parameter.

Hopefully you can see how this is different than what happened in the first example. `squareArray()` didn't try to change the *value* of the parameter *arr*. The reference in that variable was never touched. It only changed the values in the physical array that the parameter referred to.

This is a subtle distinction, but important.

Anyway, that's enough for this exercise. Next exercise we'll talk about what impact this all has on String objects.



## Study Drills

1. Create a new method called `smallerIncrementedArray()`. It should create a new array with ONE space less than the array passed in. (So if they send in an array with a capacity of FIVE it should create a new array with a capacity of FOUR.) Then fill the array with values one larger than the original array. (So if the original array is { 1, 4, 9, 25, 36 } then the new array should be { 2, 5, 10, 26 }.) Then return the reference to the new array from the method, so that the original array can use it.

# Exercise 31: Java Strings Are Immutable

By this point you understand that primitive variables just hold a value and object variables hold a *reference* to the object instead. And you understand that you can change the reference to change which object a particular variable... refers to. Um.

But without changing the reference, is it possible for the internal details of an object to change? (We call these internal details the object's "state".)

In Java, the answer is *sometimes*. Some objects have "mutable" state – the internals of the object can be *mutated* or changed. Other objects are *immutable*; once an object is created it cannot be modified in any way.

And in Java strings are immutable. Okay, let's look at some code.

ImmutableStrings.java

---

```
1  import java.util.ArrayList;
2
3  public class ImmutableStrings {
4      public static void main( String[] args ) {
5          ArrayList<Double> list = new ArrayList<Double>();
6          System.out.println(list);
7          list.add(4.0);
8          list.add(2.0);
9          System.out.println(list);
10         list.clear();
11         System.out.println(list);
12
13         String word = "impossible";
14         System.out.println("\n" + word);
15         word.substring(2);
16         System.out.println(word);
17         word.toUpperCase();
18         System.out.println(word);
19
20         String w = "workaround";
21         System.out.println("\n" + w);
22         w = w.substring(0,4);
```



```
23     System.out.println(w);
24     w = w.toUpperCase();
25     System.out.println(w);
26
27     System.out.println("\nBefore passing to method: " + word + ", " + w);
28     swap(word, w);
29     System.out.println("After returning from method: " + word + ", " + w);
30 }
31
32 public static void swap( String a, String b ) {
33     System.out.println("\tBefore \"swapping\": " + a + ", " + b);
34     String temp = a;
35     a = b;
36     b = temp;
37     System.out.println("\tAfter \"swapping\": " + a + ", " + b);
38 }
39 }
```

---

So we start here with an `ArrayList` because `ArrayLists` *are* mutable. See how easy it is to change? It has a method called `add()`, which changes the object by adding a item. The `size()` also changes when you add an item. Also there is a method called `clear()`, which resets the list back to its empty state.

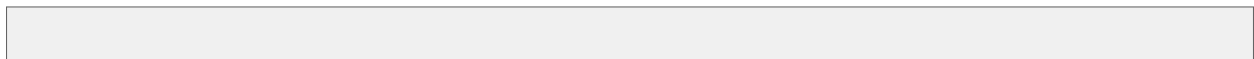
`ArrayList` methods like `size()`, `get()`, `isEmpty()` and `indexOf()` are called *accessor* methods because they report on the properties of an object but don't change anything.

`ArrayList` methods like `add()`, `clear()` and `set()` are *mutator* methods because they change the state of the list.

(In some programming languages, accessor and mutator methods are called “getters” and “setters” instead.)

In Java, Strings are immutable because the `String` class *doesn't have any mutator methods*. The `String` class only has accessor methods.

## What You Should See



```
[]  
[4.0, 2.0]  
[]  
  
impossible  
impossible  
impossible  
  
workaround  
work  
WORK  
  
Before passing to method: impossible, WORK  
    Before "swapping": impossible, WORK  
    After "swapping": WORK, impossible  
After returning from method: impossible, WORK
```

Let us look at a specific example. `substring()` is a `String` method that *reports* by returning a shorter version of the `String` object.

If the variable *word* contains "impossible", then `word.substring(2)` **returns** the `String` from character position 2 to the end ("possible"). But the `String` object that *word* refers to isn't affected.

And in the code, we called the `substring()` method, it *returned* a shorter `String` and we didn't do anything with what it gave us.

The method `toUpperCase()` is similar; it returns an upper-case version of *word*, which we threw away.

So the object that *word* refers to can't be changed; it is immutable. But we can change *which* object that variable refers to!

The `String` variable *w* contains a reference to "workaround". If we write

```
w = w.substring(0,4);
```

...then *w* is changed to refer to the new, shorter `String` object created by the method call.

The final chunk of code shouldn't surprise you. Based on what you learned in the previous exercise, you know that if you pass an object into a method as a parameter, the method can't change any references.

A method *can* ask an object to mutate itself, but since the `String` class doesn't have any mutator methods, there's pretty much no way for a method to change a `String` passed into it.



## Study Drills

1. Create a new method called `replaceFirst()`. Pass the `ArrayList` into it as a parameter and have the method `set()` the first value to `0`. Don't return anything. Then add some code into `main()` to confirm that since `ArrayLists` are mutable, your new method can actually change its state.

# Exercise 32: Parameters vs Properties

Okay, since the last two exercises were about how a method cannot change one of its parameters unless the parameter is a mutable object, let us remember what methods *can* do.

Java is an object-oriented language, so *most* of the methods you write will belong to objects. And while methods inside an object can't change the parameters passed into them, they *can* change any instance variables / fields of their own object.

Here's a silly example. I have written constructors, a getter method and a toString() method, but since they aren't the focus of this exercise I put each one all on a single line.

NumberGuy.java

---

```
1 public class NumberGuy {
2     private int number;
3
4     public NumberGuy() { number = 0; }
5     public NumberGuy( int number ) { this.number = number; }
6     public int getNumber() { return number; }
7     public String toString() { return "" + number; }
8
9     public void doubleYourself() {
10         number = number*2;
11     }
12
13     public void changeThis( int q ) {
14         q = q*2;
15     }
16 }
```

---

So the NumberGuy object has one field (instance variable): the private primitive int called *number*.

You can see that the doubleYourself() method changes this field to be twice what it used to be.

Also, there's a method called changeThis() It receives one parameter, and attempts to change the value of that parameter. (You should be suspicious that this will have no lasting effect.)

**NumberGuyDriver.java**

```
1 public class NumberGuyDriver {  
2     public static void main( String[] args ) {  
3         NumberGuy n = new NumberGuy(5);  
4  
5         System.out.println(n);  
6         n.doubleYourself();  
7         System.out.println(n);  
8  
9         int q = 4;  
10        System.out.println(q);  
11        n.changeThis(q);  
12        System.out.println(q);  
13    }  
14 }
```

In the driver we instantiate a `NumberGuy` object with a value of 5 stored in it. We display it (using the handy `toString()` method to help us). Then we ask the object to double itself using its mutator method.

It works!

Then we pass a variable containing 4 into `NumberGuy`'s `changeThis()` method. But the method isn't successful.

## What You Should See

```
5  
10  
4  
4
```

Java works this way on purpose. Java is a language of objects, and you may remember that one of the big ideas behind object-oriented programming is “encapsulation”: variables inside an object can only be changed by that object. Objects can change themselves if they want, but they shouldn't change what you send them.

This makes large programs safer and easier to reason about. You can know that after you pass a variable into a method the method can't accidentally (or on purpose!) change the value of that variable.

Make sense?

In the next exercise we will look at a different aspect of object-oriented programming called “inheritance”.



## Study Drills

1. Add a new method to the `NumberGuy` class called `doubleMe()`. You should pass in a parameter and have the method double *and return* the doubled value. Then call the new method from the driver in a way that allows *q* to be changed.

# Exercise 33: Basic Inheritance

Object-oriented programming has three main characteristics:

- Polymorphism
- Inheritance
- Encapsulation

For some reason I've always remembered these using the mnemonic phrase "PIE". There are three pieces to the object-oriented pie!

(And sometimes people throw in *Abstraction*, but encapsulation is just a more specific form of abstraction. Plus it doesn't fit in the pie.)

*Encapsulation* is combining data and code into a package / object so that the data can only be accessed or modified using the methods provided. You've seen a lot of encapsulation so far in this book.

*Polymorphism* is when you can write code once and have it work for different types of data. One way Java handles polymorphism is through generics, which you have also seen.

*Inheritance* is when a new object inherits some of the code from another object so you have less code to write overall. That's what we are going to learn in this exercise!

Keep an eye out for the keyword `extends`, which will appear in the `Square` class. That word means inheritance is happening.

Rect.java

---

```
1 public class Rect {
2     private double length, width;
3
4     public Rect() {
5         length = width = 0;
6     }
7
8     public Rect( double length, double width ) {
9         this.length = length;
10        this.width = width;
11    }
12
13    public double getLength() { return length; }
```

```
14     public double getWidth() { return width; }
15
16     public double getArea() {
17         return length*width;
18     }
19
20     public double getPerimeter() {
21         return 2*length + 2*width;
22     }
23
24     public String toString() {
25         return "Rect, " + length + "x" + width;
26     }
27 }
```

---

So this is a basic rectangle class. It's nearly identical, in fact, to an object you wrote in a previous exercise. It's important to note that it contains two private fields: *length* and *width*, and it also has a couple of constructors, and some accessor methods including `getLength()` and `getArea()`.

This rectangle class doesn't have any mutator methods (or "setters", as they are sometimes called), so `Rect` objects are immutable. That has nothing to do with inheritance, but I thought I would mention it anyway.

Square.java

---

```
1  public class Square extends Rect {
2      public Square() {
3          super();
4      }
5
6      public Square( double side ) {
7          super(side, side);
8      }
9
10     public String toString() {
11         return "Square, " + super.getLength() + "x" + super.getLength();
12     }
13 }
```

---

Notice that first line:



```
public class Square extends Rect
```

The keyword `extends` means that `Square` *inherits* from `Rect`. That means that a `Square` object *is* a `Rect` object, and it automatically gets copies of `Rect`'s fields and methods. It does not inherit any constructors, though; we'll have to rewrite those.

In this relationship, `Square` is called the “subclass” of `Rect`, and `Rect` is the “superclass” of `Square`. So when you see the keyword `super` being used, it means “go into the superclass, and use that thing.” (`super` is very similar to `this`, except it refers to the superclass instead of the current class.)

`super()` means “run the constructor of the superclass”. (Note the parens.) It is *only* legal as the first line in the subclass constructor.

So inside the default constructor for the `Square` class, the line `super()`; calls the default constructor of the `Rect` class, which sets both *length* and *width* to 0.

And inside `Square`'s parameter constructor, the line `super(side, side)`; calls `Rect`'s parameter constructor, passing in a copy of *side*'s value as the length and a second copy as the width.

Now, the `Square` class already has a copy of `Rect`'s `toString()` method. It was inherited. But we define a new version of `toString()` which overrides the inherited version.

Inside `Square`'s `toString()`, we use the keyword `super` without parens, so it doesn't call any constructor. It just refers to something in the superclass. So the expression `super.getLength()` means “call the superclass' version of `getLength()`”

In this case, we didn't have to use `super` because there's only one version of that method. So we could have omitted it. (In fact, you'll do that in one of the Study Drills.)

Okay, now let's see this inheritance in action.

BasicInheritance.java

---

```

1 public class BasicInheritance {
2     public static void main( String[] args ) {
3         Rect rect1 = new Rect(4,5);
4         System.out.println(rect1);
5         System.out.println("\tArea: " + rect1.getArea());
6         System.out.println("\tPerimeter: " + rect1.getPerimeter());
7
8         Square sq1 = new Square(3);
9         System.out.println("\n"+sq1);
10        System.out.println("\tArea: " + sq1.getArea());
11        System.out.println("\tPerimeter: " + sq1.getPerimeter());
12
13        Rect r = new Rect();
14        Rect s = new Square(2);
15        System.out.println("\n" + r + ", and " + s);

```

```
16
17     r = s;
18     System.out.println("\nHere is my rectangle: " + r);
19 }
20 }
```

---

It will look like this when you run it:

## What You Should See

```
Rect, 4.0x5.0
    Area: 20.0
    Perimeter: 18.0

Square, 3.0x3.0
    Area: 9.0
    Perimeter: 12.0

Rect, 0.0x0.0, and Square, 2.0x2.0

Here is my rectangle: Square, 2.0x2.0
```

The first thing we do in the driver is instantiate a `Rect` object. We print it and call some methods on it.

Then we instantiate a `Square` object. You will notice that we can call the `getArea()` method on behalf of a `Square` object, even though those methods are only defined in the superclass! Inheritance at work!

This is because every `Square` object is *also* a `Rect` object, just with some extra stuff added.

Then we instantiate two more objects and display them. This proves that when the `toString()` method is called, it uses the correct version for the type of object it is.

On the last two lines of the driver I did something unusual. The variable `r` is capable of holding a reference to an object of type `Rect`. The variable `s` currently holds a reference to an object of type `Square`. So,

```
r = s;
```

...copies the reference from `s` into `r`. Is this okay? Sure, because `r` is capable of holding a reference to a `Rect`, and `s` holds a reference to a `Square`, and *every Square is a Rect*. An object of type `Square` is *also* an object of type `Rect`, just with some extra stuff added.

You can see that Java is even smart enough to call the appropriate `toString()` method. Even though *r* refers to rectangles, it calls the square version of the `toString()` method because the object at the other end of the reference is really a square.

Pretty neat, huh? So in the next exercise we will look a little more deeply at how inheritance works with fields.



## Study Drills

1. In the `Square`'s `toString()` method, remove one of the occurrences of the keyword `super` entirely, and replace the other occurrence with the keyword `this`. Confirm that everything still works.
2. In the driver, add a line of code at the bottom that tries to store *rect1* into the variable *s*. Do you expect it to work? Does it? Answer in a comment.



## Common Student Questions

In the `Square` class, I'm getting some error about overriding `toString`. Also, I am using Eclipse.

Ah. Well, you're not supposed to be using Eclipse. In the `Square` class, we added a `toString()` method even though one had already been inherited. The regular Java compiler doesn't care about this, but Eclipse doesn't want you to do this *accidentally*, so it complains. (In fact, Eclipse complains about a lot of things that aren't a problem for beginners, which is why I don't recommend it.)

Anyway, add the `@Override` annotation just before line 10 and Eclipse will quit bothering you. You'll have to do this in future exercises, too, if you insist on using Eclipse.

# Exercise 34: How Fields Are Inherited

In the previous chapter I simplified things a bit too much. I said that a subclass “automatically gets copies of [the superclass’s] fields.” This is true! But the subclass may not be able to *access* those inherited fields!

Probably not, actually. And there’s a new keyword in this exercise, too: `protected`.

FieldInheritance.java

---

```
1 public class FieldInheritance {
2
3     private String first;
4     protected String last;
5
6     public FieldInheritance() {
7         first = last = "";
8     }
9
10    public FieldInheritance( String first, String last ) {
11        this.first = first;
12        this.last = last;
13    }
14
15    public void setFirst( String s ) {
16        first = s;
17    }
18
19    public void setLast( String s ) {
20        last = s;
21    }
22
23    public String getFirst() { return first; }
24    public String getLast() { return last; }
25
26    public String toString() {
27        return first + " " + last;
28    }
29 }
```

---

So this object has two fields: *first* and *last*. The String *first* is private, which is what you usually want.

The String *last* is “protected”, which is similar to private. There’s one tiny difference, though. We’ll see that shortly.

There is nothing else interesting in this file, just the usual constructors and getter and setters you usually have to write when defining a class in Java.

#### FieldInheritanceSub.java

---

```
1 public class FieldInheritanceSub extends FieldInheritance {
2
3     public FieldInheritanceSub() {
4         super();
5     }
6
7     public FieldInheritanceSub( String first, String last ) {
8         super(first, last);
9     }
10
11    public void test() {
12        // first = "GOOD";
13        last = "COFFEE";
14    }
15 }
```

---

This subclass extends the previous class, which means that it inherits copies of all the fields and methods (but not constructors). So FieldInheritanceSub *does* have a copy of both *first* and *last*.

But check out that test() method. Notice how one of the lines of code is commented out? That’s on purpose.

The subclass *has* a copy of the field *first*. But it can’t access it directly. Why? Because the field is private. Private fields can *only* be accessed within the same class. Period.

No other classes can touch private data members, *including* subclasses.

The field *last* **can** be accessed, however, because it is protected. When a field has protected visibility, it means that it is private *except* for subclasses. Subclasses can access protected fields; no other classes can.

**FieldInheritanceDriver.java**

```
1 public class FieldInheritanceDriver {  
2     public static void main( String[] args ) {  
3         FieldInheritance f = new FieldInheritance("Bill", "Brasky");  
4         System.out.println(f);  
5  
6         FieldInheritanceSub g = new FieldInheritanceSub("Turkish", "Delight");  
7         System.out.println(g);  
8         g.test();  
9         System.out.println(g);  
10    }  
11 }
```

The driver just confirms that everything works as expected. The subclass' `test()` method is able to directly change the superclass' protected field, but not the private one.

The subclass *can* still change that field, but it has to use the public getters and setters just like anyone else.

## What You Should See

```
Bill Brasky  
Turkish Delight  
Turkish COFFEE
```

If you are writing a class that you think other classes might inherit from, then it is probably a good idea to make your fields protected rather than private. And if you have a field in your class that you *know* should be off-limits to EVERY other class (including subclasses), then make it private!

It's nice to have that option. Java does a good job giving programmers very fine-grained access to data, which can make large Java programs safer.



## Study Drills

1. Uncomment the attempt to access *first* in the subclass. Does it compile? What error message do you get? Answer in a comment.
2. Add code in the `test()` method that uses the superclass setter method to change the value of *first*.

# Exercise 35: “Useful” Inheritance - A Game Board

We learned about inheritance in the last couple of exercises, but the programs we wrote weren't very useful, and it maybe wasn't obvious why you would want to use inheritance in them.

So for the next few exercises, we'll work on building a framework for creating games involving a two-dimensional grid.

This exercise is longer than most, because I need to have quite a bit of code for the inheritance to make sense. (This is true about real-world code, too.)

GamePiece.java

---

```
1 public class GamePiece {
2     protected int r, c;
3     protected GameBoard board;
4     protected String symbol;
5
6     public GamePiece() {
7         r = c = 0;
8         board = null;
9         symbol = "@";
10    }
11
12    public GamePiece(int r, int c, String symbol) {
13        this.r = r;
14        this.c = c;
15        this.board = null;
16        this.symbol = symbol;
17    }
18
19    public boolean addSelfToBoard( GameBoard board ) {
20        if ( board.canAdd(r,c) && ! isOnBoard() && board.get(r,c) == null ) {
21            this.board = board;
22            board.add(this,r,c);
23            return true;
24        }
25        return false;
26    }
```

```

27
28     public boolean isOnBoard() {
29         return board != null && board.get(r,c) == this;
30     }
31
32     public void removeSelfFromBoard() {
33         if ( isOnBoard() ) {
34             board.remove(r,c);
35             board = null;
36         }
37     }
38
39     public int getRow() { return r; }
40     public int getCol() { return c; }
41     public String getSymbol() { return symbol; }
42
43     public String toString() {
44         return "GamePiece at (" + r + ", " + c + ")";
45     }
46 }

```

---

This is the so-called “base class” for a piece that will be placed on a game board. All the fields are protected because we are expecting other classes to inherit from this one!

Each piece has a reference to the board it is placed on, and keeps track of its row and column within the grid. And since we are just drawing on a terminal screen, each piece stores a symbol.

There are some methods for the piece to add itself to a board and remove itself from a board. This code will make more sense after we use it later.

GameBoard.java

---

```

1  import java.util.ArrayList;
2
3  public class GameBoard {
4      protected GamePiece[][] g;
5
6      public GameBoard( int rows, int columns ) {
7          g = new GamePiece[rows][columns];
8      }
9
10     public void add( GamePiece a, int r, int c ) {
11         if ( ! canAdd(r,c) )
12             return;

```



```

13         g[r][c] = a;
14     }
15
16     public boolean canAdd( int r, int c ) {
17         return ( 0 <= r && r < g.length && 0 <= c && c < g[0].length );
18     }
19
20     public GamePiece get( int r, int c ) {
21         return g[r][c];
22     }
23
24     public void remove( int r, int c ) {
25         g[r][c] = null;
26     }
27
28     public String toString() {
29         String out = "";
30         for ( int r=0; r<g.length; r++ ) {
31             for ( int c=0; c<g[0].length; c++ ) {
32                 if ( g[r][c] == null )
33                     out += ".";
34                 else
35                     out += g[r][c].getSymbol();
36             }
37             out += "\n";
38         }
39         return out;
40     }
41 }
42
43 }

```

---

This is the game board! It has a two-dimensional array of GamePiece objects. (Well, technically it has a *reference* to an array of *references* to GamePiece objects.)

It has methods to add pieces to the board, remove them, etc. Notice that if they want to overwrite one piece with another, we allow them; this is fine in some games!

We only really check to make sure they don't blow up anything by putting the piece out of bounds.

## GameTestTicTac.java

---

```

1  import java.util.Scanner;
2
3  public class GameTestTicTac {
4      public static void main( String[] args ) {
5          GameBoard board = new GameBoard(3,3);
6          Scanner kb = new Scanner(System.in);
7          String p = "X";
8          int r, c, turns = 0;
9          boolean done = false;
10
11         do {
12             System.out.println("\n"+board);
13             System.out.print("'" + p + "', your turn: ");
14             r = kb.nextInt();
15             c = kb.nextInt();
16             if ( board.canAdd(r,c) && board.get(r,c) == null ) {
17                 GamePiece XO = new GamePiece(r, c, p);
18                 XO.addSelfToBoard(board);
19                 turns++;
20                 p = p.equals("X") ? "O" : "X";
21                 done = isWinner(board,"X") || isWinner(board,"O") || turns == 9;
22             }
23             else {
24                 System.out.println("\nSorry, you can't go there. Try again.");
25             }
26         } while ( !done );
27
28         System.out.println("\n"+board);
29
30         if ( isWinner(board, "X") )
31             System.out.println("X wins!");
32         else if ( isWinner(board, "O") )
33             System.out.println("O wins!");
34         else
35             System.out.println("It's a tie!");
36     }
37
38     public static boolean isWinner( GameBoard board, String p ) {
39         if ( winCheck(board, p, 0,0, 0,1, 0,2) ) return true;
40         if ( winCheck(board, p, 1,0, 1,1, 1,2) ) return true;
41         if ( winCheck(board, p, 2,0, 2,1, 2,2) ) return true;

```

```

42         if ( winCheck(board, p, 0,0, 1,0, 2,0) ) return true;
43         if ( winCheck(board, p, 0,1, 1,1, 2,1) ) return true;
44         if ( winCheck(board, p, 0,2, 1,2, 2,2) ) return true;
45         if ( winCheck(board, p, 0,0, 1,1, 2,2) ) return true;
46         if ( winCheck(board, p, 2,0, 1,1, 0,2) ) return true;
47         return false;
48     }
49
50     public static boolean winCheck(GameBoard board, String p, int a,int b,
51         int c, int d, int e, int f) {
52         if ( board.get(a,b) == null || board.get(c,d) == null
53             || board.get(e,f) == null )
54             return false;
55         String A = board.get(a,b).getSymbol();
56         String B = board.get(c,d).getSymbol();
57         String C = board.get(e,f).getSymbol();
58         return ( p.equals(A) && A.equals(B) && B.equals(C) );
59     }
60 }

```

---

Even though the GamePiece and GameBoard are supposed to be *base* classes, we can still make a game with them! So here is a simple tic-tac-toe game.

