STEVE BROWN

JAVA <FOR> FUCKING IDIOTS

LEARN THE BASICS OF JAVA PROGRAMMING WITHOUT ANY EXPERIENCE!

STEVE BROWN

JAVA <FOR> FUCKING IDIOTS

LEARN THE BASICS OF JAVA PROGRAMMING

WITHOUT ANY EXPERIENCE!

CONTENTS

INTRODUCTION

Hi, I’m Steve.

I’m here to teach you the fundamentals of Java. Or the basics of Java. One of those. Or both.

I should probably start the book with a little elevator speech on why it exists.

SICK OF TRYING TO EARN MONEY BOUNCING FROM JOB TO JOB? CLICK HERE TO LEARN HOW TO EARN $120,000 A MONTH FROM HOME!!!

Wait, that’s not right.

Let me start over…

In my experience, many books designed for “absolute beginners” spend maybe a chapter talking to you like a beginner, then start talking to you like you’re someone with average intelligence all the sudden. They forget their audience. It becomes a normal textbook.

I think it’s frustrating. I want to be treated like the idiot I am, damn it.

I assure you that in this book, I will not forget that you’re a fucking idiot like me.

The only real requirement is you can read. And if you’re downloading an eBook and can’t read then I’m not sure what to tell you, friend.

What this book is

This is a fast, easy read that will guide you through the basics of Java; from variables, methods, and operators, to classes, objects, and packages. Don’t know what those words mean? No problem.

My mastery of the English language (worthy of at least a B- grade in high school) will suck you in to a mystical world of Java programming.

The completely home-grown and organic code snippets will spell things out that English otherwise can’t.

The fact that I’m watching YouTube while writing this book will have no adverse effects whatsoever.

And I wrote all of it by myself, so you know the book is completely home grown and organic. And prone to starting a lot of sentences with “and.”

This book’s goal is for you, the air-headed reader with an interest in Java programming, to be able to close this book and say “Hey, I know quite a bit more than I did when I started reading it. And it wasn’t too bad!”

What this book is NOT

Like with any healthy relationship, we should be honest with each other. I’ll be honest with you right now and say what this book is *not* :

1. A detailed guide on how to download, install, and configure Java.

I just don’t think it fits the format or theme of this book, and I could not be less interested in writing about it. I do have a little blurb on this topic, but I’m focusing on the Java language itself.

2. A workbook chalk full of exercises for you to do.

There’s a lot of these to go around considering Java is over 24 years old. I’ll focus on teaching you the concepts, you start dicking around in the code and exercising your mind.

3. A comprehensive guide to Java that fully explores everything it has to offer in detail.

Yeah, that’s gonna be a no from me dog. The more intricate parts of Java I’m only going to lightly touch on. And others we simply won’t get to. That’s the nature of a book for fucking idiots.

4. Written by a Java guru that has decades of experience.

I’ve been writing code in Java for about 6 years but I’m just a person.

5. Emotionally stable

This book will be unfair and lash out at you, making personal attacks at times. I apologize in advance.

6. Likely going to make me any money.

But I’m willing to accept this.

How should I read this book?

It’s meant to be read like something you browse while you’re taking a poop on the toilet, but you enjoy it so much you continue to read it after you’re done. Yeah, like that.

Just read it top to bottom, left to right. No, I don’t have any special goddamn sections with special logos or icons, or any expert opinions… just read the book you shitter.

Buckle up

Get your towels ready, it’s about to go down. We’re about to start Chapter One, and once we start, there’s no turning back.

Unless you turn the page back. In which case, there absolutely is turning back.

CHAPTER 0: SOFTWARE CLIFF NOTES

Just kidding, we're starting at Chapter 0… because programming. Here's a quick summary of software in general to get you up to speed.

This is an operating system: ***Windows*** . It runs on a computer and talks to its individual parts, so we don't have to.

This is a programming language: ***C++*** . It is used by humans to write programs that run on an operating system.

This is a program: ***Google Chrome*** . It's a web browser, written in a programming language, that runs on your operating system and lets you view websites on the internet.