Maybe you think I’m really in love with tic-tac-toe games or something! No, it’s just that the code is small enough to do easily. Also, since we already did a tic-tac-toe game you can focus on how the *objects* are being used instead of the rules.)

So, at the top we create a GameBoard object with three rows and three columns.

After the human tells us which row and column they want, we ask the GameBoard if the location is *legal*. Then we check if it’s empty; in tic-tac-toe you can’t play twice in the same spot!

Then we create a new GamePiece object at that row and column and with the letter “X” as its symbol. We ask that new piece to add itself to our board.

Okay, so line 20 needs some explaining. It has nothing to do with inheritance or even object-oriented programming; I am using something generally called the “ternary operator”. (It is technically called the *conditional* operator or “inline if”, but in real life I’ve never heard it called anything but “ternary”.)

It is an expression with the following form:

$$\text{condition} ? \text{term1} : \text{term2}$$

If the condition is true, the expression simplifies to *term1*, and it simplifies to *term2* otherwise. So instead of code like this:

```
String tryword;
if ( tries == 1 )
    tryword = " try";
else
    tryword = " tries";
System.out.println( tries + tryword + " left!" );
```

...you could just write this:

```
System.out.println( tries + ( tries==1 ? " try" : " tries" ) + " left!" );
```

So I could have written:

```
if ( p.equals("X") )
    p = "O";
else
    p = "X";
```

But with the ternary operator that becomes:

```
p = p.equals("X") ? "O" : "X";
```

*p* becomes “O” if it currently equals “X” and *p* becomes “X” otherwise.

Be careful with the ternary operator, because it can lead to ugly code that is hard to read. But if you use it sparingly it can be nice.

Moving on to this line:

```
done = isWinner(board,"X") || isWinner(board,"O") || turns == 9;
```

*done* is a boolean variable. It gets set to the truth value of that expression. So if *X* wins or *O* wins or nine turns have been made, then *done* is set to `true`. Otherwise it gets set to `false`. This isn’t a ternary expression; it is just how booleans work.

So we can loop “while not done”, which is easy on my brain.

The `isWinner()` and the `winCheck()` functions are the same as in our previous tic-tac-toe game. (The *code* is different because we’re using objects now, but the *ideas* are the same.)

Are you following along so far? I hope so. Now it’s time to look at some *useful* inheritance!

## TicTacPiece.java

---

```

1 public class TicTacPiece extends GamePiece {
2     private static int whichOne = 0;
3
4     public TicTacPiece( int r, int c ) {
5         super(r,c, whichOne%2 == 0 ? "X" : "O");
6         whichOne++;
7     }
8 }

```

---

This is a subclass of `GamePiece`. It inherits all the other functionality of the original `GamePiece` class, but adds a private *static* field. (Remember that *static* means that all instances of this object share a single copy of this variable.

So the `whichOne` variable tells us whether or not the next `TicTacPiece` created should be an “X” or an “O”. The constructor calls `super()` with *r* and *c* and one of those two letters depending on how many pieces have been made so far. If it’s an even number (`whichOne%2 == 0`) then we pass in “X” to the superclass constructor.

Also note that we pretty much *needed* to use a ternary operator here, because the call to `super()` must be the *first* line in a constructor. So we couldn’t have put an `if` statement before it.

Then we add 1 to the shared copy of *whichOne* so the next piece will be set up for success.

Do you see how inheritance allows us to easily make a version of a `GamePiece` object that works better for tic-tac-toe but without having to rewrite any code? That’s the idea.

Okay, now a game board just for tic-tac-toe.

## TicTacBoard.java

---

```

1 public class TicTacBoard extends GameBoard {
2     private int numTurns;
3
4     public TicTacBoard() {
5         super(3,3);
6         numTurns = 0;
7     }
8
9     @Override
10    public boolean canAdd( int r, int c ) {
11        return super.canAdd(r,c) && g[r][c] == null;
12    }
13
14    @Override

```

---

```

15     public void add( GamePiece a, int r, int c ) {
16         if ( canAdd(r,c) ) {
17             super.add(a, r, c);
18             numTurns++;
19         }
20     }
21
22     public String currentPlayer() {
23         return numTurns%2 == 0 ? "X" : "O";
24     }
25
26     public boolean isWinner() {
27         return isWinner("X") || isWinner("O");
28     }
29
30     public boolean isOver() {
31         return isWinner() || numTurns == 9;
32     }
33
34     public boolean isWinner( String p ) {
35         if ( winCheck(p, 0,0, 0,1, 0,2) ) return true;
36         if ( winCheck(p, 1,0, 1,1, 1,2) ) return true;
37         if ( winCheck(p, 2,0, 2,1, 2,2) ) return true;
38         if ( winCheck(p, 0,0, 1,0, 2,0) ) return true;
39         if ( winCheck(p, 0,1, 1,1, 2,1) ) return true;
40         if ( winCheck(p, 0,2, 1,2, 2,2) ) return true;
41         if ( winCheck(p, 0,0, 1,1, 2,2) ) return true;
42         if ( winCheck(p, 2,0, 1,1, 0,2) ) return true;
43         return false;
44     }
45
46     private boolean winCheck(String p, int a,int b, int c,int d, int e,int f) {
47         if ( g[a][b] == null || g[c][d] == null || g[e][f] == null )
48             return false;
49         String A = g[a][b].getSymbol();
50         String B = g[c][d].getSymbol();
51         String C = g[e][f].getSymbol();
52         return ( p.equals(A) && A.equals(B) && B.equals(C) );
53     }
54 }

```

---

Our original GameBoard class already has a method called canAdd(). But rather than just checking

if the row and column are in bounds, we *also* want to make sure there’s not already a piece there. So we “override” the `canAdd()` method.

I have decided to annotate this. According to Oracle, the `@Override` annotation “informs the compiler that the element is meant to override an element declared in a superclass. While it is not required to use this annotation when overriding a method, it helps to prevent errors. If a method marked with `@Override` fails to correctly override a method in one of its superclasses, the compiler generates an error.”

Eclipse *requires* the use of this annotation. Though I don’t use Eclipse and I don’t recommend it for beginners, you will see this annotation in real Java code, so I have included it here.

To be clear, you could leave out the override annotation and the code would still work exactly the same.

In addition to overriding the `canAdd()` method, we also override the `add()` method to also keep track of the number of turns.

Finally, we add a bunch of helpful methods so that the `TicTacBoard` can tell us whose turn is next, and whether or not there has been a winner.

Okay, now the moment of truth: we are going to write the tic-tac-toe game *again* using our new subclasses to help us.

`TicTacGame.java`

---

```

1  import java.util.Scanner;
2
3  public class TicTacGame {
4      public static void main( String[] args ) {
5          TicTacBoard board = new TicTacBoard();
6          Scanner kb = new Scanner(System.in);
7          int r, c;
8
9          do {
10             System.out.println("\n"+board);
11             System.out.print("'''+ board.currentPlayer() +'', your turn: ");
12             r = kb.nextInt();
13             c = kb.nextInt();
14             if ( ! board.canAdd(r,c) ) {
15                 System.out.println("\nSorry, you can't go there. Try again.");
16                 continue;
17             }
18             TicTacPiece XO = new TicTacPiece(r, c);
19             XO.addSelfToBoard(board);
20         } while ( !board.isOver() );
21

```

```

22         System.out.println("\n"+board);
23
24         if ( board.isWinner("X") )
25             System.out.println("X wins!");
26         else if ( board.isWinner("O") )
27             System.out.println("O wins!");
28         else
29             System.out.println("It's a tie!");
30     }
31 }

```

---

You will notice that this code is only half the length of the previous version of the game! Notice that when creating a `TicTacPiece` we don't have to tell it whether it should be an “X” or “O” because it knows that already.

The only new, weird thing here is the use of the Java keyword `continue`. It means “go back up to the top of the loop.”

I did this because if you even *create* a `TicTacPiece` object then it throws off whose turn it is. So if they can't add a piece there, it skips the rest of the code in the do-while loop, doesn't bother to check if the condition is true and just starts over at the top.

## What You Should See

```

...
...
...

'X', your turn: 1 1

...
.X.
...

...skip a bunch of steps since we know how it's going to turn out...

00X
XXO
OXX

It's a tie!

```



I should confess that the classes in today’s exercise were not written all at once in a flash of brilliance. I started with rough-draft classes for Piece and Board, then changed them up a bit as I was writing the first tic-tac-toe game. (“It would sure help if the piece could do this.... Oh, the piece needs the board to do this....”)

Then once I started making the inherited classes for the TicTacPiece and TicTacBoard the base classes changed a little as well. And the inherited classes changed as I was writing the second tic-tac-toe game.

You’re only seeing the final version here, not the journey. I’ve been coding since the 1980s and I’ve been coding in an object-oriented style off and on since the mid-2000s and I still can’t “design” an object perfectly on the first try. This is just how coding works.

If you interview at a company that writes code and they claim to have a bunch of design meetings up front where they design dozens of classes on paper and then just code them up, be a little suspicious. You learn what works and what doesn’t as you fit the pieces together.

Anyway, hopefully this exercise helps you to see how inheritance can be useful. And just for fun, we are going to make a different game in the next exercise using the same base classes!



## Study Drills

1. It is a little depressing to make the last player place an “X” or “O” in the last available space, even when there’s no winner. Add a method to the TicTacBoard class that’s true when there have been 8 moves and the last remaining move wouldn’t cause anyone to win, either. Then modify the game to skip that last turn when this occurs. This is a bit harder than it seems.

# Exercise 36: A Game Called Breakthrough

Breakthrough is an award-winning strategic board game invented by Dan Troyka in 2000. And it is much easier to code than it is to win!

Each player starts with 16 pieces filling the first two rows on their side. A piece can move one space forward (either straight ahead or diagonally) if the spaces are empty *or* a piece can move to capture an opponent's piece diagonally only. (Pieces cannot capture each other straight ahead.)

The winner is the first one to reach the opposite home row.

BreakthroughPiece.java

---

```
1 public class BreakthroughPiece extends GamePiece {
2     private int rowDir;
3
4     public BreakthroughPiece( int r, int c, boolean isWhite ) {
5         super(r, c, isWhite ? "W" : "B");
6         if ( isWhite )
7             this.rowDir = -1;
8         else
9             this.rowDir = +1;
10    }
11
12    public boolean canMove( int colDir ) {
13        int newRow = r+rowDir, newCol = c+colDir;
14
15        // Can't move out of bounds.
16        if ( ! board.canAdd(newRow, newCol) )
17            return false;
18
19        // "A piece can be moved one space forward or diagonally forward,
20        //   if the target position is empty."
21        if ( board.get(newRow, newCol) == null )
22            return true;
23
24        BreakthroughBoard bb = (BreakthroughBoard)board;
25        // Can capture an enemy piece diagonally forward only.
26        if ( colDir != 0 && bb.get(newRow, newCol).rowDir != this.rowDir )
```

```

27         return true;
28
29         return false;
30     }
31
32     public void move( int colDir ) {
33         if ( ! canMove(colDir) )
34             return;
35         ((BreakthroughBoard)board).movePiece(r, c, r+rowDir, c+colDir);
36         r += rowDir;
37         c += colDir;
38     }
39 }

```

---

The BreakthroughPiece class extends the GamePiece object from the previous exercise, so it inherits all its fields and methods (except constructors).

The *rowDir* field tells whether the piece moves “north” toward the top of the board or “south”. A value of -1 means that every time the piece moves, -1 will be added to its row value – bringing it closer to the top of the grid.

So in the constructor if the *isWhite* parameter is true, then we pass the symbol “w” to the superclass constructor and then set *rowDir* to -1 since white pieces start at the bottom and move up.

The *canMove()* method says whether or not it’s legal for this piece to move in the given direction. The parameter *colDir* works similarly to *rowDir*; it will have the values -1, 0 or 1 to say whether or not it is trying to move up and left, straight up or up and right.

Since this piece is currently at row *r* and column *c*, then the location it is attempting to move to is *r + rowDir* and *c + colDir*.

If there’s no piece at all currently in that location, then it is okay to move there.

Assuming there *is* a piece there (otherwise we’d have already returned true and wouldn’t make it this far down into the method), then it is still okay to move there *if* we are moving diagonally (*colDir* isn’t 0) *and* the piece there is not the same color as us (its *rowDir* is different than our *rowDir*).

Whew.

If you survived that you might have been wondering about the cast:

```
BreakthroughBoard bb = (BreakthroughBoard)board;
```

The *board* field is inherited from GamePiece. It is a reference to a *GameBoard* object. If you look down in BreakthroughGame.java you can see that the actual object created is a BreakthroughBoard object. Because the BreakthroughBoard object extends GameBoard then it *is* a GameBoard object. So the field *board* can hold a reference to it, too.

The problem is that even though the object *really is* a BreakthroughBoard, the variable referring to it *could* just be holding a reference to a boring GameBoard object.

So if we want to call the BreakthroughBoard's version of the `get()` method instead of the GameBoard's version of the `get()` method, we have to cast it. After the cast, the variable *bb* holds the **exact same** memory address that *board* does, but the compiler now knows for sure that it can call all the methods that are unique to the BreakthroughBoard class.

There's also a cast in the `move()` method so that we can call `movePiece()`.

Okay, so now let's take a look at the differences in the BreakthroughBoard!

#### BreakthroughBoard.java

---

```

1  public class BreakthroughBoard extends GameBoard {
2      public BreakthroughBoard() {
3          super(8,8);
4          for ( int c=0; c<8; c++ ) {
5              createPieceAt(0,c,false);
6              createPieceAt(1,c,false);
7              createPieceAt(6,c,true);
8              createPieceAt(7,c,true);
9          }
10     }
11
12     @Override
13     public BreakthroughPiece get( int r, int c ) {
14         return (BreakthroughPiece)super.get(r,c);
15     }
16
17     public void createPieceAt( int r, int c, boolean isWhite ) {
18         BreakthroughPiece bp = new BreakthroughPiece(r,c,isWhite);
19         bp.addSelfToBoard(this);
20     }
21
22     public void movePiece( int curRow, int curCol, int newRow, int newCol ) {
23         if ( canAdd(curRow,curCol) && canAdd(newRow, newCol) ) {
24             g[newRow][newCol] = g[curRow][curCol];
25             g[curRow][curCol] = null;
26         }
27     }
28
29     public boolean isOver() {
30         return isWinner("W") || isWinner("B");
31     }

```

```

32
33     public boolean isWinner( String symbol ) {
34         int targetRow = symbol.equals("W") ? 0 : 7;
35         for ( int c=0; c<8; c++ )
36             if ( g[targetRow][c].getSymbol().equals(symbol) )
37                 return true;
38         return false;
39     }
40
41     @Override
42     public String toString() {
43         String out = "";
44         for ( int r=0; r<g.length; r++ ) {
45             out += "\t" + (g.length-r) + " ";
46             for ( int c=0; c<g[0].length; c++ ) {
47                 if ( g[r][c] == null )
48                     out += ".";
49                 else
50                     out += g[r][c].getSymbol();
51             }
52             out += "\n";
53         }
54         out += "\t ";
55         for ( int c=0; c<g[0].length; c++ )
56             out += (char)('A'+c);
57         return out+"\n";
58     }
59 }

```

---

Unlike tic-tac-toe, Breakthrough starts with sixteen pieces on the board, so I made a `createPieceAt()` method just to save some code.

Then in the constructor, we call the superclass constructor with 8 rows and 8 columns. Then for each column in the board (0 through 7) we create a black piece in row 0 and row 1 of that column and a white piece in row 6 and row 7.

There's a custom `get()` method for the `BreakthroughBoard`; the only difference is that it returns a reference to a `BreakthroughPiece` object instead of just a reference to a `GamePiece` object. That saves us a little code in a couple of places.

This method *does* override the superclass' `get()` method even though the return type is different because the new return type is a subclass of the original return type. In case you were wondering. (Which I was; I wasn't sure if this would count as overriding and had to look it up!)

I added a `movePiece()` method that just overwrites the new location with the reference from the original location and then sets the original reference to `null`.

There's an `isWinner()` method. (I guess I just like that name.) The "goal" row for white is row 0 (at the top of the grid) and the target row for black is row 7. So I just loop through all eight columns in the target row looking for any piece of that color. As soon as we find one, we can return `true` from the method. And if the loop completes without finding any, then we can return `false`.

Finally I overrode the base class' `toString()` just so I could label the rows with 1-8 and the columns with A-H like you usually see online. That makes it a little easier to play.

The code was a little weird though because row 0 in the 2-D array gets the label "8" and row 1 gets the label "7", etc. We will have to deal with this conversion in the `main()` but I think it is still worth doing because it makes things easier on the humans playing the game.

#### BreakthroughTest.java

---

```
1 public class BreakthroughTest {
2     public static void main( String[] args ) {
3         BreakthroughBoard board = new BreakthroughBoard();
4
5         board.get(6,3).move(0);
6         board.createPieceAt(4,4,false);
7         board.createPieceAt(4,2,true);
8
9         System.out.println("\n"+board);
10        System.out.println( "Can move left: " + board.get(5,3).canMove(-1) );
11        System.out.println( "Can move straight: " + board.get(5,3).canMove(0) );
12        System.out.println( "Can move right: " + board.get(5,3).canMove(1) );
13    }
14 }
```

---

When I was first writing the two classes above I wanted to test the `BreakthroughPiece`'s `canMove()` method without having to deal with the rest of the game. So I made a little tester, which you see above.

You don't have to type it in unless you want to. It can be helpful for finding bugs, though!

## BreakthroughGame.java

```
1  import java.util.Scanner;
2  import static java.lang.System.out;
3
4  public class BreakthroughGame {
5      public static void main( String[] args ) {
6          BreakthroughBoard board = new BreakthroughBoard();
7          Scanner kb = new Scanner(System.in);
8          String col;
9          int row, r, c, colDir;
10         String player = "W";
11         boolean goLeft, goForward, goRight;
12
13         do {
14             out.println("\n"+board);
15             out.print("'" + player + "', your turn: ");
16             col = kb.next();
17             row = kb.nextInt();
18
19             r = 8-row;
20             c = col.charAt(0) - 'A';
21
22             BreakthroughPiece pick = board.get(r,c);
23             if ( pick == null || ! pick.getSymbol().equals(player) ) {
24                 out.println("\nThere's no piece you can move there. Try again.");
25                 continue;
26             }
27
28             goLeft = pick.canMove(-1);
29             goForward = pick.canMove(0);
30             goRight = pick.canMove(1);
31             if ( !goLeft && !goForward && !goRight ) {
32                 out.println("\nThat piece has no legal moves. Try again.");
33                 continue;
34             }
35
36             out.println();
37             do {
38                 if ( goLeft )
39                     out.println("1) Move forward diagonally to the left.");
40                 if ( goForward )
41                     out.println("2) Move forward straight ahead.");
```

```

42         if ( goRight )
43             out.println("3) Move forward diagonally to the right.");
44         out.print("\nWhich move? ");
45         colDir = kb.nextInt();
46         if ( ! pick.canMove(colDir-2) )
47             out.println("\nThat wasn't one of the options. Try again.");
48     } while ( ! pick.canMove(colDir-2) );
49
50     pick.move(colDir-2);
51     player = player.equals("W") ? "B" : "W";
52
53     } while ( !board.isOver() );
54
55     if ( board.isWinner("W") )
56         out.println("Player \"W\" wins!");
57     else
58         out.println("Player \"B\" wins!");
59 }
60 }

```

---

Finally here's the code for the game itself. It's pretty easy, since most of the hard parts have already been handled.

Unlike tic-tac-toe, we don't make the human figure out the raw row number 0-7, we let them type in which column (A-H) and then which row (1-8) as labeled.

So then we have to convert from biggest-to-smallest 1-8 rows to 0-7 rows the other direction. And convert from a String containing the letter "A" to a column number 0-7.

Fortunately Strings have a `charAt()` method which gives us a `char` instead of a `String`, and `chars` are numeric. That is, if you print out a `char` on the screen you'll see a letter shape, but internally they are stored as number variables containing the Unicode value of the letter. So the letter 'A' (note the single quotes) has the value 65 and 'B' is 66.

We then get a reference to the piece they intend to move. We check if there is *no* piece there or if there *is* a piece there but it's the wrong color. If so, we make them pick again.

If we're still around, it means there *was* a piece there and it is the proper color. So we check the three possible moves. If *none* of them are true, then they also need to pick again.

Then we display a little menu of their legal moves. For consistency, left is always 1, straight is always 2 and right is always 3. But only the currently legal options get displayed.

Then we ask them which one they want. Since "left" corresponds to a `colDir` of -1 but a menu choice of 1, I was lazy and just subtracted 2 from their input value.

If they typed a bad choice we complain and make them choose again.



Then, finally, we ask the piece to move itself in the chosen *colDir* and then switch players.

It's virtually impossible to have a "tie" in Breakthrough, so I only checked those two options after the loop ends.

## What You Should See

```

      8 BBBB BBBB
      7 BBBB BBBB
      6 .....
      5 .....
      4 .....
      3 .....
      2 WWWWWW
      1 WWWWWW
      ABCDEFGH

'W', your turn: D 2

1) Move forward diagonally to the left.
2) Move forward straight ahead.
3) Move forward diagonally to the right.

Which move? 2

      8 BBBB BBBB
      7 BBBB BBBB
      6 .....
      5 .....
      4 .....
      3 ...W....
      2 WWW.WWWW
      1 WWWWWW
      ABCDEFGH

'B', your turn:

```

Pretty neat, huh? It's nice that you can code a real strategy game like this in only 60 lines of code, and a lot of that is error-checking!

I think this game does a pretty good job of showing off how object-oriented programming and inheritance can be helpful. We can make simple modules for a board and for a piece and then make more specific versions of those modules and fit them together without going crazy.

It's a tradeoff, I guess. The code gets a little more complex because of all the object-oriented stuff and inheritance and overriding methods and casting. But then when you're done you can write a game in sixty lines of code and understand everything.

More about overriding methods in the next exercise!



## Study Drills

1. Once a player has picked which piece they are going to move, show them visually what their legal moves are. Maybe put a '#' or something in the squares they are able to move to. But be sure to remove them afterward and don't accidentally overwrite one of the other pieces!
2. See if you can find a way to highlight the piece they have chosen to move. I thought about changing it to a lowercase 'w' or 'b' temporarily, but that will probably mess with some of the code in `isWinner()` and `canMove()`. So figure out something else! :) (Mr. Mitchell's real-life students don't need to do this Study Drill to receive credit.)

# Exercise 37: Two Kinds of Equality

You have seen a lot of `if` statements comparing two things. By now maybe you have learned that primitive types get to use `==` for comparisons but “objects” must use `equals()`.

In this exercise, we’ll see how to override the default `equals()` method that gets inherited from `Object`!

We have been looking at inheritance for the past several exercises and how to extend other objects. But did you know that *all* Java objects actually inherit? That’s right,

```
public class Dude {
```

...and

```
public class Dude extends Object {
```

...are the same! *Every* class in Java extends the built-in class `Object` *unless* it extends something else. `Object` is the base class for every object in Java. As Oracle’s [documentation](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html)<sup>14</sup> says, “Class `Object` is the root of the class hierarchy. Every class has `Object` as a superclass. All objects, including arrays, implement the methods of this class.”

Among the helpful methods defined in `Object` are `equals()` and `toString()`.

Dude.java

---

```
1 public class Dude {
2     private int age;
3     private String name;
4
5     public Dude(String name, int age) {
6         this.name = name;
7         this.age = age;
8     }
9
10    @Override
11    public boolean equals(Object obj) {
12        Dude other = (Dude)obj;
13        return ( this.age == other.age && this.name.equals(other.name) );
```

---

<sup>14</sup><https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html>

```

14     }
15
16     @Override
17     public String toString() {
18         return name + "|" + age;
19     }
20 }

```

---

The Dude class extends Object. So in this object we are overriding the versions of equals() and toString() that are inherited from the Object class.

There's a small complication when overriding equals(). The method expects a parameter of type Object, *not* of type *Dude*. So assuming they pass in a Dude object, we “cast” the reference from type Object back to type Dude like so:

```
Dude other = (Dude)obj;
```

This will blow up if the object they sent in is some *other* kind of object, so:

```

Dude a = new Dude("Dynamite", "Bamford");
String s = "fuse";
if ( a.equals(s) ) // BOOM

```

...will blow up the program instead of just being false. That's not good. We should probably write equals() differently to check for that:

```

if ( obj == null )
    return false; // this object is not equal to the 'null' reference, okay?
Dude other = null;
try {
    other = (Dude)obj;
}
catch ( ClassCastException e ) {
    return false; // they're not equal if it isn't even a Dude
}
return ( this.age == other.age && this.name.equals(other.name) );

```

That would be much nicer. But you're still learning, so what we did is good enough.

The last line of the equals() method just means “return true if the ages are the same AND the names equal each other, or return false if not.” You could also write:

```
if ( this.age == other.age && this.name.equals(other.name) )
    return true;
else
    return false;
```

That would work exactly the same, but I like the way I did it better.

DudeDriver.java

---

```
1 public class DudeDriver {
2     public static void main( String[] args ) {
3         Dude one = new Dude("Mitch", 39);
4         Dude two = new Dude("Mitch", 39);
5         Dude three = one;
6         Dude four = new Dude("Curly", 71);
7
8         if ( one.equals(two) )
9             System.out.println(one + " equals " + two);
10        else
11            System.out.println(one + " does not equal " + two);
12
13        if ( one.equals(three) )
14            System.out.println(one + " equals " + three);
15        else
16            System.out.println(one + " does not equal " + three);
17
18        if ( one.equals(four) )
19            System.out.println(one + " equals " + four);
20        else
21            System.out.println(one + " does not equal " + four);
22
23        System.out.println();
24        if ( one == two )
25            System.out.println(one + " == " + two);
26        else
27            System.out.println(one + " != " + two);
28
29        if ( one == three )
30            System.out.println(one + " == " + three);
31        else
32            System.out.println(one + " != " + three);
33    }
34 }
```

---

The driver just creates some `Dude` objects and compares them. `Dude one` and `Dude two` are “equivalent” objects: they are two different objects with the same field values. `Dude one` and `Dude three` are two different references to the same single object.

## What You Should See

```
Mitch-39 equals Mitch-39
Mitch-39 equals Mitch-39
Mitch-39 does not equal Curly-71

Mitch-39 != Mitch-39
Mitch-39 == Mitch-39
```

So `one.equals(two)` and `one.equals(three)`, as expected. But `one == three` because the references (the shallow values) are also the same.



## Study Drills

1. Add the try-catch and the check for `null` to `Dude`'s `equals()` method.
2. Add a field to `Dude` for weight/mass. Make it a `double`, and add it to the constructor and the `toString()` method. Then change the `equals()` method so that the ages and names have to match *and* the weights have to be within `0.01` of each other.

# Exercise 38: Implementing Interfaces

We have been learning about inheritance. Inheriting code from another class can be a good way to reuse code. But in Java (and in most “newer” programming languages), you can only inherit from a *single* class.

If you want combined functionality from more than one place, you need something else. In Java, that “something else” is an *interface*.

An interface is like a contract; it is a promise to implement certain methods. Let’s start with a small example. Notice that it begins with `public interface` rather than `public class`.

ArbitraryInterface.java

---

```
1 public interface ArbitraryInterface {
2     public double getNumber();
3     public void doSomething(int n);
4 }
```

---

There is no missing code; that’s it! Classes which choose to implement the `ArbitraryInterface` are promising to have two public methods: one called `getNumber()` and one called `doSomething()`. The first *must* have no parameters and must return a `double` in order to fulfill the contract. The second must have a single integer parameter and must return nothing. (The integer parameter doesn’t have to be called *n*, though.)

So now here is a class that successfully *implements* this interface:

InterfacePleaser.java

---

```
1 public class InterfacePleaser implements ArbitraryInterface {
2     public double getNumber() {
3         return 4;
4     }
5
6     public void doSomething(int n) {
7         n = Math.abs(n);
8         for ( int i=1; i<=n; i++ ) {
9             System.out.println(i);
10        }
11    }
12 }
```

---

```

13     public int doSomethingElse(int x) {
14         return x*x;
15     }
16 }

```

---

This class implements the interface `ArbitraryInterface`. It has to have those two methods and it does! Notice that the interface doesn't care what the methods **do**. They just have to be present.

In this case, I decided to always return 4 from the required `getNumber()` method and to print out the numbers from 1 up to the absolute value of the parameter in the required `doSomething()` method.

This class also has a third method! The interface doesn't care about that. It's fine.

Okay, so here is another class that successfully implements the same interface.

#### InterfacePleaser2.java

---

```

1  public class InterfacePleaser2 implements ArbitraryInterface {
2      private int max;
3
4      public InterfacePleaser2() {
5          this(0);
6      }
7
8      public InterfacePleaser2( int max ) {
9          this.max = max;
10     }
11
12     public int getMax() { return max; }
13
14     public void doSomething( int somethingOtherThanN ) {
15     }
16
17     public double getNumber() {
18         return Math.random()*max;
19     }
20 }

```

---

This class has some constructors and a field and a getter method, too, but it provides implementations for the two required methods, so it satisfies the contract.

(Inside the default constructor there's a call to `this()`. This has nothing to do with interfaces. Just like `super()`, it's only legal as the first line of a constructor, and just calls a *different* constructor of the same class. In this case, it calls the parameter constructor but passes in 0 as the max.)

Now let us look at a class that tries to implement an interface, but fails. There is no reason to type it in unless you don't believe me that it doesn't compile.



**InterfaceDisappointment.java**

---

```
1 // DOES NOT COMPILE
2 public class InterfaceDisappointment implements ArbitraryInterface {
3     public void doSomething(double n) {
4         System.out.println(n);
5     }
6 }
```

---

This class fails for a couple of reasons. There are two required methods in the interface, but this class only tries to implement one of them. Also, the `doSomething()` method must have an integer parameter, but in this class the parameter is a double instead.

This class doesn't fulfill the contract, so it doesn't compile. That's a good thing! Part of the idea with interfaces is that the compiler can help enforce the contract.

Finally, here is some code that uses these classes.

**InterfaceDriver.java**

---

```
1 public class InterfaceDriver {
2     public static void main( String[] args ) {
3         ArbitraryInterface a1 = new InterfacePleaser();
4         ArbitraryInterface a2 = new InterfacePleaser2(10);
5         // ArbitraryInterface a3 = new ArbitraryInterface(); // WON'T COMPILE
6
7         a1.doSomething(5);
8         a2.doSomething(5);
9
10        System.out.println( a1.getNumber() );
11        System.out.println( a2.getNumber() );
12    }
13 }
```

---

The variable *a1* can hold a reference to *any* object that implements the `ArbitraryInterface`. This is pretty cool and it's a form of polymorphism.

*a1* holds a reference to an instance of the first implementing class and *a2* holds a reference to the other one. This is quite powerful.

It's also important to note that we can *not* actually instantiate an `ArbitraryInterface` *object*. That isn't a class, so we can't create one.

Using those references we can call both methods from the interface. We can't call any other methods, even if the classes have them, because the interface only guarantees those two.

## What You Should See

```
1
2
3
4
5
4.0
1.3093886978048475
```

It is okay at this point if you are a little confused about *why* you would want to create or implement an interface. We will see examples of this in future exercises.



### Study Drills

1. Create a third class that implements the `ArbitraryInterface`. Call the file `InterfacePleaser3.java`, but implement the methods however you like. Also add a bit of code to the driver to instantiate your new class.

# Exercise 39: The Comparable Interface

In this exercise I'm going to attempt to show how implementing an interface can be *useful*. We'll get there by the end of the exercise, I swear!

There is a built-in interface in Java called `Comparable`. (It is found inside `java.lang`, so you don't need to import anything to get to it.)

It looks like this:

```
public interface Comparable {  
    public int compareTo(Object o);  
}
```

It is very simple. In order to be “Comparable”, a class just needs to implement one method: `compareTo()`. That method *should* return a negative number if it is “less” than the parameter, 0 if it is equal in some sense and return a positive number if it's “greater” than the parameter.

Of course, all it *has* to do is accept an `Object` parameter and return any integer at all, but it is a lot more useful if you play along.

Buddy.java

---

```
1 public class Buddy implements Comparable {  
2     private int age;  
3     private String name;  
4  
5     public Buddy(String name, int age) {  
6         this.name = name;  
7         this.age = age;  
8     }  
9  
10    public boolean equals(Object obj) {  
11        Buddy other = (Buddy)obj;  
12        return ( this.age == other.age && this.name.equals(other.name) );  
13    }  
14  
15    public int compareTo(Object obj) {  
16        Buddy other = (Buddy)obj;
```

```

17         if ( this.name.equals(other.name) )
18             return (this.age - other.age);
19         else
20             return this.name.compareTo(other.name);
21     }
22
23     public String toString() { return name + "-" + age; }
24 }

```

---

Here is a boring little class that implements the Comparable interface. (And also overrides the default equals() method.)

Remember that what they're passing in is an Object, not specifically Buddy, so we have to cast it.

If the names are the same, we return the *difference* between the ages, which will end up being a positive number if this is older than *other*, zero if they're the same, and a negative number if this is younger.

If the names are different, we cheat and just return the same number that the String class' compareTo() method gives us.

#### BuddyTester.java

---

```

1  import java.io.File;
2  import java.util.Scanner;
3
4  public class BuddyTester {
5      public static void main( String[] args ) throws Exception {
6
7          int correct = 0;
8          int total  = 0;
9          Scanner in = new Scanner(new File("datafiles/buddytests.txt"));
10         while ( in.hasNext() ) {
11             String name1 = in.next();
12             int age1 = in.nextInt();
13             String name2 = in.next();
14             int age2 = in.nextInt();
15             int wanted = in.nextInt();
16
17             total++;
18             int got = test(name1,age1 , name2,age2);
19             if ( got == wanted )
20                 correct++;
21             else {

```

```

22         System.out.print("\t" + name1 + "-" + age1 + " compared with ");
23         System.out.print(name2 + "-" + age2 + " should have given ");
24         System.out.println(wanted + ", but I got " + got + " instead.");
25     }
26 }
27 in.close();
28 System.out.println(correct + " out of " + total + " tests passed.");
29
30 }
31
32 public static int test( String n1,int a1, String n2,int a2 ) {
33     Buddy one = new Buddy(n1, a1);
34     Buddy two = new Buddy(n2, a2);
35     int num = one.compareTo(two);
36     if ( num == 0 )
37         return 0;
38     return (num / Math.abs(num));
39 }
40 }

```

---

This class tests out our implementation of `compareTo()` by using a bunch of test cases. The file 'buddytests.txt' has a bunch of pairs of names and ages in it *and* the expected output of the comparison.

For example, the first two lines of the file look like this:

Bob	30	Bob	30	0
Alice	25	Bob	30	-1

So if we compare a Buddy with name “Bob” and age 30 to *another* Buddy with the same name and the same age we would expect `compareTo()` to tell us 0.

If we compare a Buddy with the name “Alice”, age 25 to an object with the name “Bob”, age 30, we would expect a negative number back since “Alice” comes before “Bob”.

The static method `test()` does this for us. It creates two Buddy objects from the data provided and collects the result of running our `compareTo()` method. If it gives us zero we return that. Otherwise, we divide the number it gives us by that number’s absolute value. So if *num* is -3, then  $-3 / 3$  is -1. (This is because the `compareTo()` method is only guaranteed to return *some* negative number but the test cases in the file are always either -1 or +1.)

So the tester program here reads all the values from the file, sends them in to the `test()` method to see what our `compareTo()` method will say about them. If it matches the expected output, we count it correct! Otherwise, we complain.

## What You Should See

8 out of 8 tests passed.

Kind-of boring, I know. But that's because we did it right! Our `compareTo()` method is correct, so all the tests pass.

Now, just saying `implements Comparable` is not actually how things are currently done in Java. Since generics were added to Java, the “new” `Comparable` interface now looks like this:

```
public interface Comparable<T> {  
    public int compareTo(T o);  
}
```

The `Comparable` interface now has a generic parameter, which makes it a *tiny* bit easier to write our code.

Guy.java

```
1 public class Guy implements Comparable<Guy> {  
2     private int age;  
3     private String name;  
4  
5     public Guy(String name, int age) {  
6         this.name = name;  
7         this.age = age;  
8     }  
9  
10    public boolean equals(Object obj) {  
11        Guy other = (Guy)obj;  
12        return ( this.age == other.age && this.name.equals(other.name) );  
13    }  
14  
15    public int compareTo(Guy other) {  
16        if ( this.name.equals(other.name) )  
17            return (this.age - other.age);  
18        else  
19            return this.name.compareTo(other.name);  
20    }  
21}
```

```

22     public String toString() { return name + "-" + age; }
23 }

```

---

The Guy class here is pretty much identical to the Buddy class except that it implements `Comparable<Guy>`. This means that the compiler will make sure that we are only attempting to `compareTo()` one Guy object with another Guy object.

You can see in the `compareTo()` method the parameter is an object of type Guy, not just `Object`, and we don't have to use a cast at the beginning, either. (We still have to have a cast at the beginning of `equals()` since *that* method is overriding the one inherited from `Object` and that's how it is defined originally.)

Anyway, I promised you something *useful*.

There is a utility class in Java called `Collections`. It has several useful static methods, including one that will sort the values in an `ArrayList`. That's handy. Sorting is annoying to write, as you may remember.

There's only one problem, however. `ArrayLists` only hold references to objects, and not all objects in Java can be compared. Does it make sense for a `Scanner` object to be "less than" another `Scanner` object? What about the `Math` class? Can one `Math` object be "greater" than another?

So `Collections.sort()` is defined something like this:

```
public static void sort( List<Comparable> list )
```

(It's not *actually* defined like that, so look it up if you're brave and curious.)

This means that `Collections.sort()` can sort any `ArrayList` holding references to *Comparable* objects. So if the `ArrayList` only holds references to objects that implement the `Comparable` interface, then it can sort that `ArrayList` using the objects' `compareTo()` method. Neat, huh?

GuyDriver.java

---

```

1  import java.util.Scanner;
2  import java.util.ArrayList;
3  import java.util.Collections;
4
5  public class GuyDriver {
6      public static void main( String[] args ) {
7
8          ArrayList<Guy> list = getListFromFile("datafiles/buddytests.txt");
9          System.out.println(list+"\n");
10
11         Collections.sort(list);
12         for ( Guy g : list )

```

```
13         System.out.println(g);
14     }
15
16     public static ArrayList<Guy> getListFromFile( String filename ) {
17         ArrayList<Guy> list = new ArrayList<>();
18         Scanner in = null;
19
20         try {
21             in = new Scanner(new java.io.File(filename));
22         }
23         catch ( java.io.FileNotFoundException e ) {
24             System.err.println("Couldn't open 'buddytests.txt': " + e);
25             System.exit(1);
26         }
27
28         while ( in.hasNext() ) {
29             Guy g = new Guy(in.next(), in.nextInt());
30             if ( ! list.contains(g) )
31                 list.add(g);
32             in.next(); // read these from the file but just throw them away
33             in.nextInt();
34             in.nextInt();
35         }
36         in.close();
37         return list;
38     }
39
40 }
```

---

This program creates Guy objects from the first part of each line in our data file and adds all the unique ones to an ArrayList of Guy objects.

It then uses Collections.sort() to sort them and then displays the result.

## What You Should See





```
[Bob-30, Alice-25, Bob-25, Carol-35, Eve-35, Mallory-40, Mallory-20]
```

```
Alice-25  
Bob-25  
Bob-30  
Carol-35  
Eve-35  
Mallory-20  
Mallory-40
```

Think about what just happened. Inside the `getListFromFile()` method, the `ArrayList` code written by the creators of Java was able to use a `contains()` method to tell if there was already an identical `Guy` object in the list.

Somebody wrote that `contains()` method years ago. They had no way of knowing that in 2016 I would create an object called `Guy` and want them to work with it. But yet their code can work with my code just because I defined a custom `equals()` method for my new class.

And `Collections.sort()` was able to sort an `ArrayList` of objects the creators of Java never dreamed of, and all I had to do was implement `Comparable` in my class.

New code running old code is pretty common: any programming language can do that. But when old code can run my new code, that's useful.

Java has quite a few interfaces built-in. When you write code that implements one of those interfaces, you allow lots of *other* built-in Java code to be able to work with your code for free. And that's pretty neat.

We will look at another popular Java interface in the next exercise.



## Study Drills

1. Edit one of the lines in the `buddytests.txt` data file so that one of the tests now fails. (Then change it back.)
2. Add code to `GuyDriver.java` that uses the `indexOf()` `ArrayList` method to find the index of a particular `Guy` object in the list.

# Exercise 40: List and Map

In this exercise we are going to look at two standard Java interfaces that are very common. One of them you have been using throughout this entire book!

Both interfaces have too many methods to list here, but you can check out the official javadocs:

- [java.util.List](#)<sup>15</sup>
- [java.util.Map](#)<sup>16</sup>

Notice in particular the list of “All Known Implementing Classes”.

A list is an object that stores objects with an index. A map is an object that maps keys to values. (In other programming languages, a “map” is called a dictionary or a hash.)

In this code, I’m using a map to store how many times words appear in a long sample of English text. The keys are the words and the “values” are Integer objects to hold the counts.

ListAndMap.java

---

```
1  import java.util.Scanner;
2  import java.util.Map;
3  import java.util.HashMap;
4  import java.util.TreeMap;
5  import java.util.List;
6  import java.util.ArrayList;
7  import java.util.LinkedList;
8
9  public class ListAndMap {
10
11      public static void main( String[] args ) throws Exception {
12          Map<String, Integer> wordCounts;
13          if ( Math.random() < 0.5 )
14              wordCounts = new TreeMap<>();
15          else
16              wordCounts = new HashMap<>();
17
18          Scanner input = new Scanner(new java.io.File("datafiles/en-sample.txt"));
```

---

<sup>15</sup><http://docs.oracle.com/javase/8/docs/api/java/util/List.html>

<sup>16</sup><http://docs.oracle.com/javase/8/docs/api/java/util/Map.html>

```
19      buildMap(input, wordCounts);
20      input.close();
21
22      List<String> winners;
23      if ( Math.random() < 0.5 )
24          winners = new ArrayList<>();
25      else
26          winners = new LinkedList<>();
27
28      buildList(wordCounts, winners, 1000);
29      java.util.Collections.sort(winners);
30      System.out.println(winners);
31
32      Scanner kb = new Scanner(System.in);
33      String word;
34      do {
35          System.out.print("\nEnter a word or ENTER to quit: ");
36          word = kb.nextLine().toLowerCase();
37          if ( word.length() == 0 )
38              break;
39          int count = wordCounts.getDefault(word, 0);
40          System.out.println(word + ": " + count);
41      } while ( true );
42  }
43
44  public static void buildMap( Scanner source, Map<String, Integer> map ) {
45      while ( source.hasNext() ) {
46          String s = source.next().toLowerCase();
47          if ( ! map.containsKey(s) ) {
48              map.put(s, 1);
49          }
50          else {
51              int current = map.get(s);
52              map.replace(s, current+1);
53          }
54      }
55  }
56
57  public static void buildList( Map<String, Integer> map,
58      List<String> list, int cutoff ) {
59      for ( String key : map.keySet() )
60          if ( map.get(key) > cutoff )
```

```
61         list.add(key);
62     }
63 }
```

---

The Map object is defined to map Strings to Integers. And even though it is sort of silly, I randomly instantiated *wordCounts* with either a TreeMap or a HashMap object, since it doesn't matter! Both classes implement the interface, so either one will work.

In real life, you'd pick the one that has the performance characteristics you are looking for. TreeMap is implemented using a self-balancing binary search tree called a "red-black tree", and features pretty fast lookups and relatively efficient in-order traversal of the keys.

HashMap is implemented using a hash table, which provides very fast lookups but isn't very good at in-order traversal, and you might have to care about the "load factor".

But a nice thing about using an interface is that if you don't know the difference or don't understand any of that or don't care, you can just use the interface.

Well, sort-of. Interfaces can't be instantiated, so there's no way to just say

```
Map<String, Integer> wordCounts = new Map<>();
```

You *have* to pick an implementation but as long as you stick to the methods defined in the interface you can change your mind later and you'll only need to change one line of code.

The *buildMap()* function ("static method") isn't very object-oriented. You pass into it a "full" Scanner and an empty Map and it fills the map up.

It reads a word, and if that word isn't already used as a key in the map, it adds it with an associated value of 1. If the word *is* already found, it *get()*s the current value and then replaces it with a larger value.

The *buildList()* function isn't very object-oriented, either. You pass in a full Map and an empty List and it adds any word to the list with more than a certain number of occurrences.

It's only three lines of code, though!

Finally, the *main()* prompts the user for a word and looks it up in the Map. Maps have a method called *getOrDefault()* which returns the value for a given key *if* the key is present. Otherwise, it returns the default value you give.

I also take advantage of the fact that the Scanner's *nextLine()* method reads everything you type until you press ENTER. Unlike *next()*, if you type *nothing* and then press ENTER, it will read nothing and return an empty String.

Inside the do-while loop I also used a weird technique which is called "loop-and-a-half": I loop forever and then break when the single condition is true. Without this technique, I would have needed to write something like this:

```
do {  
    System.out.print("\nEnter a word or ENTER to quit: ");  
    word = kb.nextLine().toLowerCase();  
    if ( word.length() > 0 ) {  
        int count = wordCounts.getOrDefault(word, 0);  
        System.out.println(word + ": " + count);  
    }  
} while ( word.length() > 0 );
```

It might seem like a small thing, but I don't like having to write the same boolean condition twice: once in the if statement inside the do-while loop and again in the condition of the do-while.

There are other ways to deal with this “problem”, but this is the way I personally prefer.