This is the programming language we will use in this book: ***Java*** .

This is the powerhouse of the cell: the ***Mitochondria*** .

These are parentheses: ( )

These are curly braces: { }

These are square braces / brackets: [ ]

These are angle brackets: < >

This is a giraffe:

Wonderful, now that you're up to speed… to the rest of the book!

Java

Программирование

ПРИСОЕДИНИТЬСЯ

CHAPTER 1: WHAT THE FUCK IS A JAVA?

Java is a ***programming language*** (the most popular one in the world, btw). A programming language is a special kind of language that computers can understand.

When you write things down in a programming language, what you create is often referred to as ***code*** . Not Morse code, or any sort of secret code you use to tell someone you're upset instead of just saying *why* you're upset. None of that; computer code.

A programmer therefore is someone that writes code, in a programming language, so that computers can do things for us.

However, computers are idiots, even more so than you are. If you show your Java code to your computer, it doesn't know how to read it. Hah! Stupid machine!

We do something called ***compilation*** to turn our programming language code into a format the computer will understand. Com – pill – ayy – shin. The act of compiling. Like compiling a list of everything you want to order from the Chinese restaurant.

Now, when I saw "we" I don't mean that a human actually *does* this "compilation" thing. Only that we rely upon the "compilation" process to create our programs.

Let me give you an example:

1. Steve writes some code for a program that can generate memes, automagically!
2. A compiler turns his ***source code*** (what he wrote) into ***machine code*** .
3. Steve's computer runs that machine code so he can enjoy his fresh memes.

So, Steve’s source code (human readable) is ***compiled*** by a compiler to create machine code (machine readable). And the machine code is what runs on the computer.

It’s like hiring a translator to speak to someone in a different language.

Source Code -> *Compiler* -> *Machine Code*

“Computer, add 2 and 2 please” -> Compiler -> “01100110101”

Make sense?

You write *source* code, a compiler turns it into *machine* code, and the computer runs that machine code. Get it into your thicc skull (two c’s).

Compiling interest

Now, there is a part I left out. You see, machine code is created specifically for the ***CPU*** of a computer.

At this point your eyes are probably about to roll into the back of your head. Either because you know what this is already, or you’re already confused. Bear with me.

For those of you out of the loop, the CPU (central processing unit) is the brain of the machine that does all the hard work. Some people call it the “chip” or the “processor”. Doesn’t matter what you call it, for all I care you could call it the doorknob.

Anyways, so we create machine code to run on a computer’s processor/CPU. What would happen if we gave that machine code to a different computer?

More specifically, what would happen if I write some code, compile it into a program to run on my computer (with processor A), then give it to my friend to run (with processor B)?

Hmm. I write code and compile it for processor A….my friend has processor B… I guess it will work! Yeah, programs are smart and just work, it’ll be fine!

NO, the answer is that it *won’t work* !

Imagine writing a novel in English and hiring a professional to translate it into Spanish. Would you expect someone who only speaks *Swedish* to understand the translator’s product?

Unfortunately for us, CPUs read specific kinds of computer instructions thanks to how they’re architected. Processor A needs different instructions than Processor B.

How can we solve this? Death? Maybe!

More realistically, I will need to compile my program for processor A, then compile it for processor B, then make sure the right version of the program is given to the right computer.

“Well that sucks!” – Randy Marsh

The Big Idea: Most programming languages are *compiled* from their human-readable source code to *machine code* to run on a specific type of computer.

A reading from the book of Java

The creators of Java grew frustrated and asked, “what if we could write a programming language that worked on every kind of computer?” And that they did.

They came down to mankind and proclaimed for all to hear.

“Behold. We shall create a programming language that will work on every CPU. Programmers will be able to compile code once and run it everywhere!”

“But how is that possible?” the masses of programmers replied. “How can you compile for every machine at the same time? There has to be a version of the program for each type of machine!”

“Fear not, child. The language is Java and Java is good. You shall write your code in Java, and it will be compiled to run on the ***Java Virtual Machine*** .” replied the voices in the sky.