## What You Should See

```
[a, about, all, an, and, are, as, at, be, been, but, by, could, did, do, for,  
from, had, has, have, he, her, him, his, i, if, in, into, is, it, its, jean,  
like, little, man, me, more, my, no, not, of, old, on, one, only, or, out,  
said, she, so, some, than, that, the, their, them, then, there, these, they,  
this, those, to, two, up, very, was, we, were, what, when, which, who, will,  
with, would, you, your]
```

```
Enter a word or ENTER to quit: cat  
cat: 41
```

```
Enter a word or ENTER to quit: mouse  
mouse: 8
```

```
Enter a word or ENTER to quit: dog  
dog: 72
```

```
Enter a word or ENTER to quit: donald  
donald: 1
```

```
Enter a word or ENTER to quit: hillary  
hillary: 0
```

```
Enter a word or ENTER to quit:
```

Java has a *lot* of interfaces built-in with several implementations each. A lot of the time when creating a program in Java you end up finding something built-in that does part of what you need and then figuring out how to hook it up to some other built-in thing.



## Study Drills

1. Add a method called `theVeryBest()` that has a `Map` passed into it and which returns a `String`. It should look through all the keys in the `Map` and return the key with the *largest* count.
2. Add a method called `longTail()`. You should pass in a “full” `Map` and an empty `List`. It should fill up the `List` with all the words that have a count of exactly 1.

# Exercise 41: Implementing Several Interfaces

In Java, classes are only allowed to extend/inherit from a single base class, but they can implement as many interfaces as they want. For example, the `TreeMap` class implements five interfaces: `Serializable`, `Cloneable`, `Map`, `NavigableMap` and `SortedMap`!

But we haven't done that ourselves yet. So here is *another* two-dimensional board game piece to demonstrate that. I apologize if it seems like this is the only sort of program I know how to make. Object-oriented programming doesn't always make sense for small programs, and games are an exception to that.

Hopefully I'll make it up to you, though; we are going to make a graphical version of this game in a later exercise. You'll even use the mouse!

This piece class does *not* inherit from the `GamePiece` object we created a few exercises back. There's no great reason for that except that I'm trying to demonstrate different things with this code.

But first an interface!

`Locatable.java`

---

```
1 public interface Locatable {  
2     Location getLocation();  
3 }
```

---

Pretty simple. Things which are "Locatable" must have a method called `getLocation()` that returns a `Location` object. (We will define that object shortly.)

`Translatable.java`

---

```
1 public interface Translatable {  
2     public void translate(int dx, int dy);  
3 }
```

---

In geometry and in computer graphics, "translate" means to change an object's position without changing its size or shape. So we're going to make an interface that guarantees a `translate()` method.

## Location.java

---

```
1 public class Location implements Comparable<Location> {
2     public int row;
3     public int col;
4
5     public Location() {
6         row = col = 0;
7     }
8
9     public Location( int r, int c ) {
10         row = r;
11         col = c;
12     }
13
14     public boolean equals( Object obj ) {
15         Location other = (Location)obj;
16         return this.row == other.row && this.col == other.col;
17     }
18
19     public int compareTo( Location other ) {
20         if ( this.row == other.row )
21             return this.col - other.col;
22         return this.row - other.row;
23     }
24
25     public Location below() {
26         return new Location(this.row+1, this.col);
27     }
28
29     public Location above() {
30         return new Location(this.row-1, this.col);
31     }
32
33     public Location left() {
34         return new Location(this.row, this.col-1);
35     }
36
37     public Location right() {
38         return new Location(this.row, this.col+1);
39     }
40
41     public String toString() {
```



```
42         return "(" + row + ", " + col + ")";
43     }
44 }
```

---

A Location object just encodes a row and a column, but also has helper methods to give you adjacent locations. You'll notice that this class also implements an interface.

#### DropGamePiece.java

---

```
1  public class DropGamePiece
2      implements Locatable, Comparable<DropGamePiece>, Translatable {
3      private Location loc;
4      private String symbol;
5
6      public DropGamePiece() {
7          loc = new Location(0,0);
8          symbol = "x";
9      }
10
11     public DropGamePiece(int r, int c, String sym) {
12         this(new Location(r,c), sym);
13     }
14
15     public DropGamePiece(Location loc, String sym) {
16         this.loc = loc;
17         symbol = sym;
18     }
19
20     public Location getLocation() {
21         return loc;
22     }
23
24     public String getSymbol() {
25         return symbol;
26     }
27
28     public int compareTo( DropGamePiece other ) {
29         return loc.compareTo(other.loc);
30     }
31
32     public void translate(int dr, int dc) {
33         loc.row += dr;
34         loc.col += dc;
```

```
35     }
36
37     public String toString() {
38         return "DropGamePiece " + symbol + " at " + loc;
39     }
40 }
```

---

This class implements the two interfaces we created and also implements a third interface that is built-in to Java. The class itself is pretty boring, though. It does have two different constructors, so you can create a piece using a row and column as integers or just using a Location object.

#### DropGamePieceTester.java

---

```
1 public class DropGamePieceTester {
2     public static void main( String[] args ) {
3         DropGamePiece one = new DropGamePiece(5,5, "A");
4         Location oneLoc = one.getLocation();
5         DropGamePiece two = new DropGamePiece(oneLoc.below(), "B");
6         System.out.println( one + "\n" + two );
7         two.translate(-1,0);
8         System.out.println( one + "\n" + two );
9         System.out.println( one.compareTo(two) );
10    }
11 }
```

---

This is just a little tester to make sure the class works.

## What You Should See

```
DropGamePiece A at (5, 5)
DropGamePiece B at (6, 5)
DropGamePiece A at (5, 5)
DropGamePiece B at (5, 5)
0
```

This was a pretty short exercise but we'll make up for it next time when we actually write the code for a text version of our game.



## Study Drills

1. Implement an `equals()` method for the `DropGamePiece` that returns `true` if the two pieces have the same symbol and the same `Location`. You must use the `Location`'s `compareTo()` method to do it, and remember that `equals()` is passed an `Object` that it must cast.

# Exercise 42: DropGame and Assertions

In this exercise, we are going to write the code for a game similar to Connect Four by Milton Bradley.

Two players take turns dropping pieces into a grid. The winner is the first person to get one piece to match three of its neighbors. (Unlike Connect Four, our game does not check for four in a row.)

Like we did with the original NoughtsAndCrosses, we are going to create an object that contains most of the rules in the game.

Look for the keyword `assert` in here; it is new!

DropGameHelper.java

---

```
1 public class DropGameHelper
2 {
3     private DropGamePiece[][] board;
4     private int numRows, numCols;
5
6     public DropGameHelper( int numRows, int numCols ) {
7         board = new DropGamePiece[numRows][numCols];
8         this.numRows = numRows;
9         this.numCols = numCols;
10
11         for ( int r=0; r<numRows; r++ )
12             for ( int c=0; c<numCols; c++ )
13                 board[r][c] = null;
14     }
15
16     public int numRows() { return numRows; }
17     public int numCols() { return numCols; }
18
19     public boolean isWinner( String p ) {
20         for ( int r=0; r<numRows; r++ )
21             for ( int c=0; c<numCols; c++ )
22                 if ( winCheck(p, board[r][c]) )
23                     return true;
24         return false;
25     }
```

```
26
27     private boolean winCheck( String p, DropGamePiece piece ) {
28         if ( piece == null )
29             return false;
30         if ( ! p.equals(piece.getSymbol()) )
31             return false;
32         Location loc = piece.getLocation();
33         if ( ! isValid(loc) )
34             return false;
35
36         Location left, right, below, above;
37         int adjacentCount = 0;
38         left = loc.left();
39         right = loc.right();
40         below = loc.below();
41         above = loc.above();
42         if ( isValid(left) && p.equals(this.playerAt(left)) )
43             adjacentCount++;
44         if ( isValid(right) && p.equals(this.playerAt(right)) )
45             adjacentCount++;
46         if ( isValid(below) && p.equals(this.playerAt(below)) )
47             adjacentCount++;
48         if ( isValid(above) && p.equals(this.playerAt(above)) )
49             adjacentCount++;
50
51         return adjacentCount >= 3;
52     }
53
54     public boolean isFull() {
55         int usedCount = 0;
56         for ( int r=0; r<numRows; r++ )
57             for ( int c=0; c<numCols; c++ )
58                 if ( board[r][c] != null )
59                     usedCount++;
60
61         return usedCount == numRows*numCols;
62     }
63
64     public boolean isFull(int col) {
65         assert isValid(col);
66         int nullCount = 0;
67         for ( int r=0; r<numRows; r++ )
```

```
68         if ( board[r][col] == null )
69             nullCount++;
70
71     return nullCount == 0;
72 }
73
74 public boolean isValid( Location loc ) {
75     return ( 0 <= loc.row && loc.row < numRows
76             && 0 <= loc.col && loc.col < numCols
77     );
78 }
79
80 public boolean isValid( int col ) {
81     return ( 0 <= col && col < this.numCols );
82 }
83
84 public String playerAt( Location loc ) {
85     if ( isValid(loc) && board[loc.row][loc.col] != null )
86         return board[loc.row][loc.col].getSymbol();
87     else if ( isValid(loc) && board[loc.row][loc.col] == null )
88         return " ";
89     else
90         return "ERROR:" + loc;
91 }
92
93 public String toString() {
94     String out = "", line;
95     for ( int r=0; r<numRows; r++ ) {
96         line = "|";
97         for ( int c=0; c<numCols; c++ ) {
98             DropGamePiece piece = board[r][c];
99             if ( piece == null )
100                 line += "_|";
101             else
102                 line += piece.getSymbol() + "|";
103         }
104         out += line + "\n";
105     }
106     // draw numbers under each column
107     line = "|";
108     for ( int c=0; c<numCols; c++ )
109         line += c + "|";
```

```

110         out += line + "\n";
111         return out;
112     }
113
114     private int findLowestEmptyRow( int col ) {
115         assert isValid(col);
116         assert ! isFull(col);
117         int row = numRows-1;
118         while ( board[row][col] != null )
119             row--;
120         return row;
121     }
122
123     public void playMove( String p, int col ) {
124         assert isValid(col);
125         assert ! isFull(col);
126         int row = findLowestEmptyRow(col);
127         board[row][col] = new DropGamePiece(row, col, p);
128     }
129 }

```

---

You first see `assert` in the second version of the `isFull()` method. An assertion is designed to indicate a bug or some other serious flaw. If the code is run with assertions turned on, then the `assert` statement checks the condition and blows up the program if it is false.

In this case, we are asserting that the column parameter to `isFull()` is valid. If not, the program shuts down. This isn't really a very *good* use of assertions, because you're not supposed to use them to check parameters, but I wanted to include them in the book and this seemed like a good place to do it.

There are also some assertions in `playMove()`.

#### DropGameConsole.java

---

```

1  import java.util.Scanner;
2
3  public class DropGameConsole {
4      public static void main( String[] args ) {
5          Scanner keyboard = new Scanner(System.in);
6
7          String p = "O";
8          DropGameHelper game = new DropGameHelper(6, 7);
9          int col;
10

```

```

11     while ( !(game.isWinner("O") || game.isWinner("#") || game.isFull()) ) {
12         System.out.println(game);
13         System.out.print("'" + p + "'", choose your column: ");
14         col = keyboard.nextInt();
15
16         while ( ! game.isValid(col) || game.isFull(col) ) {
17             if ( game.isValid(col) == false )
18                 System.out.println("Not a valid location. Try again.");
19             else if ( game.isFull(col) )
20                 System.out.println("That column is full. Try again.");
21
22             System.out.print( "Choose your column: " );
23             col = keyboard.nextInt();
24         }
25         game.playMove(p, col);
26
27         if ( p.equals("O") )
28             p = "#";
29         else
30             p = "O";
31     }
32
33     System.out.println(game);
34
35     if ( game.isWinner("O") )
36         System.out.println("O is the winner!");
37     else if ( game.isWinner("#") )
38         System.out.println("# is the winner!");
39     else if ( game.isFull() )
40         System.out.println("The game is a tie.");
41 }
42 }

```

---

This is a pretty simple terminal-based version of the game. All of the interesting code happens in DropGameHelper.

## What You Should See



```

|_|_|_|_|_|_|_|
|_|_|_|_|_|_|_|
|_|_|_|_|_|_|_|
|_|_|_|_|_|_|_|
|_|_|_|_|_|_|_|
|_|_|_|_|_|_|_|
|_|_|_|_|_|_|_|
|0|1|2|3|4|5|6|

'O', choose your column: 0
|_|_|_|_|_|_|_|
|_|_|_|_|_|_|_|
|_|_|_|_|_|_|_|
|_|_|_|_|_|_|_|
|_|_|_|_|_|_|_|
|_|_|_|_|_|_|_|
|_|_|_|_|_|_|_|
|0|_|_|_|_|_|_|
|0|1|2|3|4|5|6|

'#', choose your column: 3
|_|_|_|_|_|_|_| |
|_|_|_|_|_|_|_|
|_|_|_|_|_|_|_|
|_|_|_|_|_|_|_|
|_|_|_|_|_|_|_|
|_|_|_|_|_|_|_|
|0|_|_||#|_|_|_|
|0|1|2|3|4|5|6|

'O', choose your column: 3
|_|_|_|_|_|_|_| |
|_|_|_|_|_|_|_|
|_|_|_|_|_|_|_|
|_|_|_|_|_|_|_|
|_|_|_|_|_|_|_|
|_|_|_|0|_|_|_|
|0|_|_||#|_|_|_|
|0|1|2|3|4|5|6|

'#', choose your column:

```

We will revisit the DropGameHelper after we learn about graphics. Then we will make a graphical version of this game!



## Study Drills

1. If you want, rewrite the `isWinner()` method to actually check for four in a row horizontally or vertically like the real Connect Four game. (Mr. Mitchell's real-life students don't need to do this Study Drill to receive credit.)
2. Also add the ability to check for four in a row diagonally. This is pretty difficult. (Mr. Mitchell's real-life students don't need to do this Study Drill to receive credit.)

# Exercise 43: Abstract Classes and Final Methods

There are only two major concepts in object-oriented programming that we haven't looked at yet: abstract classes and packages.

An abstract class is a compromise between inheriting from a class and implementing an interface, with features from both.

AbstractDoor.java

---

```
1 public abstract class AbstractDoor {
2     protected boolean isOpen;
3
4     public AbstractDoor(boolean isOpen) { this.isOpen = isOpen; }
5     public void open() { isOpen = true; }
6     public void close() { isOpen = false; }
7
8     public abstract double soundProofing();
9
10    public final String toString() {
11        return "The door is " + (isOpen ? "open." : "closed.");
12    }
13 }
```

---

*AbstractDoor* is an abstract class. If a class is abstract, it can't be instantiated. You have to inherit from it and instantiate the subclass.

Abstract classes may have abstract *methods* as well. (But they aren't required to; if a class has any abstract methods then the class *must* be labeled abstract, but you can have an abstract class with no abstract methods if you want.)

The `soundProofing()` method is abstract so there is no implementation, just a method definition. This is similar to the methods in an interface: subclasses are required to provide an implementation.

I have also decided to make the `toString()` method `final`. (This doesn't have anything to do with abstract classes or methods; I just wanted to mention it in this chapter.)

A `final` method is sort-of like the opposite of an abstract method. Abstract methods *have* to be implemented in classes that extend this one; `final` methods can *not* be overridden in subclasses.

So here is an example of a subclass that doesn't do things correctly.

**BadDoor.java**

---

```
1  // DOES NOT COMPILE
2  public class BadDoor extends AbstractDoor {
3      public BadDoor(boolean b) { super(b); }
4
5      @Override
6      public String toString() {
7          return "The bad door is " + (isOpen ? "open." : "closed.");
8      }
9  }
```

---

The *BadDoor* class has two problems. First: it extends an abstract class but *fails* to provide an implementation for the abstract method `soundProofing()`. The compiler will complain about this.

Secondly, it tries to override the `toString()` method even though that method was marked `final` in the superclass. The compiler will complain about this, too.

Now let's look at a class that does it correctly.

**GoodDoor.java**

---

```
1  public class GoodDoor extends AbstractDoor {
2
3      public GoodDoor(boolean b) { super(b); }
4
5      public double soundProofing() {
6          if ( isOpen )
7              return 0;
8          else
9              return 8.5;
10     }
11 }
```

---

The *GoodDoor* class does everything right. It implements the abstract method and doesn't try to override any final methods. That's a good door.

## DoorDriver.java

```
1 public class DoorDriver {  
2     public static void main(String[] args) {  
3         // AbstractDoor d1 = new AbstractDoor(true); // CAN'T BE INSTANTIATED  
4         // BadDoor d2 = new BadDoor(true); // WON'T COMPILE  
5  
6         GoodDoor d3 = new GoodDoor(false);  
7         System.out.println(d3);  
8         System.out.println("\tSound-proofing level: " + d3.soundProofing());  
9         d3.open();  
10        System.out.println(d3);  
11        System.out.println("\tSound-proofing level: " + d3.soundProofing());  
12  
13        AbstractDoor d4 = new GoodDoor(true);  
14        System.out.println(d4);  
15        System.out.println("\tSound-proofing level: " + d4.soundProofing());  
16    }  
17 }
```

Similar to an interface, abstract classes cannot be instantiated. However, it is legal to have a reference variable as we see here:

```
AbstractDoor d4 = new GoodDoor(true);
```

The variable *d4* holds a reference to any *AbstractDoor* object. Since the *GoodDoor* class extends *AbstractDoor*, then any *GoodDoor* object is *also* an *AbstractDoor* object. And so *d4* can hold a reference to it, just like with regular non-abstract superclasses.

## What You Should See

```
The door is closed.  
    Sound-proofing level: 8.5  
The door is open.  
    Sound-proofing level: 0.0  
The door is open.  
    Sound-proofing level: 0.0
```

Java has interfaces, so abstract methods and abstract classes aren't used too much. Some older languages (like C++) don't have anything like interfaces, so everything is handled with abstract classes instead. They would make a so-called “pure” abstract class where *all* of the methods are abstract where Java programmers would probably use an interface.

Oracle's [Java Tutorial on Abstract Methods](https://docs.oracle.com/javase/tutorial/java/landl/abstract.html)<sup>17</sup> has more details, but usually you use an abstract class if you “want to share code among several closely-related classes”, and you use an interface when you “expect that unrelated classes would implement your interface.”



## Study Drills

1. Remove the `@Override` annotation from the `BadDoor` class. Does it change the compiler error message? Answer in a comment.
2. Write a new class called `WeakDoor` that also extends `AbstractDoor`. Implement the abstract method however you like. Also add some code to the driver that instantiates your new class.

---

<sup>17</sup><https://docs.oracle.com/javase/tutorial/java/landl/abstract.html>

# Exercise 44: Packages

As we are exploring the last major O.O. concept in this book, let us think back on object-oriented programming.

Object-oriented programming is about giving you tools so that *large* programs are bug-free and easy to fit together in complicated ways. We don't just put 100,000 lines of code in a single function, we break our code into methods.

We don't just put *all* our variables in a big pile so every line of code has access to them; we explicitly pass them into methods as a parameter. Not every value gets *out* of a method, either – only the values we return.

We group related methods into larger chunks called `classes`, with some shared variables (fields).


And finally, we group related *classes* into even larger chunks called *packages*.

You have been *using* packages since the beginning, you just haven't been writing your own. Consider this import statement:


```
import java.util.ArrayList;
```

The class `ArrayList` is contained inside a package called “util”. That's all it means. Packages are optional in Java, but almost all large projects use them.

Before we get started typing code, create a new folder/directory inside your code folder. Call the new folder `zoo`. You can do this from the terminal using the `mkdir` command:

```
 mkdir zoo
```

Then, save the next two files into that folder. You can either do this from your text editor, or just save them in the usual place and *move* them in there from the terminal:

```
 mv Horse.java Llama.java zoo
```

(This will move the files `Horse.java` and `Llama.java` into the folder `zoo`, assuming `zoo` is a sub-folder of the current folder.)

Also remember you can move into and out of a folder in the terminal using the `cd` (change directory) command:



```
cd zoo
cd ..
```

(The `..` means to back out to the previous folder, probably `javahard2`.)

Okay, so type up the following code. Name it `Horse.java` but make sure it's in the `zoo` folder eventually.

`zoo/Horse.java`

---

```
1 package zoo;
2
3 public class Horse {
4     public String toString() { return "Horse"; }
5 }
```

---

This is a *very* simple class, but the package line at the beginning signifies that this class is part of a package named `zoo`. Which means that the code must be in a *folder* called “`zoo`”. The package line must be the very first line in the file, even above any import statements.

`zoo/Llama.java`

---

```
1 package zoo;
2
3 public class Llama {
4     public String toString() { return "Llama"; }
5 }
```

---

Here is a second class that is in the same package as `Horse`.

If you don't include a package line, then your class becomes part of the “unnamed” package. This is what we have been doing for the past two books. It is fine to have small simple programs in the unnamed package.

But large programs with many related classes should put them in a package.

Okay, this is the driver code; it is *not* saved inside the `zoo` folder. It is not part of the `zoo` package; it merely *uses* the classes from that package.



## ZooDriver.java

```
1 import zoo.Horse;
2 import zoo.Llama;
3
4 public class ZooDriver {
5     public static void main( String[] args ) {
6         Horse bj = new Horse();
7         Llama kuz = new Llama();
8
9         System.out.println( bj );
10        System.out.println( kuz );
11    }
12 }
```

Classes that are all part of the unnamed package can find each other when compiling. And classes that are in the *same* package can find each other.

But since `ZooDriver.java` is in the unnamed package and the other two files are in the `zoo` package, we have to import them.

```
import zoo.Horse;
```

...means “import the Horse class from inside the zoo package.” Simple.

## What You Should See

```
Horse
Llama
```

That’s all there is to it!

Most big companies use package names that are a reversed form of their Internet domain name. That is, `google.com` has Java Android classes in packages like so:

```
com.google.android.gms.auth
```

(You can see a list of all their [Android packages](https://developers.google.com/android/reference/packages)<sup>18</sup> if you want.)

<sup>18</sup><https://developers.google.com/android/reference/packages>

This means that their Java files would have to be in a folder called *com*, then inside another one called *google*, then one called *android*, etc.



## Study Drills

1. Save a copy of the driver class called `ZooDriver2.java`. Add the line `package zoo;` at the very top of the new file. What changes do you have to make so that it will compile and run correctly? Make the necessary changes, and explain what you did in some comments at the bottom of your new file.

# Exercise 45: Creating a JAR File

Java programs are often distributed in a package called a JAR file (Java ARchive). You wouldn't expect your customers to be able to open a terminal and compile and run your code there!

We are going to make a JAR file out of the files we wrote for the previous exercise, but you can make a JAR file out of *any* Java class, even if it doesn't use packages.

So there's no new code for this exercise!

Go into your terminal window and make sure the classes you want to use are compiled. This is because you only need to put the `.class` files in the JAR. (You can put the `.java` files in there too, but it's weird and they aren't needed.)



```
javac ZooDriver.java
```

After this command, there should be a file called `ZooDriver.class` in the current folder that contains the bytecode version of your program. There should also be files called `Horse.class` and `Llama.class` inside the `zoo` folder.

We are going to create a JAR file called `Zoo.jar` using the `jar` command. If you can run the `javac` and `java` commands you ought to be able to run the `jar` command in the same way.

Type the following in your terminal window:



```
jar cvfe Zoo.jar ZooDriver ZooDriver.class zoo/*.class
```

## What You Should See

```
added manifest
adding: ZooDriver.class(in = 480) (out= 327)(deflated 31%)
adding: zoo/Horse.class(in = 274) (out= 202)(deflated 26%)
adding: zoo/Llama.class(in = 274) (out= 205)(deflated 25%)
```

If you are an experienced Unix user familiar with the “tar” command, what you typed might seem a little familiar. But for everyone else...

You are passing in four arguments to the *jar* command: *c*, *v*, *f* and *e*. The *c* means “create” because we are trying to create a new JAR. The *v* means to be “verbose” and to print on the screen what is going on. You can leave out the *v* and it will work silently instead.

The *f* means you want to supply the “filename” of the JAR in this command; it will be the first thing after the space.

JAR files have a special text file in them called a “manifest”. You can create the manifest yourself and pass it in to the *jar* command, or you can just tell the *jar* command which Java class is the “entry point” and let it create the manifest for you. So *e* means that we are supplying the entry point only.



```
jar cvfe Zoo.jar ZooDriver ZooDriver.class zoo/*.class
```

So, *jar* command with options create, verbose, filename and entriypoint. After that is the name of the JAR file to create: *Zoo.jar*. Which should now exist in the current folder. You can confirm by typing the *ls* command in the terminal.

After the output filename is *ZooDriver*, the “entry point” class to put in the manifest. This is the class containing the *main()* method.

We could have created a manifest file ourselves. The manifest file is typically called *manifest.mf*, but it is a bit easier to create and edit if you call it *manifest.txt*. For this JAR, the manifest would have looked like this:

**manifest.txt**

---

Main-Class: ZooDriver

---

(It’s just one line, but it has to have a newline at the end, so hit ENTER at least once before you save it.)

So if you wanted to provide your own manifest file like this, the *jar* command would look like so:




```
jar cvfm Zoo.jar manifest.txt ZooDriver.class zoo/*.class
```




It doesn’t matter what you call the manifest file because the *jar* utility just reads the information from it and then creates its own file called *MANIFEST.MF* and puts it into a folder called *META-INF* in the archive file.

One more detail: we list the name of the JAR file first and the manifest or entry point second because the `f` option is listed before the `m` option. We could have also done this:




```
jar cvmf manifest.txt Zoo.jar ZooDriver.class zoo/*.class
```

or this:



```
jar cvef ZooDriver Zoo.jar ZooDriver.class zoo/*.class
```


Finally, after the names of the archive and entry point you just list all the names of the `.class` files you want to put into the JAR. In this case, we are putting in `ZooDriver.class` from the current folder and all the class files from the `zoo` folder. The star/asterisk (`*`) is a wildcard meaning “everything ending in `.class`.” You could also have named them all explicitly:



```
jar cvfe Zoo.jar ZooDriver ZooDriver.class zoo/Horse.class zoo/Llama.class
```

Whew.

So now you have created the JAR file! You can run it from the terminal like so:



```
java -jar Zoo.jar
```

## What You Should See



```
Horse  
Llama
```

JAR files can also be double-clicked to run them as long as the Java runtime (JRE) is installed. However, if your JAR only displays stuff in a terminal window (like all of our programs so far), double-clicking the JAR file probably won’t automatically open up a terminal, so they won’t be able to see anything.

Once we start learning how to make programs with a graphical user interface (GUI) in the next exercise, JAR files will make more sense.

If you are on Linux (and probably a Mac), you also won't be able to double-click the JAR file to run it until that file is made "executable". For safety, these operating systems won't just execute any random file. On my machine, you can make a file executable in a graphical way (right-click the file and choose "Properties" then the "Permissions" tab) or you can do it from the terminal like so:



```
chmod a+x Zoo.jar
```

This is the "change mode" command, and it should work like this on Linux or on a Mac. I don't actually know how to do it from Powershell on Windows! Sorry! If any reader can figure it out on Windows, email me and I'll add the instructions to a future edition of the book.

Even if you have heard of JAR files before today you probably didn't know that JAR files are just ZIP files with a little extra information in them and a different file extension. It's true! Make a copy of your JAR file, but with a different extension:



```
cp Zoo.jar Zoo.zip
```

(Make a copy of `Zoo.jar` called `Zoo.zip`.)

Then graphically browse to the current folder, and double-clicking on it should now open the archive to show its contents. Mine contains a compressed version of `ZooDriver.class`, a folder called `zoo` with two more compressed class files in it, and a folder called `META-INF` containing the manifest file.

There's a lot more you can do with JAR files, including digital signatures and other cool things. You can read up on them in Oracle's Java Tutorial for [Packaging Programs in JAR Files](https://docs.oracle.com/javase/tutorial/deployment/jar/)<sup>19</sup>.

JAR files are nice because they allow you to distribute a single file with all your classes and resources in it. For years Minecraft was distributed as a JAR file!



## Study Drills

1. Create a JAR file from a previous exercise. (Make sure to pick one with more than one class.) Test running it with `java -jar Whatever.jar`.

---

<sup>19</sup><https://docs.oracle.com/javase/tutorial/deployment/jar/>

# Exercise 46: A Simple Graphical Window

Unless your friends are weird, most of them probably do not use a computer by typing commands into a terminal window. They probably use a some sort of graphical user interface (or GUI, pronounced “gooey”) like 99.99% of humanity.

Making GUI programs is much easier in Java than in some other programming languages, but it still requires a pretty good understanding of object-oriented programming techniques, so I couldn’t explain it before now.

But now is the time!

SimpleWindow.java

---

```
1  import javafx.application.Application;
2  import javafx.scene.Group;
3  import javafx.scene.Scene;
4  import javafx.stage.Stage;
5
6  public class SimpleWindow extends Application {
7      @Override
8      public void start(Stage stage) {
9          Group root = new Group();
10         Scene scene = new Scene(root, 600, 400);
11
12         stage.setTitle("Simple Window");
13         stage.setScene(scene);
14         stage.show();
15     }
16
17     public static void main( String[] args ) {
18         launch(args);
19     }
20 }
```

---

Assuming you have version 1.8 of the JDK installed, this program should compile and run just fine. If you are using an older version of Java, you’ll have to figure out how to install “JavaFX”.

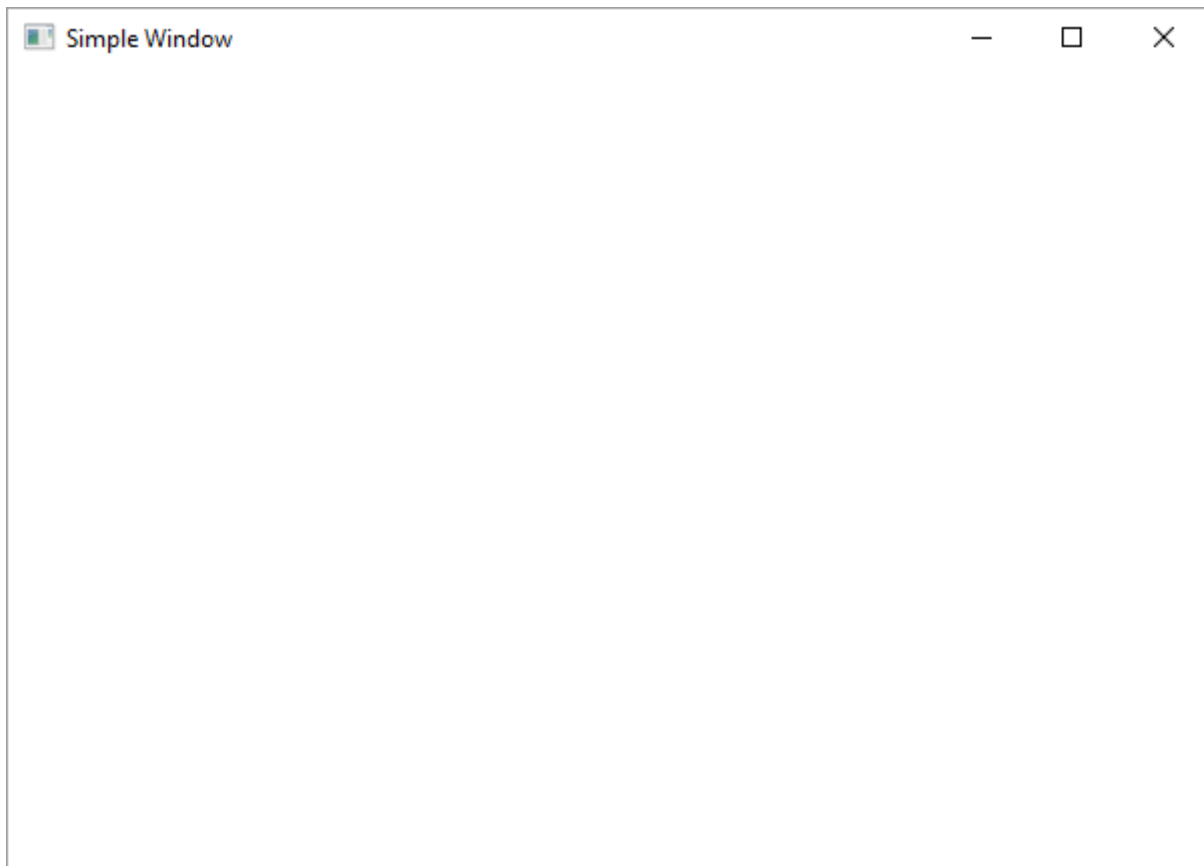
If, like me, you are using the OpenJDK version of Java on Linux, you will have to install JavaFX separately:



```
sudo apt-get install openjfx
```

If you're using Oracle's release of the JDK, though, this should just work.

## What You Should See



Windows 10 screenshot of SimpleWindow

If you're using Windows 10, your (boring) window will probably look a lot like the above screenshot. I typically do work on a PC running Ubuntu Linux, so when I ran it on my machine, it looked more like this:





Linux screenshot of SimpleWindow

I don't have access to a Mac to get a screenshot, but it shouldn't matter. The whole *idea* of JavaFX is that your graphical programs will look pretty much the same on different operating systems *and* they will look like native apps for that platform as much as possible.

Java has had a GUI library since the very first version of Java in 1995. Originally they used AWT (the Abstract Window Toolkit). It was better than nothing, but it had some problems.

Then, in 1998 when version 1.2 of Java was released, a new GUI library called "Swing" was introduced. This was a lot better than AWT, but still not incredibly great.

The latest and greatest GUI library from Oracle is called JavaFX, and it first appeared in Java version 1.7 in late 2011. It used to be a separate download, though.

JavaFX is now the recommended way of doing graphics in a Java program, and it is installed by default starting with version 1.8 of Java in 2014. JavaFX is pretty nice, so that's what I'm going to show you.

A word of warning, however. I don't do much GUI programming at all, and JavaFX is still pretty new to me. So I'll do my best to show you the basic ideas, and you can learn the rest of the details

on your own. Oracle's [JavaFX Tutorials](http://docs.oracle.com/javase/8/javafx/get-started-tutorial/jfx-overview.htm)<sup>20</sup> are pretty good if you know OOP really well!

Okay, let's talk about the code you just wrote!

Every JavaFX application starts as a subclass of `javafx.application.Application`. The method called `start()` is where things begin and is required. JavaFX creates a basic window called a "Stage" and passes a reference to it into the `start()` method.

Inside the Stage object there is a Scene object. The Scene is a container for everything else in the window. Inside the scene is a much more complicated linked (tree or graph) structure called the "scene graph". This program has an extremely simple scene graph: just a single node called the "root".

Every scene graph must have a root node. In this exercise, I just used a container node called `Group`, but there's nothing inside the group.



Every JavaFX application needs three things: a stage, a scene, and a scene graph with at least one node. (Don't worry, that doesn't make much sense to me, either.)

The root node of the scene graph is a container that holds the whole rest of the user interface (UI) elements.

In our code, we are given a reference to the Stage window. We instantiate a new `Group` object. Then we instantiate a `Scene` object and tell it that the `Group` object we just made is going to be the root of the scene graph. We also tell it to make the window have a default size of 600 pixels wide by 400 pixels tall.

Then, we should set the title of the window/stage. We need to give the Stage object a reference to its Scene. And finally we ask the window to show itself. (Otherwise the window will exist but it will be invisible.)

JavaFX programs don't *have* to have a `main()` method. You can create them using something called the "JavaFX Packager tool". But if you're going to just be running them from the terminal like this, you need a `main()` method, and all it has to do is `launch()`, a static method inherited from `Application`.

This window is pretty simple, but it has a title and it has a size. You can move the window around, change its size, minimize it, etc.

And more importantly, this is the basic "shell" that all JavaFX applications will have. It is okay if you don't really understand what all this code does or how it works, because it will pretty much be the same in every one.

(The only real difference will be what sort of container we pick for the root node of our scene graph.)

---

<sup>20</sup><http://docs.oracle.com/javase/8/javafx/get-started-tutorial/jfx-overview.htm>



## Study Drills

1. Change the title of the stage. Recompile and make sure it works.
2. Create a JAR file out of this class. Make it executable if you need to, then double-click the JAR file! It ought to work! You can send this single JAR file with no other files to a friend, and as long as they have Java installed (only the Java runtime/JRE is needed) they should be able to run it!

# Exercise 47: An Interactive Window

In this exercise we are going to look at a more complicated GUI window that actually *does* something. It will have a box where you can type some text and a button. When you click the button, it will display the text in the terminal.

I apologize in advance for all the imports at the top. We are finally to the point in our learning where it would probably make sense to start using a proper Integrated Development Environment (IDE) that could auto-complete some of those imports for you. (If you're going to go that route, I recommend just the Java SE bundle of the Netbeans IDE.)

Alternatively, you could just use some wildcards to import entire packages to save you some typing:

```
import javafx.application.Application;
import javafx.event.*;
import javafx.geometry.*;
import javafx.scene.*;
import javafx.scene.control.*;
import javafx.scene.layout.*;
```

Anyway, when you have a lot of import statements, most organizations like them to be in alphabetical order, which I have tried to do in these GUI exercises.

MessengerWindow.java

---

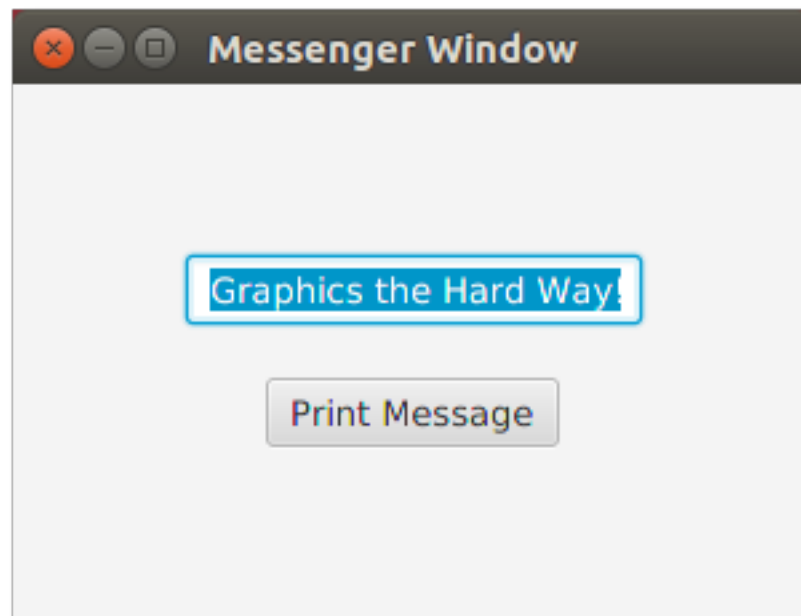
```
1  import javafx.application.Application;
2  import javafx.event.ActionEvent;
3  import javafx.event.EventHandler;
4  import javafx.geometry.HPos;
5  import javafx.geometry.Pos;
6  import javafx.scene.Scene;
7  import javafx.scene.control.Button;
8  import javafx.scene.control.TextField;
9  import javafx.scene.layout.GridPane;
10 import javafx.stage.Stage;
11
12 public class MessengerWindow extends Application {
13     @Override
14     public void start(Stage stage) {
15         GridPane root = new GridPane();
```

```
16     Scene scene = new Scene(root, 300, 200);
17
18     TextField tf = new TextField("Graphics the Hard Way!");
19     Button btn = new Button("Print Message");
20
21     btn.setOnAction(new EventHandler<ActionEvent>() {
22         @Override
23         public void handle(ActionEvent event) {
24             System.out.println( tf.getText() );
25         }
26     });
27
28     root.setAlignment(Pos.CENTER);
29     root.setVgap(20);
30
31     root.add(tf, 0, 0);
32     root.add(btn, 0, 1);
33
34     root.setHalignment(btn, HPos.CENTER);
35
36     stage.setTitle("Messenger Window");
37     stage.setScene(scene);
38     stage.show();
39 }
40
41 public static void main( String[] args ) {
42     launch(args);
43 }
44 }
```

---

Just like last time, this application has a stage, a scene, and at least one node to serve as the root/container for our scene graph. In this case our root node will be a `GridPane`, which makes it easy to arrange components in rows and columns.

## What You Should See



Linux screenshot of MessengerWindow

After setting up the GridPane and scene, we create two “controls”, a TextField and a Button. Then we set up what is called an event handler for the button.

JavaFX has a lot of built-in controls, including checkboxes, date pickers, progress bars, and dialog boxes. Most of them are in `javafx.scene.control`, but things that pop up a whole new “window” are technically stages of their own so – for example – `FileChooser` is in `javafx.stage` instead.

So I have used an anonymous class in setting up the event handler, since that’s the way most Java programmers do it. You haven’t seen one of these before. It starts like this:

```
btn.setOnAction(new EventHandler<ActionEvent>() {
```

Notice that at the end of this line, we still have a curly brace that needs closing and a parenthesis that needs closing. You can see them several lines later, including the semicolon to end the `setOnAction()` statement!

The “proper” way to do this, which no one actually does, would be to create a whole new class that implements the `EventHandler` interface, like so:

```
// DON'T TYPE THIS
import javafx.event.*;
import javafx.scene.control.*;
public class ButtonClickyOne implements EventHandler<ActionEvent> {
    private TextField tf;
    public ButtonClickyOne( TextField tf ) { this.tf = tf; }
    public void handle(ActionEvent event) {
        System.out.println( tf.getText() );
    }
}
```

...and then you just instantiate this new class and pass it in to `setOnAction()`:

```
ButtonClickyOne bc1 = new ButtonClickyOne(tf);
btn.setOnAction(bc1);
```

Notice that we have to add a parameter to the constructor so that we can pass in a reference to the `TextField` object so that the handler can access it!

This is really annoying to do dozens of times in a large GUI application, so Java allows you to just create an *anonymous* class like so:

```
new EventHandler<ActionEvent>() { ... }
```

All the code that should be inside the new class just lives inside the curly braces, and the class is *inside* the current method so it gets access to all local variables of that method without having to pass them in somehow.

If you look in the code folder after compiling, you'll see that in addition to `MessengerWindow.class` there is *also* a bytecode file called `MessengerWindow$1.class`. That's the bytecode for the anonymous class!

Anyway, the `EventHandler` interface only has a single method that we have to implement. It's called `handle()`. In this case we just call the `getText()` method of the `TextField` object and print out the `String` that it gives us.

The rest of the code in the `start()` method is making the `GridPane` look like we want. We want the controls inside the pane to be centered horizontally and vertically, so we call `setAlignment()` on the `GridPane` object. We also want the controls to have a 20-pixel vertical gap between them.

We then add the `TextField` object to the grid at column 0, row 0 and add the `Button` to the grid at column 0, row 1.

Finally we ask the root node to set the horizontal alignment of the button object to be centered. (Otherwise the button aligns left.)

And then, just like before, we set the title of the window, attach the scene containing our scene graph to the stage, and make it visible.

Graphical programs get pretty complicated pretty quickly, which is why I don't do them with beginners. In the next exercise, I will show you a different way to set up a user interface in JavaFX.



## Study Drills

1. See if you can figure out how to change the code to use a `java.io.PrintWriter` object to output the messages to a text file instead of just printing them on the screen. (Mr. Mitchell's real-life students don't need to do this Study Drill to receive credit.)



# Exercise 48: Using (F)XML to Define Your Interface

It doesn't usually make sense to write Java code to define the look of your interface. Do I really need to recompile just because I want the button to be centered instead of aligned to the left?

So, like most other graphics libraries, JavaFX includes a way to describe the layout of the user interface in a separate text file that can be created by hand or using a tool. Then when the application runs it loads the file to set up its interface.

Java uses the XML markup language for this, and calls its flavor FXML. Here is a very similar program to the previous exercise, but using FXML instead of Java code to set up the interface.

MessengerFXML.java

---

```
1  import javafx.application.Application;
2  import javafx.fxml.FXMLLoader;
3  import javafx.scene.Parent;
4  import javafx.scene.Scene;
5  import javafx.stage.Stage;
6
7  public class MessengerFXML extends Application {
8      @Override
9      public void start(Stage stage) throws Exception {
10         Parent root = FXMLLoader.load(getClass().getResource("messenger.fxml"));
11         Scene scene = new Scene(root, 300, 200);
12
13         stage.setTitle("Messenger FXML");
14         stage.setScene(scene);
15         stage.show();
16     }
17
18     public static void main( String[] args ) {
19         launch(args);
20     }
21 }
```

---

The main class is much simpler than before, because the entire scene graph is loaded from an FXML file instead! There's not much to this code, honestly.

## MessengerFXMLController.java

---

```

1  import javafx.event.ActionEvent;
2  import javafx.fxml.FXML;
3  import javafx.scene.control.Alert;
4  import javafx.scene.control.TextField;
5
6  public class MessengerFXMLController {
7      @FXML private TextField tf;
8
9      @FXML protected void handleButtonClickOrWhatever(ActionEvent event) {
10         Alert alert = new Alert(Alert.AlertType.INFORMATION, tf.getText());
11         alert.showAndWait();
12     }
13 }

```

---

Well, we can't use an anonymous class anymore, so here is the class that holds the method to run when they click on the Button. The `@FXML` annotation marks variables or methods you want to be able to reference in your FXML file.

Notice that this class can be anything; it does *not* have to implement the `EventHandler` interface. And since we're not implementing a specific method for an interface, our handler method can be called whatever we want.

If we had ten buttons, we could put all ten handler methods in this same class. We could even put them in the original class, but separating them a bit is a little cleaner.

## messenger.fxml

---

```

1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <?import javafx.scene.layout.GridPane?>
4  <?import javafx.scene.control.TextField?>
5  <?import javafx.scene.control.Button?>
6
7  <GridPane fx:controller="MessengerFXMLController"
8      xmlns:fx="http://javafx.com/fxml" alignment="center" vgap="20">
9
10     <TextField fx:id="tf"
11         text="Graphics the FXML Way!"
12         GridPane.columnIndex="0" GridPane.rowIndex="0"/>
13
14     <Button text="Print Message"
15         GridPane.columnIndex="0" GridPane.rowIndex="1"

```

```
16         GridPane.halignment="CENTER"
17         onAction="#handleButtonClickOrWhatever"/>
18     </GridPane>
```

---

Here is our FXML file. I have called it `messenger.fxml` but you could call it whatever you want as long as the name matches what you loaded at the beginning of the `start()` method in the main application.

The first line shows that this is just an XML file. The next few lines are Java import statements, but with `<?` at the beginning and `?>` at the end instead of semicolons.

We didn't have to import anything in this file. We could have just written `javafx.scene.control.TextField` on line 10 instead of just `TextField`, just like you can in a Java program.

You can see that our scene graph is contained inside a `GridPane` object just as before. Except now there's an opening XML `<GridPane>` tag and later a closing XML `</GridPane>` tag. And our two controls are defined in between.

```
<GridPane attributes>
    other tags
</GridPane>
```

The `GridPane` tag has a required attribute `xmlns:fx` that gives the XML namespace being used for the file. There's also an attribute called `fx:controller` that gives the name of the class that will have our handlers in it.

Inside the opening `GridPane` tag we have also defined the alignment and the vertical gap between rows with two more attributes.

The `TextField` tag isn't a container like the `GridPane` tag is, so there isn't an opening and closing tag. There's just a single tag which is self-closing:

```
<TextField attributes attributes attributes />
```

One of the attributes inside the `TextField` tag is `fx:id` which connects this `TextField` description to a `TextField` variable in our controller. Remember how in our controller file we had a line like this?

```
@FXML private TextField tf;
```

And in our FXML file we have an attribute like this:

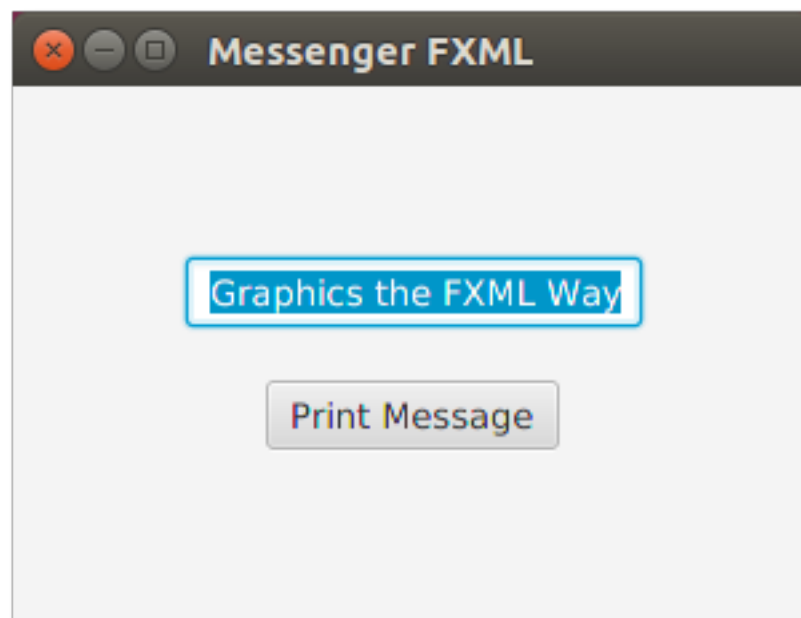
```
<TextField fx:id="tf">
```

This is how the program knows that our private `TextField` variable in `MessengerFXMLController.java` is the same as the `TextField` object that is in column 0 and row 0 in `messenger.fxml`.

The other attributes inside the `TextField` self-closing tag set the default text and the column and row inside the `GridPane`.

The only other tag inside the FXML file is our `Button` tag. There's no variable in the code this needs to be tied to, so there's no `fx:id` attribute in this one. There is an `onAction` attribute, though, which gives the name of the method in the controller to run when this button is clicked. No anonymous inner class needed!

## What You Should See



Linux screenshot of MessengerFXML

As you can see, this looks pretty much the same as the previous exercise when you run it! Which is the point. User interfaces in JavaFX can be defined entirely in FXML.

To be honest, most developers don't hand-create FXML files like this, either. They use an external program that writes the FXML file for them, like Oracle's JavaFX Scene Builder tool. The details of FXML and such tools are beyond the scope of this book, but I wanted to at least show you how such a thing is done!



## Study Drills

1. Change the FXML file so that the `TextField` is in row 1 and the `Button` is in row 0. Notice how you don't even have to recompile your code!
2. See if you can figure out how to add a `javafx.scene.control.Label` above the `TextField` that says "Type your message". Make it aligned to the right. You should only have to change the FXML file and not the Java code.

# Exercise 49: Canvas Basics

Most graphical programs written in Java are user interfaces, with buttons and fields and such. But usually my high-school students don't care anything about graphical programs like that. They only care about making video games!

And you *can* make video games in Java. In fact, Minecraft (one of the best-selling indie video games of all time) is written in Java. Games tend to want very detailed control over what gets drawn on the screen and when, so they tend to just use a “canvas” and do everything manually.

So the next few exercises will look at drawing on the canvas in a JavaFX program.

CanvasBasics.java

---

```
1  import javafx.application.Application;
2  import javafx.scene.Group;
3  import javafx.scene.Scene;
4  import javafx.scene.canvas.Canvas;
5  import javafx.scene.canvas.GraphicsContext;
6  import javafx.scene.paint.Color;
7  import javafx.stage.Stage;
8
9  public class CanvasBasics extends Application {
10     @Override
11     public void start(Stage stage) {
12         Group root = new Group();
13         Scene scene = new Scene(root);
14
15         Canvas canvas = new Canvas(800,600);
16         GraphicsContext gc = canvas.getGraphicsContext2D();
17
18         gc.setStroke(Color.GREEN);
19         gc.setFill(Color.GREEN);
20         gc.strokeRect(50, 20, 100, 200);
21         gc.fillOval(160, 20, 100, 200);
22         gc.setFill(Color.BLUE);
23         gc.fillRect(200, 400, 200, 20);
24         gc.strokeOval(200, 430, 200, 100);
25
26         gc.setFill(Color.BLACK);
27         gc.fillText("Graphics are pretty neat.", 500, 100);
```

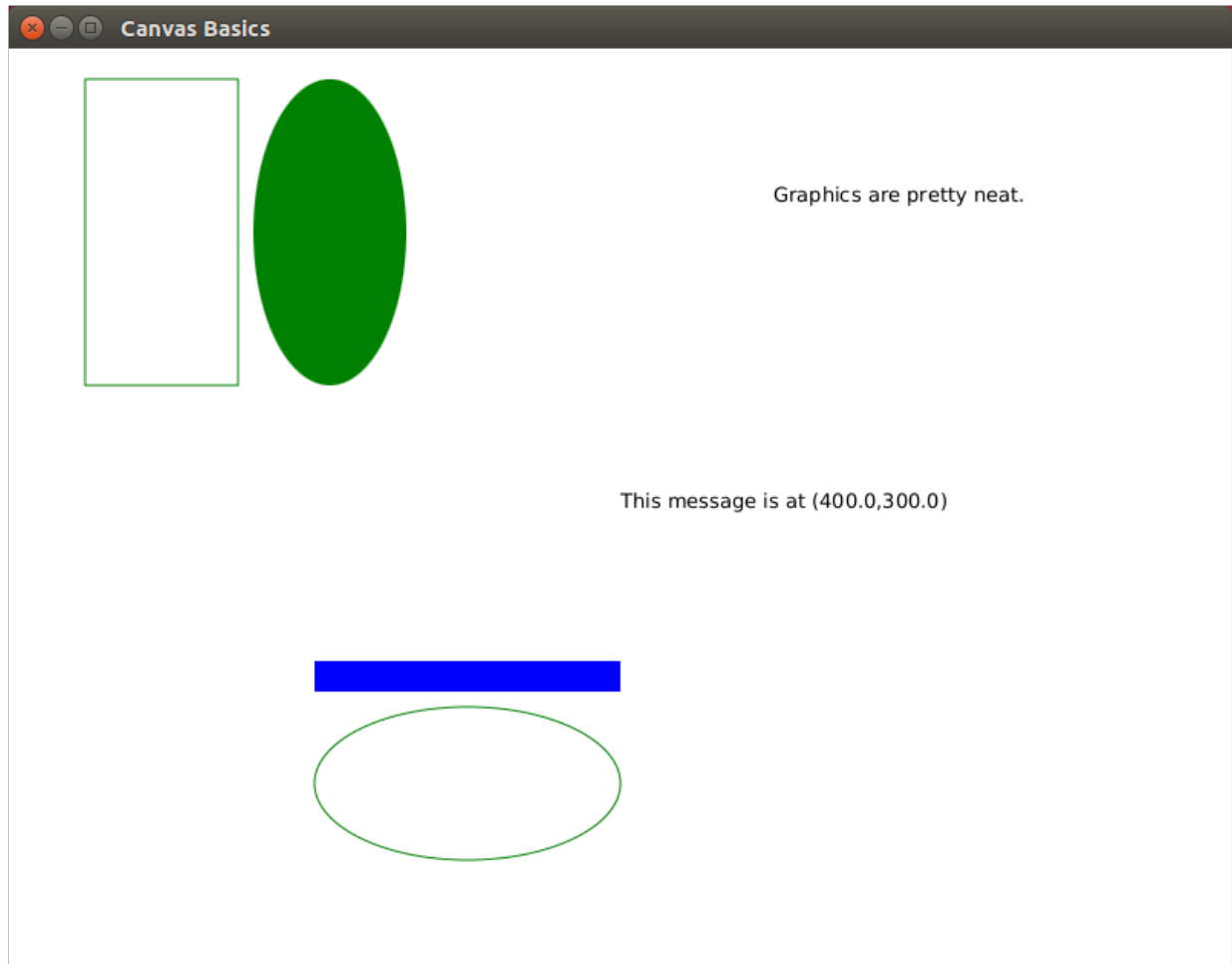
```
28
29     double x = canvas.getWidth() / 2;
30     double y = canvas.getHeight() / 2;
31     gc.fillText("This message is at (" + x + "," + y + ")", x, y);
32
33     root.getChildren().add(canvas);
34
35     stage.setTitle("Canvas Basics");
36     stage.setScene(scene);
37     stage.show();
38 }
39
40 public static void main( String[] args ) {
41     launch(args);
42 }
43 }
```

---

At the very beginning of the `start()` method we do something a *little* different. Instead of telling the Scene how big to make the window, we tell the Canvas object how large to be and then let the Scene resize itself to fit the Canvas.

Also, instead of using a `GridPane` we are back to just using a boring `Group` as the root of our scene graph. In fact, the scene graph will have nothing in it except the root node and the canvas.

## What You Should See



Linux screenshot of CanvasBasics

The Canvas is what we are going to draw on, but the `GraphicsContext` object has all the methods in it. (This is similar to the way other programming languages do it.)

So we tell the `GraphicsContext` that whenever it draws the stroke of a line on the canvas to use the color green (`setStroke()`). Also, whenever it fills in a shape on the canvas, it should *also* use green (`setFill()`).

We begin by drawing an outline-only (stroked) rectangle on the canvas: 50 pixels from the left side, 20 pixels from the top and with a width of 100 pixels and a height of 200 pixels.

Almost every 2-D graphics library defines the point (0,0) to be in the upper-left of the screen. X-values get bigger as you go toward the right and Y-values get bigger as you go toward the bottom of the screen. And there are no negative values.



Next we draw a filled-in oval. The oval will touch the sides of a box whose upper-left corner is at the point (160, 20), and the box (and oval) is 100x200 pixels. You can see that the oval and the rectangle we drew earlier are the same size, and the oval starts 110 pixels to the right of the rectangle.

We change the fill color to blue and draw a filled-in 200x20 rectangle with an upper-left-hand corner at the point (200, 400). Then we draw an outline-only oval just below the rectangle. We never changed the GraphicsContext's *stroke* property, so the outline oval is still green.

Then we change the fill color to black and display the words "Graphics are pretty neat" on the canvas. The message will start at the point (500, 100).

Finally we compute the center point of the canvas and draw a different message starting there.

Now, none of this will show up unless you add the Canvas object to the scene graph. And we still have to tell the Stage that this scene belongs to it and make the window/stage visible.

That maybe isn't the most interesting graphics program in the world, but you'll need to understand how the GraphicsContext object draws onto the Canvas to be able to keep up in the next few exercises!



## Study Drills

1. Add code to draw a filled-in red square in the lower-right-hand corner of the canvas.

# Exercise 50: Getting Mouse Input

Since we are using a canvas, we can't just rely on a Button's event handler. We need to do two things differently: we must handle the mouse click event ourselves, and we must manually set up a timer to redraw the canvas several times a second so that changes we make will be redrawn.

In a normal JavaFX GUI, changes to the scene graph are magically redrawn when necessary. (We'll see an example of this in the next exercise). But if you are going to use a Canvas object you have to take care of that yourself.

Again, sorry about all the imports. I would copy-and-paste them if I were you.

MouseDemo.java

---

```
1  import javafx.animation.AnimationTimer;
2  import javafx.application.Application;
3  import javafx.event.EventHandler;
4  import javafx.scene.Group;
5  import javafx.scene.Scene;
6  import javafx.scene.canvas.Canvas;
7  import javafx.scene.canvas.GraphicsContext;
8  import javafx.scene.input.MouseEvent;
9  import javafx.scene.paint.Color;
10 import javafx.scene.text.Font;
11 import javafx.stage.Stage;
12
13 public class MouseDemo extends Application {
14     private Color curColor;
15     private String curMessage;
16
17     @Override
18     public void start(Stage stage) {
19         Group root = new Group();
20         Scene scene = new Scene(root);
21
22         curColor = Color.RED;
23         curMessage = "The square is red.";
24
25         Canvas canvas = new Canvas(800,400);
26         GraphicsContext gc = canvas.getGraphicsContext2D();
27         gc.setFont(Font.font(24));
```

```

28
29     canvas.addEventHandler(MouseEvent.MOUSE_CLICKED,
30         new EventHandler<MouseEvent>() {
31             @Override
32             public void handle(MouseEvent t) {
33                 curColor = Color.GREEN;
34                 curMessage = "The square is green.";
35             }
36         });
37
38
39     root.getChildren().add(canvas);
40
41     stage.setTitle("Mouse Demo");
42     stage.setScene(scene);
43     stage.show();
44
45     new AnimationTimer() {
46         @Override
47         public void handle(long now) {
48             gc.setFill(curColor);
49             gc.fillRect(150, 100, 100, 100);
50             gc.setFill(Color.WHITE);
51             gc.fillRect(295, 78, 260, 30);
52             gc.setFill(Color.BLACK);
53             gc.fillText(curMessage, 300, 100);
54         }
55     }.start();
56 }
57
58 public static void main( String[] args ) {
59     launch(args);
60 }
61 }

```

---

In the first MessengerWindow exercise, I told you that if you use an anonymous inner class, you can reference local variables without having to pass them in. That's true, but you only get *read-only* access to those variables that way. If you're going to *modify* the variables they can't be *local* variables. But instance variables are fine.

So we have defined two private fields for our Application object so that the anonymous inner classes in our handlers can read and change them.

Once we get inside the `start()` method, we set the initial values for our `Color` variable and our message. We create a `Canvas` object, set the font size to 24pt, then add an `EventHandler` to that canvas object to handle any mouse clicks.

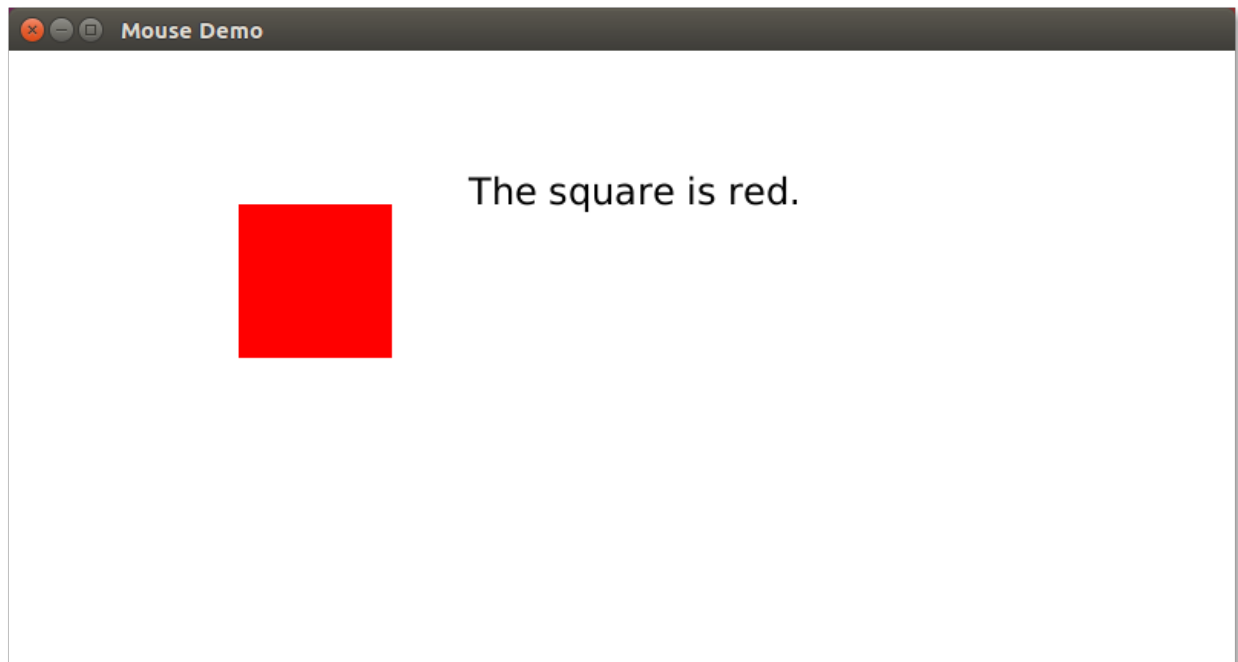
If the user clicks the mouse anywhere in the canvas, the handler changes the color to green and changes the message.

That's it for our initial set up. We go ahead and add the canvas object to the scene graph and show the stage as usual.

Then we create a second anonymous class to create an anonymous `AnimationTimer` object, and call `start()` on it. The `AnimationTimer` will call its `handle()` method approximately 60 times per second. (The parameter *now* contains a increasing number of nanoseconds that can be useful in figuring out *exactly* how long it has been since the previous call to `handle()`. This is needed for smooth animation in complex programs.)

The handler draws a rectangle in the current color and displays the current message on the screen. However, the canvas is *not* cleared between calls to the `handle()` method, so we have to manually “erase” the previous message by painting over it with a white rectangle. Otherwise when the message changes it will overwrite the earlier message and be unreadable.

## What You Should See



Linux screenshot of MouseDemo

This is a weird design for a program, but it demonstrates a very common way of writing video games with a main game loop that runs many times per second.

We will take a detour in the next exercise to see how if you *don't* use a Canvas, changes to the scene graph are automatically redrawn. But then we will be back to do more animation with a canvas.



## Study Drills

1. Make it so that when you click the mouse more than once, the message and the color change back and forth between green and red every time you click.

# Exercise 51: More Complex Mouse Interaction

We are back to using the scene graph in this exercise and not drawing everything manually on a Canvas, so there's no need for an AnimationTimer. We are also going to be using Java's built-in Circle objects.

TrafficLight.java

---

```
1  import javafx.application.Application;
2  import javafx.event.EventHandler;
3  import javafx.scene.Group;
4  import javafx.scene.Scene;
5  import javafx.scene.control.Label;
6  import javafx.scene.input.MouseEvent;
7  import javafx.scene.paint.Color;
8  import javafx.scene.shape.Circle;
9  import javafx.stage.Stage;
10
11 public class TrafficLight extends Application {
12     @Override
13     public void start(Stage stage) {
14         Group root = new Group();
15         Scene scene = new Scene(root, 300, 600);
16
17         Label label = new Label("Click on one of the circles!");
18         label.relocate(20, 10);
19
20         Circle circle1 = new Circle(150, 140, 75);
21         circle1.setFill(Color.RED);
22         Circle circle2 = new Circle(150, 310, 75);
23         circle2.setFill(Color.YELLOW);
24         Circle circle3 = new Circle(150, 480, 75);
25         circle3.setFill(Color.web("#008000"));
26
27         scene.addEventHandler(MouseEvent.MOUSE_CLICKED,
28             new EventHandler<MouseEvent>() {
29             @Override
30             public void handle(MouseEvent t) {
```

```

31         if ( circle1.contains(t.getX(), t.getY()) )
32             label.setText("You clicked on the red circle!");
33         else if ( circle2.contains(t.getX(), t.getY()) )
34             label.setText("You clicked on the yellow circle!");
35         else if ( circle3.contains(t.getX(), t.getY()) )
36             label.setText("You clicked on the green circle!");
37         else
38             label.setText("Click at " + t.getX() + ", " + t.getY());
39     }
40 });
41
42 root.getChildren().addAll(label, circle1, circle2, circle3);
43
44 stage.setTitle("Traffic Light");
45 stage.setScene(scene);
46 stage.show();
47 }
48
49 public static void main( String[] args ) { launch(args); }
50 }

```

---

In an earlier exercise we used a `GridPanel` as the root node of our scene graph. We only had to tell it which row and column to put the controls in, and it managed the locations of everything.

This time we are just using a generic `Group` node as our root, so we are going to manually specify the positions of all of our elements within that group.

The `Label` object just displays text. Unlike the `TextField`, the text can't be edited by the user, but we can still change it. The label would place itself at the point (0, 0) by default, so we manually move it to begin at (10,20) instead.

Then we create three `Circle` objects, specifying the center point and the radius of each, then tell each `Circle` object which color to fill itself with.

For the third circle, we could have just used `Color.GREEN`, but I wanted to demonstrate that JavaFX allows you to set colors using web-like hex values, too.

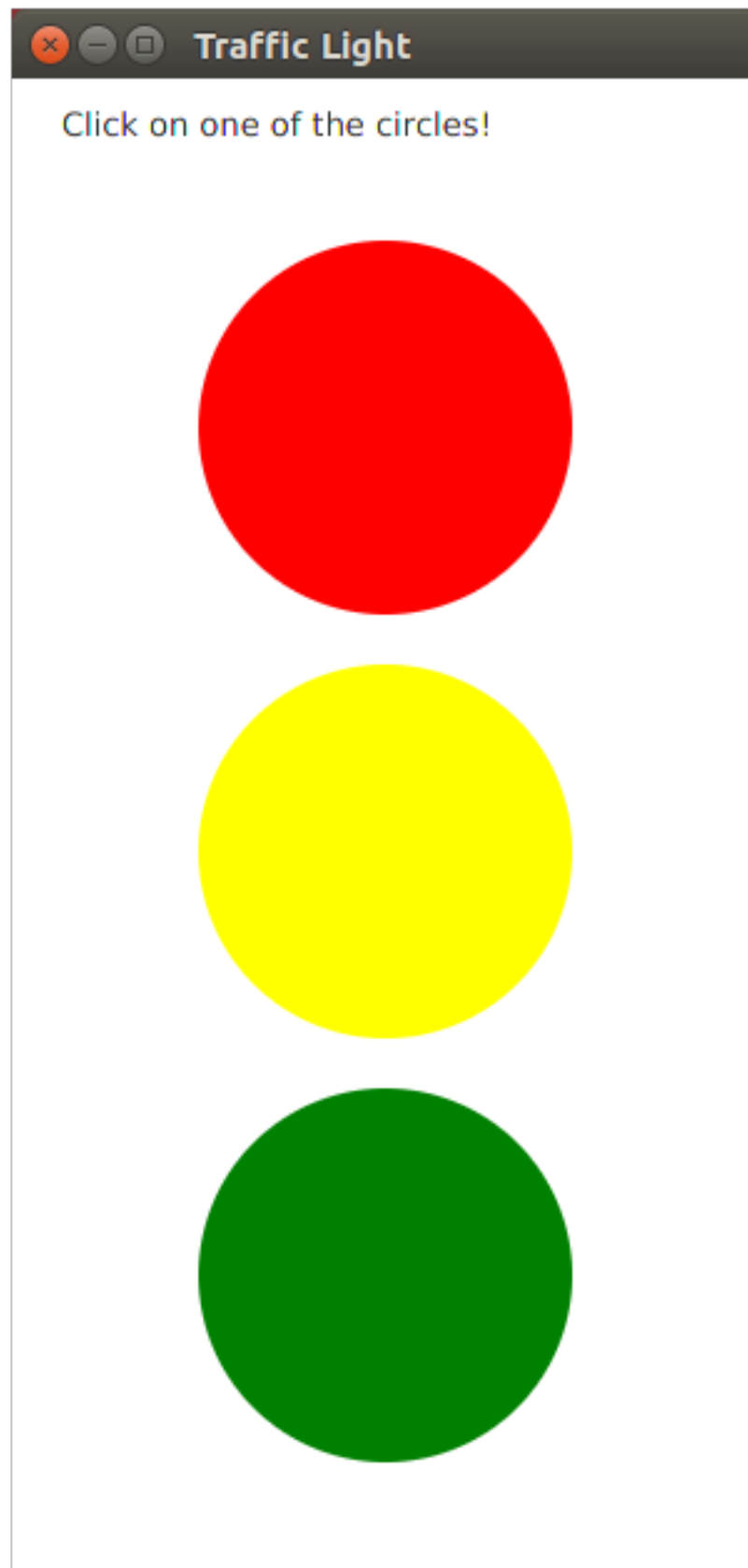
Then, we add an anonymous class to the `Scene` object to handle mouse clicks. Unlike the previous exercise, we actually want to know *where* they clicked, so we are going to be calling the `getX()` and `getY()` methods of the `MouseEvent` parameter.

JavaFX `Circle` objects are handy, because you can just ask them if a point is inside them using their built-in `contains()` method. So if the click was inside the first circle, we update the text of the label to say so. And if the click wasn't inside *any* of the circles, we just change the label to say where they clicked.

Finally, we just add the label and our three circles to the scene graph using the handy `addAll()` method. (We could also just have called `add()` four times.)



## What You Should See



Linux screenshot of TrafficLight

JavaFX is nice, because all we have to do is change the label in our `MouseEvent` handler, and the application notices automatically that one of the nodes in the scene graph has changed and it redraws everything for us. We don't even have to paint over the old message using a white box!



## Study Drills

1. Look at the javadoc documentation for `javafx.scene.paint.Color` and change the code so that when you click on the *middle* circle, it changes color to RGB (255, 182, 193). Then it should change back to yellow when you click it again.

# Exercise 52: Animation

In this exercise we are still using the scene graph's ability to redraw itself when things have changed, but we are now back to using an `AnimationTimer` to get things to move around without any input from the user.

We're going to make a logo "bounce" around the screen.

BouncingLogo.java

---

```
1  import javafx.animation.AnimationTimer;
2  import javafx.application.Application;
3  import javafx.event.EventHandler;
4  import javafx.scene.Group;
5  import javafx.scene.Scene;
6  import javafx.scene.image.Image;
7  import javafx.scene.image.ImageView;
8  import javafx.scene.input.KeyCode;
9  import javafx.scene.input.KeyEvent;
10 import javafx.scene.paint.Color;
11 import javafx.scene.shape.Rectangle;
12 import javafx.stage.Stage;
13
14 public class BouncingLogo extends Application {
15     private boolean flashing = false;
16     private double x, y, dx, dy;
17
18     @Override
19     public void start(Stage stage) {
20         Group root = new Group();
21         Scene scene = new Scene(root, 900, 700);
22
23         x = 500;
24         y = 350;
25         dx = 5;
26         dy = 5;
27
28         Image img = new Image("file:datafiles/ljthw-logo-32px.png");
29         ImageView logo = new ImageView(img);
30         logo.relocate(x,y);
```

```
31
32     Rectangle rect = new Rectangle(x, y, img.getWidth(), img.getHeight());
33     rect.setFill(Color.WHITE);
34
35     scene.addEventHandler(KeyEvent.KEY_PRESSED,
36         new EventHandler<KeyEvent>() {
37         @Override
38         public void handle(KeyEvent t) {
39             if ( t.getCode() == KeyCode.SPACE )
40                 flashing = ! flashing;
41         }
42     });
43
44     root.getChildren().add(rect);
45     root.getChildren().add(logo);
46
47     stage.setTitle("Bouncing Logo");
48     stage.setScene(scene);
49     stage.show();
50
51     new AnimationTimer() {
52         @Override
53         public void handle(long now) {
54             x += dx;
55             y += dy;
56
57             if ( x < 0 || x+img.getWidth() > scene.getWidth() )
58                 dx = -dx;
59             if ( y < 0 || y+img.getHeight() > scene.getHeight() )
60                 dy = -dy;
61
62             if ( flashing ) {
63                 double red = Math.random();
64                 double green = Math.random();
65                 double blue = Math.random();
66                 rect.setFill(Color.color(red,green,blue));
67             }
68             logo.relocate(x,y);
69             rect.relocate(x,y);
70         }
71     }.start();
72 }
```

```
73
74     public static void main( String[] args ) { launch(args); }
75 }
```

---

We use two variables  $x$  and  $y$  to keep track of the location of the image. The variable  $dx$  (delta- $x$ ) will be used to keep track of how much the  $x$ -value of the image will change in one frame, and  $dy$  is the same but for the  $y$ -value. It is possible to use an angle and a speed instead, but then you have to get into trigonometry, and that's annoying to my high school students.

Since these variables will be changed by our anonymous inner classes we have to make them instance variables and not just local variables inside `start()`.

We load an image from a file in the `datafiles` folder and create an `Image` object out of it. But then we wrap the `Image` in an `ImageView` object, which is a node that can be added to the scene graph. (You can draw `Image` objects directly onto a *Canvas* using the `GraphicsContext`'s `drawImage()` method, but you can't add `Image` objects directly to the scene graph unless they're inside an `ImageView`. Go figure.) We also create a `Rectangle` object of the same size that will be drawn underneath the image.

So the image (and the rectangle underneath it) start a little right of the center of the screen, and they will move 5 pixels right ( $dx$ ) and 5 pixels down ( $dy$ ) every frame.

We add a `KeyEvent` handler to the `Scene` to manually deal with key presses. When they press a key, if that key was the space bar, we toggle the value of the *flashing* variable. (We'll see what this variable does shortly.)

Then we add the `Rectangle` to the scene graph and *then* add the `ImageView` node. Adding them in this order draws the `Rectangle` underneath and the `Image` on top of the `Rectangle`. And then we display the window.

The anonymous class for our `AnimationTimer` is a little more complicated. Every frame we add  $dx$  and  $dy$  to the position of the image.

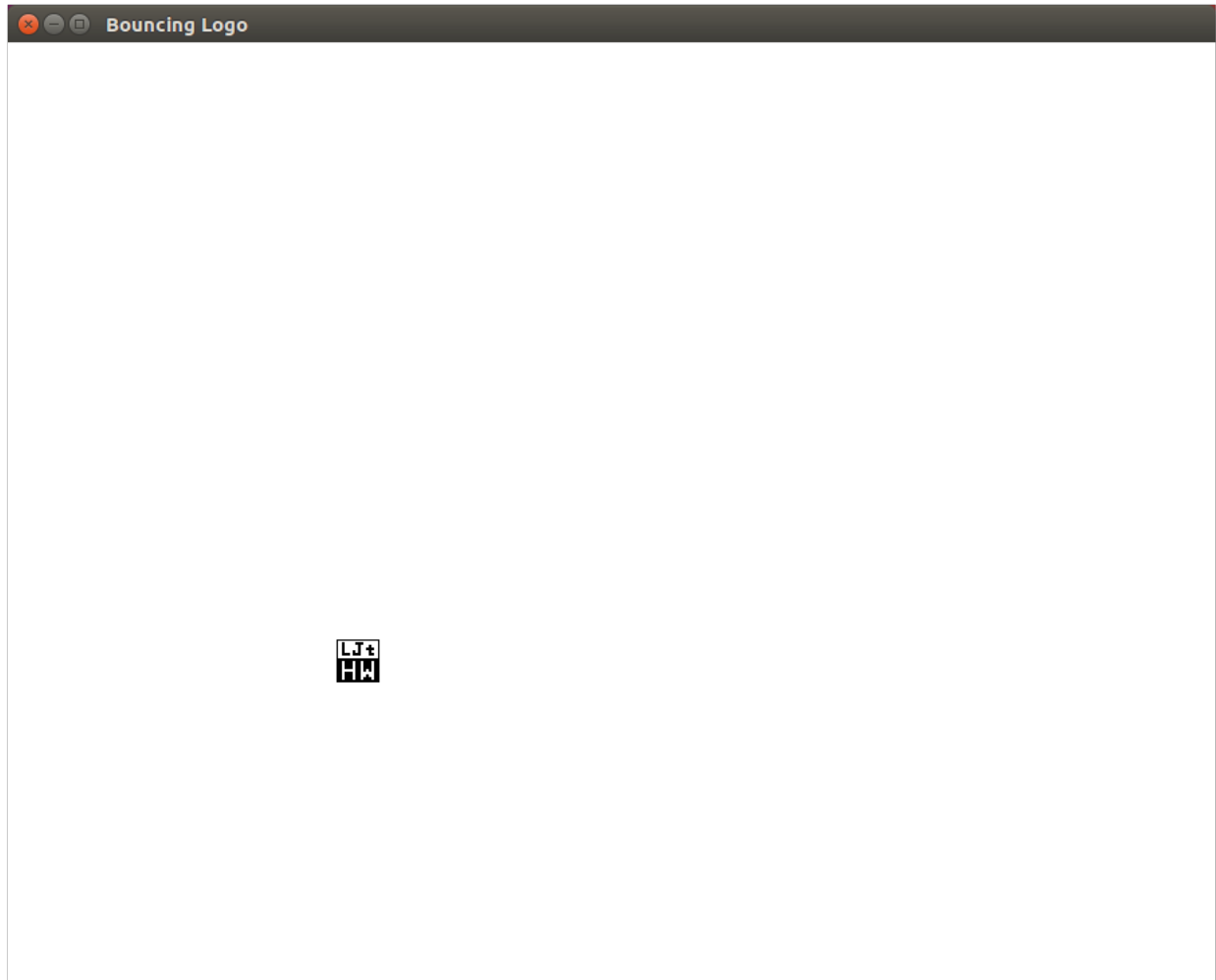
Then, if the new value for  $x$  is below 0 or closer to the wall than the image's width, we have "hit" either the left or right wall. So we change the sign of  $dx$ . (That is, if it used to be 5, it is now -5 and vice-versa.) This will make the image go in the other direction next frame.

Similarly, if  $y$ 's new value is out of bounds, we change  $dy$  to bounce vertically.

If the *flashing* variable is `true`, we pick random values from 0 to 1 for the red, green and blue components of a new color, then tell the rectangle to fill itself using that new color. This will happen every frame, so the rectangle under the image will flash annoyingly until they press the space bar again.

Finally, we `relocate()` the `ImageView` and `Rectangle` objects to that new location, which modifies the scene graph and triggers a re-draw of the screen.

## What You Should See



Linux screenshot of BouncingLogo

(It looks cooler in motion.)



## Study Drills

1. Add some code to the `KeyEvent` handler to reset the rectangle's fill color to white *and* stop flashing when they press "B" (`KeyCode.B`).
2. Add more code to the `KeyEvent` handler to make the image (and rectangle) stop moving permanently when they press "Q".
3. Add code to automatically quit the program when they press the ESCAPE key. (You can quit any running Java application by calling `System.exit(0)`).
4. If you want, see if you can figure out how to make the image and rectangle start moving in a random direction and random speed when they press "R". Picking a random  $dx$  and random  $dy$  between -7 and +7 is a good approach. (Mr. Mitchell's real-life students don't need to do this Study Drill to receive credit.)

# Exercise 53: Handling Keypress Events

It's easy to handle keypresses in a game incorrectly. Say you want the character to move right when they press the right arrow. You might think of code like this:

```
if ( t.getCode() == KeyCode.RIGHT )
    x += 5;
```

And that *will* work. But it ends up being choppy, and they can't just hold down the key! So let's look at a way that works much better.

Plotter.java

---

```
1  import javafx.animation.AnimationTimer;
2  import javafx.application.Application;
3  import javafx.event.EventHandler;
4  import javafx.scene.Group;
5  import javafx.scene.Scene;
6  import javafx.scene.canvas.Canvas;
7  import javafx.scene.canvas.GraphicsContext;
8  import javafx.scene.input.KeyCode;
9  import javafx.scene.input.KeyEvent;
10 import javafx.scene.paint.Color;
11 import javafx.stage.Stage;
12
13 public class Plotter extends Application {
14     private double x, y;
15     private boolean up, down, left, right;
16
17     @Override
18     public void start(Stage stage) {
19         Group root = new Group();
20         Scene scene = new Scene(root);
21
22         Canvas canvas = new Canvas(900,700);
23         GraphicsContext gc = canvas.getGraphicsContext2D();
24
25         x = 450;
26         y = 350;
```



```
27     int speed = 7;
28
29     scene.addEventHandler(KeyEvent.KEY_PRESSED,
30         new EventHandler<KeyEvent>() {
31             @Override
32             public void handle(KeyEvent t) {
33                 if ( t.getCode() == KeyCode.UP || t.getCode() == KeyCode.W )
34                     up = true;
35                 if ( t.getCode() == KeyCode.DOWN || t.getCode() == KeyCode.S )
36                     down = true;
37                 if ( t.getCode() == KeyCode.LEFT || t.getCode() == KeyCode.A )
38                     left = true;
39                 if ( t.getCode() == KeyCode.RIGHT || t.getCode() == KeyCode.D )
40                     right = true;
41
42                 if ( t.getCode() == KeyCode.DIGIT1 )
43                     gc.setFill(Color.BLACK);
44                 else if ( t.getCode() == KeyCode.DIGIT2 )
45                     gc.setFill(Color.RED);
46                 else if ( t.getCode() == KeyCode.DIGIT3 )
47                     gc.setFill(Color.GREEN);
48                 else if ( t.getCode() == KeyCode.DIGIT4 )
49                     gc.setFill(Color.BLUE);
50             }
51         });
52
53     scene.addEventHandler(KeyEvent.KEY_RELEASED,
54         new EventHandler<KeyEvent>() {
55             @Override
56             public void handle(KeyEvent t) {
57                 if ( t.getCode() == KeyCode.UP || t.getCode() == KeyCode.W )
58                     up = false;
59                 if ( t.getCode() == KeyCode.DOWN || t.getCode() == KeyCode.S )
60                     down = false;
61                 if ( t.getCode() == KeyCode.LEFT || t.getCode() == KeyCode.A )
62                     left = false;
63                 if ( t.getCode() == KeyCode.RIGHT || t.getCode() == KeyCode.D )
64                     right = false;
65             }
66         });
67
68     root.getChildren().add(canvas);
```

```

69
70     stage.setTitle("Plotter");
71     stage.setScene(scene);
72     stage.show();
73
74     new AnimationTimer() {
75         @Override
76         public void handle(long now) {
77             if ( up )
78                 y -= speed;
79             else if ( down )
80                 y += speed;
81             if ( left )
82                 x -= speed;
83             else if ( right )
84                 x += speed;
85
86             gc.fillRect(x, y, 20, 20);
87         }
88     }.start();
89 }
90
91 public static void main( String[] args ) { launch(args); }
92 }

```

---

We are using four booleans to keep track of their intended movement. If they want to go up, then *up* is true. If they don't want to go up, then *up* should be false.



Remember that private fields / instance variables have a default value. For booleans, the default is `false` so I didn't bother to manually set the variables to `false` this time.

You should know that if you type the letter “q” in a word processing document, there are three separate key events that will occur:

- A `KEY_PRESSED` event happens for the letter “Q”.
- A `KEY_TYPED` event happens for the letter “q”.
- A `KEY_RELEASED` event happens once they let go of the “Q” key.

`KEY_PRESSED` and `KEY_RELEASED` are very low-level and run for any key, including modifier keys like `SHIFT` and `F1`. `KEY_TYPED` is high-level and only triggers when an actual character is typed. For example, when I want to type a capital “Q”, there are *five* `KeyEvent`s generated:

1. KEY\_PRESSED for SHIFT
2. KEY\_PRESSED for Q
3. KEY\_TYPED for “Q”
4. KEY\_RELEASED for “Q”
5. KEY\_RELEASED for SHIFT

Notice that there’s no difference between a capital “Q” and a lowercase “q” in the KEY\_PRESSED or KEY\_RELEASED events. They are both just “Q”.

Also, if you hold down the keys for a long time, your operating system’s key repeat will kick in after a second, and you will get multiple KEY\_TYPED events.

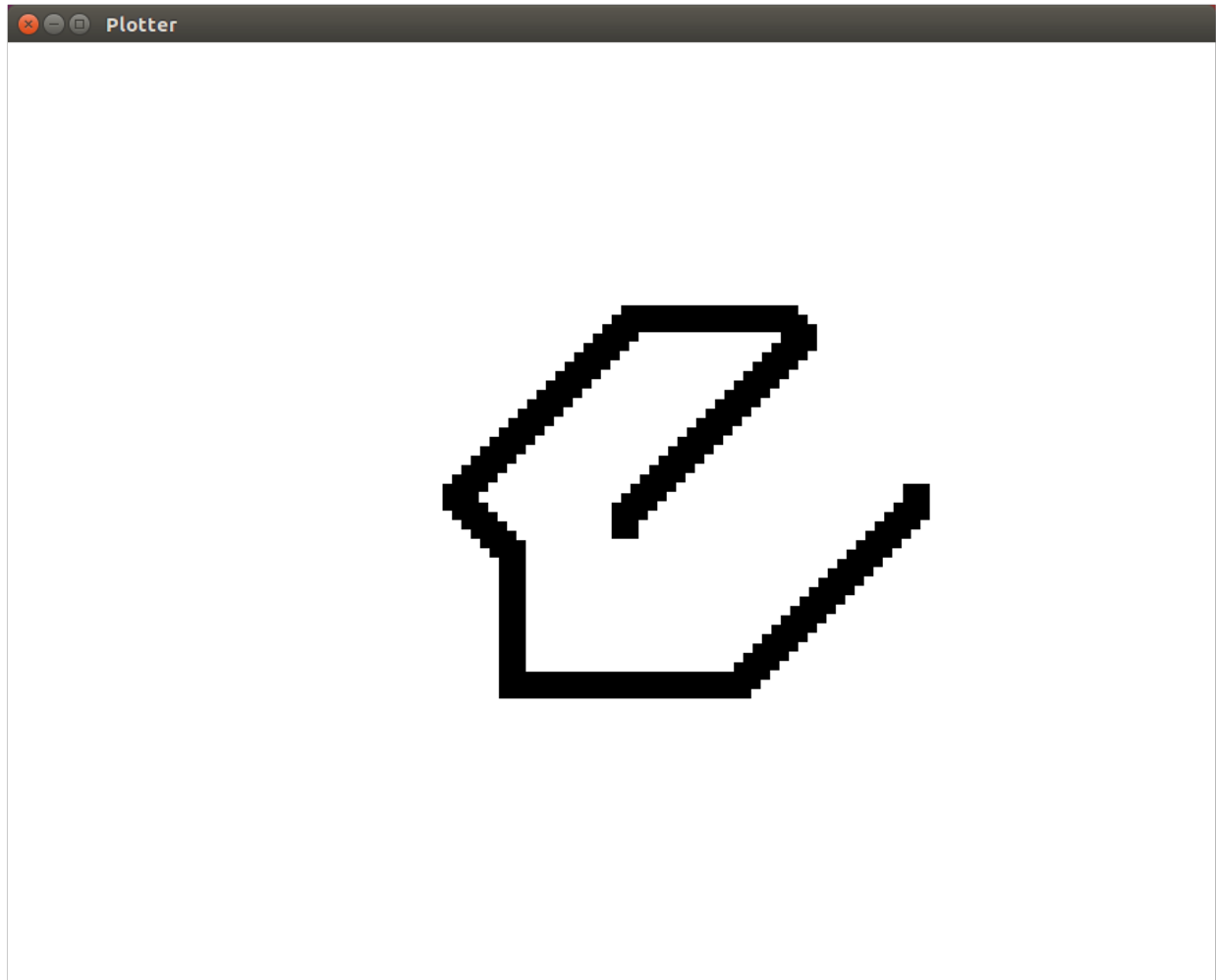
So here’s how we deal with it: if the up arrow is PRESSED, we will set the corresponding boolean variable to `true`. Then, when the up arrow is RELEASED later, we change the variable back to `false`.

This way, we don’t have to deal with keyboard repeat issues, and they can hold down more than one key at the same time and everything just works.

Then, the AnimationTimer handler doesn’t have to care about keys at all. If the boolean variable for *up* is `true`, then move up.

I used some `else if` statements to make it so that *up* trumps *down* and *left* trumps *right* but you could remove those and just make all four of them `if` statements and it should still work out. If they hold down UP and DOWN at the same time, they will cancel each other out anyway.

## What You Should See



Linux screenshot of Plotter

I guess I should add that you only use `getCode()` for `PRESSED` and `RELEASED` events. If you actually want to check the `KEY_TYPED` event then you'll want to call `getCharacter()` instead to get the `String` they actually typed.

So that's it for new stuff about graphics. In the next two exercises we will fit everything together to make graphical interfaces for two of the games you made in previous exercises!



## Study Drills

1. Make a second box. Use WASD to move around the first and the arrow keys for the second. It is okay if they both change color or if only one of them does.
2. Make it so that they can increase the speed of the boxes with PAGE\_UP and slow them down with PAGE\_DOWN. This is trickier than it sounds because any changes you make will run a *bunch* of times between PRESSED and RELEASED.

# Exercise 54: Graphical Noughts and Crosses

Way back in exercise 15 we made `NoughtsCrossesObject`, an object with methods for all the interesting bits of tic-tac-toe. You'll need that code for this exercise, so if you didn't type it up (and fix it!) you should do that now.

In this exercise, we are going to make a simple graphical interface for the game, so that we can use the mouse!

`NoughtsCrossesGUI.java`

---

```
1  import javafx.animation.AnimationTimer;
2  import javafx.application.Application;
3  import javafx.event.EventHandler;
4  import javafx.scene.Group;
5  import javafx.scene.Scene;
6  import javafx.scene.canvas.Canvas;
7  import javafx.scene.canvas.GraphicsContext;
8  import javafx.scene.input.MouseEvent;
9  import javafx.scene.paint.Color;
10 import javafx.scene.text.Font;
11 import javafx.scene.text.FontWeight;
12 import javafx.stage.Stage;
13
14 public class NoughtsCrossesGUI extends Application {
15     private GraphicsContext gc;
16     private String player = "X";
17     private NoughtsCrossesObject ttt;
18     private boolean started = false;
19     private boolean gameOver = false;
20     private boolean drawLabels = false;
21
22     @Override
23     public void start(Stage stage) {
24         Group root = new Group();
25         Scene scene = new Scene(root);
26
27         Canvas canvas = new Canvas(800,800);
```

```
28     gc = canvas.getGraphicsContext2D();
29
30     ttt = new NoughtsCrossesObject();
31
32     canvas.addEventHandler(MouseEvent.MOUSE_CLICKED,
33         new EventHandler<MouseEvent>() {
34             @Override
35             public void handle(MouseEvent t) {
36                 if ( gameOver ) {
37                     started = false;
38                     gameOver = false;
39                     ttt = new NoughtsCrossesObject();
40                     return;
41                 }
42
43                 if ( ! started ) {
44                     started = true;
45                     return;
46                 }
47
48                 int r = ((int)t.getY() - 85) / 200;
49                 int c = ((int)t.getX() - 125) / 200;
50
51                 if ( ttt.isTaken(r,c) == false ) {
52                     ttt.playMove(player,r,c);
53                     player = ( player.equals("X") ? "O" : "X" );
54                 }
55             }
56         });
57
58     root.getChildren().add(canvas);
59
60     stage.setTitle("Noughts and Crosses");
61     stage.setScene(scene);
62     stage.show();
63
64     new AnimationTimer() {
65         @Override
66         public void handle(long now) {
67             drawEverything();
68         }
69     }.start();
```

```
70     }
71
72     public void drawEverything() {
73         gc.setFill(Color.WHITE);
74         gc.fillRect(0, 0, 800, 800);
75
76         if ( ! started )
77         {
78             gc.setFill(Color.BLACK);
79             gc.setFont(Font.font("Arial", FontWeight.BOLD, 48) );
80             gc.fillText( "click to play", 275, 400 );
81         }
82         else
83         {
84             drawEmptyBoard();
85             for ( int r=0; r<3; r++ )
86                 for ( int c=0; c<3; c++ )
87                     if ( ttt.isTaken(r,c) )
88                         drawSymbol(ttt.playerAt(r,c), r, c);
89
90             gameOver = (ttt.isWinner("X") || ttt.isWinner("O") || ttt.isFull());
91             if ( gameOver )
92             {
93                 gc.setFont(Font.font("Arial", FontWeight.BOLD, 144));
94                 gc.setFill(Color.BLACK);
95                 if ( ttt.isWinner("X") )
96                     gc.fillText("X wins!", 160, 400);
97                 if ( ttt.isWinner("O") )
98                     gc.fillText("O wins!", 160, 400);
99                 if ( ttt.isCat() )
100                     gc.fillText("TIE GAME", 50, 400);
101             }
102             else
103             {
104                 gc.setFont(Font.font("Arial", FontWeight.BOLD, 72));
105                 gc.setFill(Color.BLACK);
106                 gc.fillText(player + ", go.", 10, 730);
107             }
108         }
109     }
110
111     public void drawEmptyBoard() {
```



```
112     gc.setStroke(Color.BLACK);
113     // horizontal lines
114     gc.strokeLine(100,250,700,250);
115     gc.strokeLine(100,450,700,450);
116     // vertical lines
117     gc.strokeLine(300,50,300,650);
118     gc.strokeLine(500,50,500,650);
119     // labels
120     if ( drawLabels ) {
121         gc.setFont(Font.getDefault());
122         gc.setFill(Color.BLACK);
123         gc.fillText("(0,0)",105,65);
124         gc.fillText("(1,0)",105,265);
125         gc.fillText("(2,0)",105,465);
126
127         gc.fillText("(0,1)",305,65);
128         gc.fillText("(1,1)",305,265);
129         gc.fillText("(2,1)",305,465);
130
131         gc.fillText("(0,2)",505,65);
132         gc.fillText("(1,2)",505,265);
133         gc.fillText("(2,2)",505,465);
134     }
135 }
136
137 public void drawSymbol(String p, int r, int c) {
138     int x = 200*c + 125;
139     int y = 200*r + 85;
140     if ( p.equals("X") ) {
141         gc.setStroke(Color.RED);
142         gc.strokeLine(x,y,x+150,y+150);
143         gc.strokeLine(x+150,y,x,y+150);
144     }
145     else if ( p.equals("O") ) {
146         gc.setFill(Color.YELLOW);
147         gc.fillOval(x,y,150,150);
148     }
149 }
150
151 public static void main( String[] args ) { launch(args); }
152 }
```

---

The first complication is that now a mouse click means different things depending on which state the game is in. A click could mean “begin the game” or “put a symbol here” or “start over”.

One (good) way to handle it would be to create several different Scene objects, each with their own MouseEvent handlers and switch between the Scenes when the game changes state.

I have not done that. I have opted for the less-clean “variables and if statements” approach.

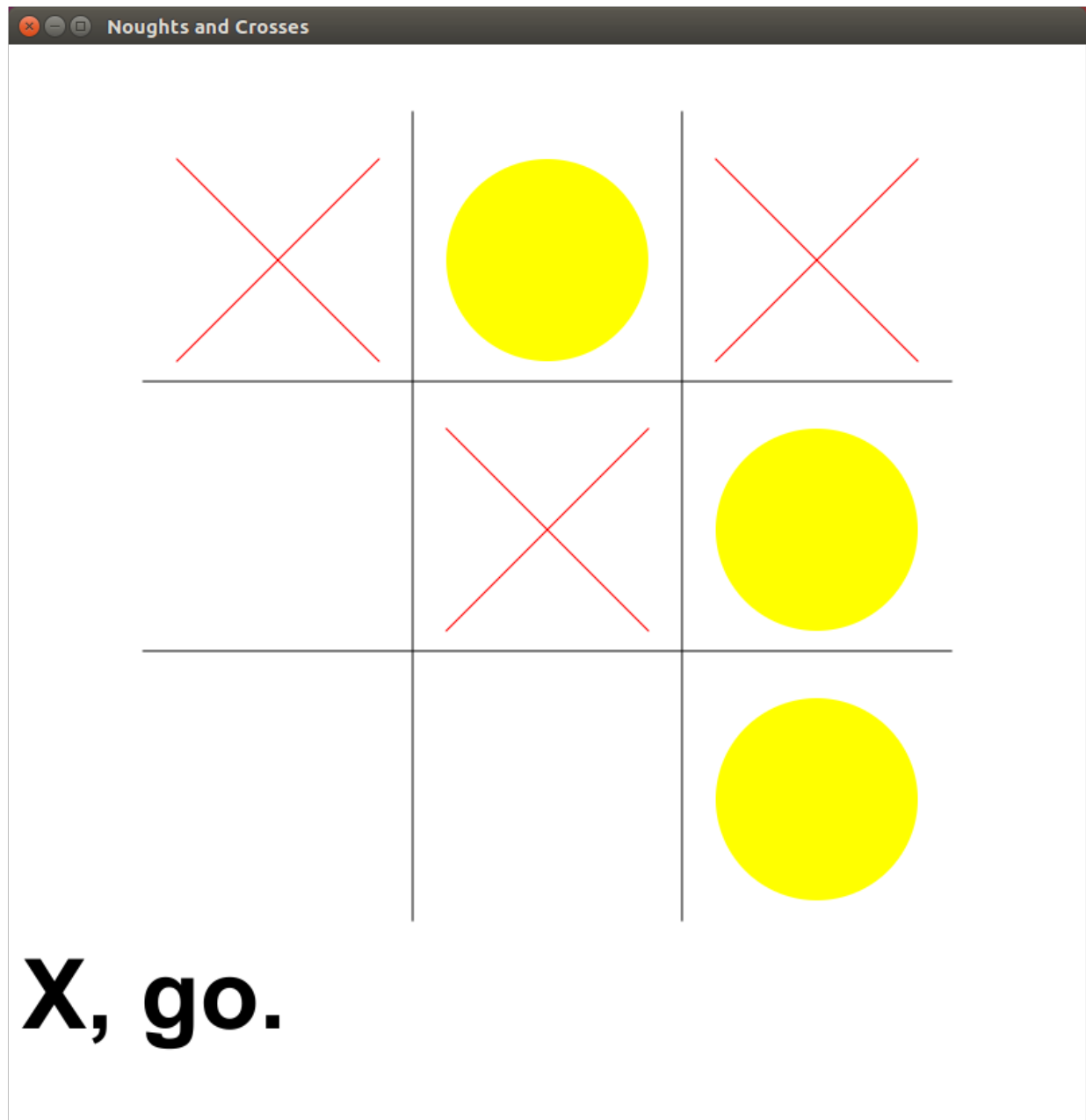
When they click, if the game is over we change the variables so that the game is *not* over anymore and reset the game board by just creating a new NoughtsCrossesObject.

If they click when the game is “not started” we just change the state to started and return to get out.

Finally, if they click during normal gameplay, we have to figure out which row and column they clicked in. A little math gives us the answer. Then we attempt to place a symbol in that square and switch players.

If the square they clicked on is already taken, we don’t display an error message; we merely do nothing.

## What You Should See



Linux screenshot of NoughtsCrossesGUI

The `AnimationTimer` handler is pretty simple; we just call a method called `drawEverything()`. Creative, huh?

The `drawEverything()` method begins by “clearing the screen” (it paints over the entire canvas with a big white box). But what it does next depends on which state the game is in. If the game hasn’t

started yet, we just say “click to play”.

Once the game *has* started we draw the grid on the screen and then loop through the rows and columns. If a particular square has a symbol in it we draw that symbol on the screen.

Then we check if the game is over for some reason, and if so we display the appropriate message on the screen. If the game *isn't* over we ask the current player to take their turn.

The only tricky bit left is in the `drawSymbol()` method. It is called with the row and column to draw, but needs to know the screen coordinates instead. So there is a little math, there, too.

Not bad, eh? Barely 150 lines of code to make a mouse-driven user interface for a game! JavaFX is nice.

Some pretty tough Study Drills this time, then another GUI for a game in the final exercise!



## Study Drills

1. Make a single-player version of the game where the computer takes turns with the human. It is okay if the computer isn't very good, but it shouldn't break the rules.
2. If you're feeling fancy, make the “click to play” screen show a menu so they can choose 1 or 2 players depending on where they click.
3. Figure out how to draw a line through the three winning symbols when someone wins. This is pretty difficult.

# Exercise 55: Graphical Drop Game

For our final exercise we will make a graphical interface for the DropGameHelper you made several exercises ago. You'll need to make sure that DropGameHelper.java is in the same folder as the code you write for this exercise, in addition to the code for DropGamePiece.java, Location.java and the helper interfaces Locatable.java and Translatable.java!

DropGameGUI.java

---

```
1  import javafx.animation.AnimationTimer;
2  import javafx.application.Application;
3  import javafx.event.EventHandler;
4  import javafx.scene.Group;
5  import javafx.scene.Scene;
6  import javafx.scene.canvas.Canvas;
7  import javafx.scene.canvas.GraphicsContext;
8  import javafx.scene.input.MouseEvent;
9  import javafx.scene.paint.Color;
10 import javafx.scene.text.Font;
11 import javafx.scene.text.FontWeight;
12 import javafx.stage.Stage;
13
14 public class DropGameGUI extends Application {
15     private GraphicsContext gc;
16     private int player = 1;
17     private DropGameHelper game;
18     private boolean started = false;
19     private boolean gameOver = false;
20     private Location[][] locs;
21
22     @Override
23     public void start(Stage stage) {
24         Group root = new Group();
25         Scene scene = new Scene(root);
26
27         Canvas canvas = new Canvas(900, 800);
28         gc = canvas.getGraphicsContext2D();
29
30         game = new DropGameHelper(6, 7);
31         locs = new Location[6][7];
```

```

32     for ( int r=0; r<locs.length; r++ )
33         for ( int c=0; c<locs[0].length; c++ )
34             locs[r][c] = new Location(r,c);
35     gc.setStroke(Color.BLACK);
36
37     canvas.addEventHandler(MouseEvent.MOUSE_CLICKED,
38         new EventHandler<MouseEvent>() {
39         @Override
40         public void handle(MouseEvent t) {
41             if ( gameOver ) {
42                 started = false;
43                 gameOver = false;
44                 game = new DropGameHelper(6, 7);
45             }
46             else if ( ! started ) {
47                 started = true;
48             }
49             else {
50                 int xOffset = 100, colWidth = 100;
51                 int col = ((int)t.getX() - xOffset) / colWidth;
52                 if ( game.isValid(col) && ! game.isFull(col) ) {
53                     game.playMove(""+player, col);
54                     player = (player%2) + 1;
55                 }
56             }
57         }
58     });
59
60     root.getChildren().add(canvas);
61
62     stage.setTitle("Drop Game");
63     stage.setScene(scene);
64     stage.show();
65
66     new AnimationTimer() {
67         @Override
68         public void handle(long now) {
69             drawEverything();
70         }
71     }.start();
72 }
73

```

```

74     public void drawEverything() {
75         gc.setFill(Color.WHITE);
76         gc.fillRect(0, 0, 900, 800);
77
78         if ( ! started )
79         {
80             gc.setFill(Color.BLACK);
81             gc.setFont(Font.font("Arial", FontWeight.BOLD, 48) );
82             gc.fillText( "click to begin", 275, 400 );
83         }
84         else
85         {
86             drawBoard();
87             gameOver = game.isWinner("1") || game.isWinner("2") || game.isFull();
88             if ( gameOver ) {
89                 gc.setFont(Font.font("Arial", FontWeight.BOLD, 120));
90                 gc.setFill(Color.BLACK);
91                 if ( game.isWinner("1") )
92                     gc.fillText("Player 1 wins!", 50, 400);
93                 else if ( game.isWinner("2") )
94                     gc.fillText("Player 2 wins!", 50, 400);
95                 else if ( game.isFull() )
96                     gc.fillText("TIE GAME", 150, 400);
97             }
98             else {
99                 gc.setFont(Font.font("Arial", FontWeight.BOLD, 72));
100                gc.setFill(Color.BLACK);
101                gc.fillText("Player " + player + ", go.", 30, 750);
102            }
103        }
104    }
105
106    public void drawBoard() {
107        double xOffset = 100;
108        double yOffset = 50;
109        double colWidth = 100;
110        double rowWidth = 100;
111        for ( int r=0; r<game.numRows(); r++ ) {
112            for ( int c=0; c<game.numCols(); c++ ) {
113                double x = xOffset + c*colWidth;
114                double y = yOffset + r*rowWidth;
115                if ( game.playerAt(locs[r][c]).equals(" ") )

```

```
116         gc.strokeOval(x, y, 80, 80);
117     else {
118         if ( game.playerAt(locs[r][c]).equals("1") )
119             gc.setFill(Color.RED);
120         else
121             gc.setFill(Color.YELLOW);
122         gc.fillOval(x, y, 80, 80);
123     }
124 }
125 }
126 }
127
128 public static void main( String[] args ) { launch(args); }
129 }
```

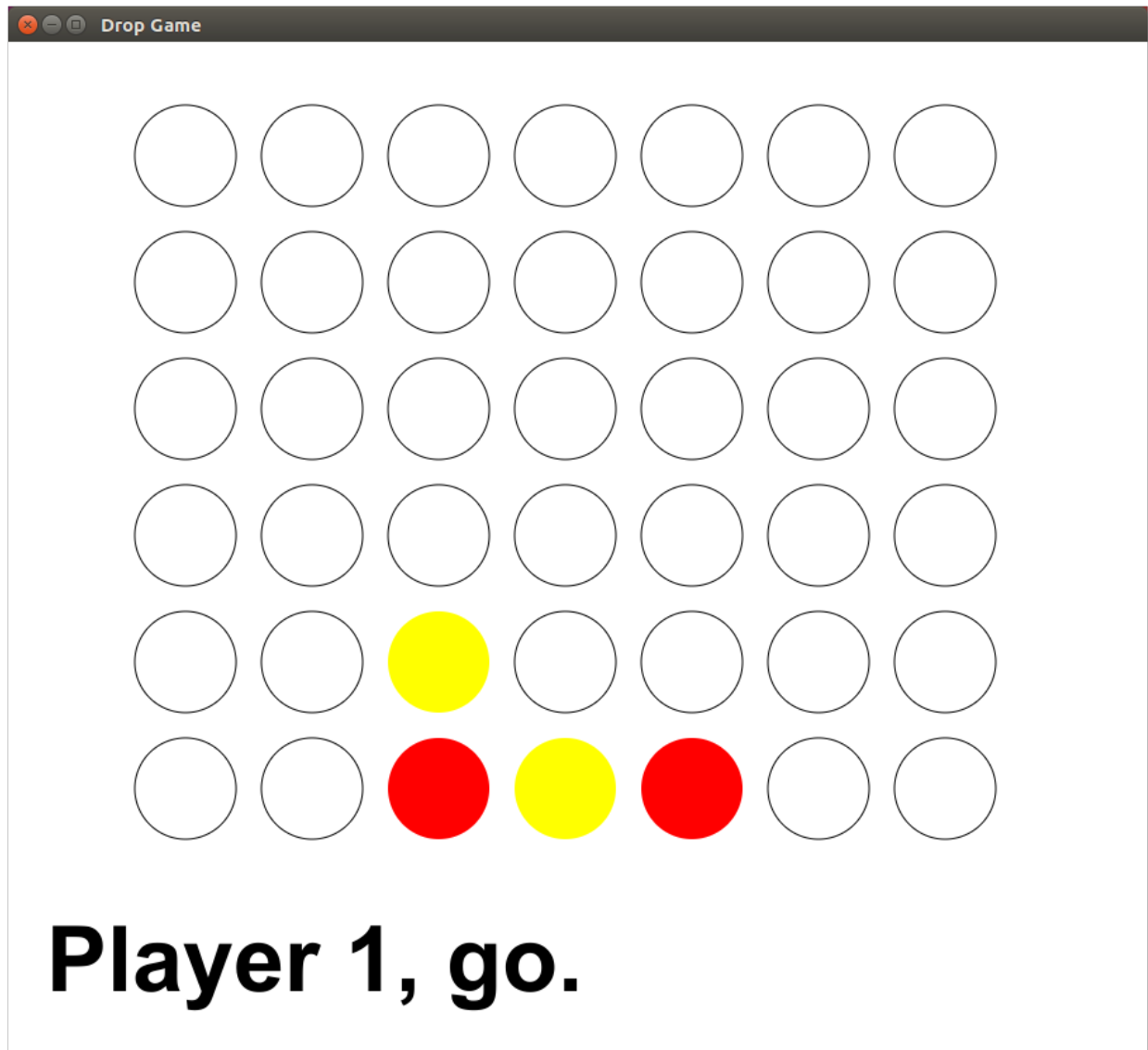
---

This shares a lot of the design with the previous game, since I wrote both of them!

Using the `Location` objects made the design of the original `DropGame` easier, but it makes this a little trickier, since we are going to get rows and columns out of our mouse clicks, but the `DropGameHelper` needs `Locations`. So I made a two-dimensional array of `Location` objects at the very beginning.



## What You Should See



Linux screenshot of DropGameGUI

The mouse click handler is similar to the previous exercise; how we respond depends on which state the game is in. Assuming the game is started and not over, we figure out which column they clicked in using some math. Then, if the game object tells us it is a legal column and it isn't already full, we play the appropriate piece in there.

You might have noticed that in the original DropGame, we stored “#” and “O” symbols in the board. But the game doesn't care what symbols we use, as long as we are consistent.

So for this version, I decided to use the String “1” for the first player's moves and “2” for the second

player. This allows me to just use an integer to keep track of whose turn it is, and I convert it to a String before putting it in the game board.

```
player = (player%2) + 1;
```

I don't know if you remember that “%” means “modulus”; it is the remainder operator in Java (and many other languages). So that line of code takes the remainder when *player* is divided by 2. It then adds 1 to that remainder and makes that the new value of *player*.

So if *player* is 1, the remainder will be 1... plus 1 is 2. And if *player* is 2, the remainder will be 0... plus 1 is 1. I could have used a ternary operator like I have been using, but I figured the last exercise wasn't too late to show you a new trick!

The rest of the code is pretty clear, I think.

In the `drawBoard()` method, we loop through all the rows and columns. We want to know what is in the current square, but the `DropGameHelper` will only deal with `Locations`, not rows and columns. So...

```
game.playerAt( locs[r][c] )
```

...pulls the right `Location` object out of the 2-D array. I could have just written

```
game.playerAt( new Location(r,c) )
```

However, that would *create* 42 new `Location` objects every time `drawBoard()` is called, which is around 60 times per second. The Java runtime would probably figure it out and do the right thing, but I just didn't feel good about creating and throwing away 2500 `Location` objects every second.

I could have also just gone back into the code for the `DropGameHelper` and added a second `playerAt()` method that accepts a row and column instead of a `Location` object. Which probably wouldn't be a bad idea, either.

Anyway, there's no grid lines in this board. I decided it looked nice enough to show an outline circle for blank spaces and a colored circle for the different players, and it made the code for `drawBoard()` pretty nice.

In fact, this code is even shorter than the previous exercise!



## Study Drills

1. Convert all the DropGame classes to use a package called `dropgame`. Make sure to include `Locatable`, `Translatable`, `Location` and `DropGamePiece`.
2. Create a JAR file out of the `DropGameGUI` and all its classes.
3. If you can, figure out how to make the game show where the next piece will be dropped. Make a new `MouseEvent` handler for the `MOUSE_MOVED` event and draw a light grey piece in the column the mouse is currently hovering over. It should move from column to column as they move the mouse around. This is pretty tough.

# Next Steps

You made it!

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

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

However, your journey isn't over yet. Typing in someone else's programs is hard but writing your own programs from scratch is a lot harder.

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

Currently, you are lacking in two main areas:

- You need a lot more practice, especially writing your own programs.
- You don't know the standard library very well.

## You don't know the standard library.

Professional Java programmers spend a lot of time writing small amounts of code to “glue together” modules written by other people. Many of these are included with Java itself and are available to “import”.

## You need a lot more practice.

The students I teach at my public school *do* use the exercises in this book to start learning, but that's not all I make them do. They also have to write a lot of programs from scratch. If you haven't already worked through the assignments at

[Programming by Doing](https://programmingbydoing.com/)<sup>21</sup>

I *highly* recommend working through them. (They aren't very object-oriented, though.)

---

<sup>21</sup><https://programmingbydoing.com/>

- [CodingBat<sup>22</sup>](#) - short, targeted practice problems. These are especially good practice for students preparing for the Advanced Placement (AP) exam in Computer Science.
- [Project Euler<sup>23</sup>](#) - a series of challenging mathematical/computer programming problems that will require more than just mathematical insights to solve. Although mathematics will help you arrive at elegant and efficient methods, the use of a computer and programming skills will be required to solve most problems.

Thanks so much for using my book to start your journey coding. Please tweet @grahammitchell using the hashtag #LJtHW to let me know!

Happy coding!

– Graham Mitchell

---

<sup>22</sup><http://codingbat.com/java>

<sup>23</sup><https://projecteuler.net/>