Confused, they questioned the Creators: “But what is the Java Virtual Machine?”

“The JVM? It’s simple,” they answered. “Each machine shall have a program on it called the Java Virtual Machine that understands compiled Java code. You will not need to compile for the machines themselves; you will need to compile for the Java Virtual Machine!”

“But my toaster! I compile my BASIC code to run on the toaster architecture, how will I be able to run my programs?” questioned one brave coder.

“Hah! Haha! Silly human. We have a version of the JVM for toasters. And

"Hah! Haha! Silly human. We have a version of the JVM for toasters. And refrigerators. And PCs, and Macs, and Linux. Install the right JVM for each system, and Java programs will be able to run."

And so, it came to be.

Serious time

Do you wish you'd never laid eyes upon such a passage? Do you hate my guts now?

I don't blame you, but let's really quickly touch on some terms that come up when this topic of programming language, uh… "portability," is discussed.

After all, you probably don't want to quote a fake bible tale when trying to describe something technical. I still don't trust you to even *attempt* to describe anything technical, but if you need to, this part should help more.

Machine Language: The extremely specific instructions to a computer's CPU that tells it how to process the 0s and 1s, perform mathematical operations, and that kind of confusing shit.

Assembly Language: A type of language that's a *little bit* more understandable than machine language. An assembly language creates the machine language, on our behalf, for a particular CPU architecture. It's basically shorthand for the machine language. Stay away.

Programming Language: The languages that humans in the modern-day use to write software. Like assembly language, it compiles to the right type of machine code based on the type of Processor you're targeting.

But rather than being an obtuse, impossible to understand piece of garbage like Assembly Language (seriously, google it), we get to use relatively easy to write and comprehend programming language. The programming language's compiler takes care of getting us down to machine code!

Java's selling point

Java programmers don't need to worry about writing for a Dell, or an HP, or a MacBook computer. They write for a make-believe computer called the Java Virtual Machine (JVM).

The JVM acts as basically the ultimate translator for us; it can speak the machine

The JVM acts as basically the ultimate translator for us, it can speak the machine language for pretty much any kind of processor. We just give it the compiled Java code, and it takes care of the rest.

So, Java is a programming language that replaces the practice of “compiling to *machine* language” with “compiling to ***Java Virtual Machine*** language.”

What is compiled Java code then, if it’s not machine code? I mean, what language does the Java Virtual Machine (JVM) understand?

It’s called ***bytecode*** . Any source code you write in Java that wants to run on the Java Virtual Machine needs to be compiled to “bytecode” first.

No, you don’t need to worry about what bytecode actually is. It’s just the result of compiling a Java program. C++ might compile to machine code, but Java is compiled to bytecode.

So, it be like this:

1. You write source code in Java.
2. The Java Compiler “compiles” or translates the source code to *bytecode* .
3. The Java Virtual Machine (JVM) running on your computer executes that *bytecode* .
4. That same bytecode can run on a billion different computers as long as they have the JVM.

Got it?

If you do not get it, please mail me a letter: 1600 Pennsylvania Ave NW, Washington, DC 20500.

(Note: this is not my real address).

Fun fact!

Some compilers of Java are written in Java. Many parts of the Java language are also written in Java. Try and wrap your head around that shit.

What’s in the box?

Oracle is the company that maintains Java. They provide us developers (that includes you now) with this majestic programming language and its related tools in the form of the Java Development Kit (JDK). What’s included?

- The Java Compiler, so you can compile the programs written in Java to bytecode, for the virtual machine to execute.
- The Java Virtual Machine (JVM), so your computer can run programs written in Java.
- The Java Standard Library, so you don't need to reinvent the wheel to do common things many other Java programmers do.
- A whole bunch of other shit.

But where's the Java programming language itself? How do I install it?

The Java language is just a bunch of rules, really. It's how we can type words and characters in a combination that the compiler will understand. You don't install Java the programming language. It'd be like installing English before you can write your essay.

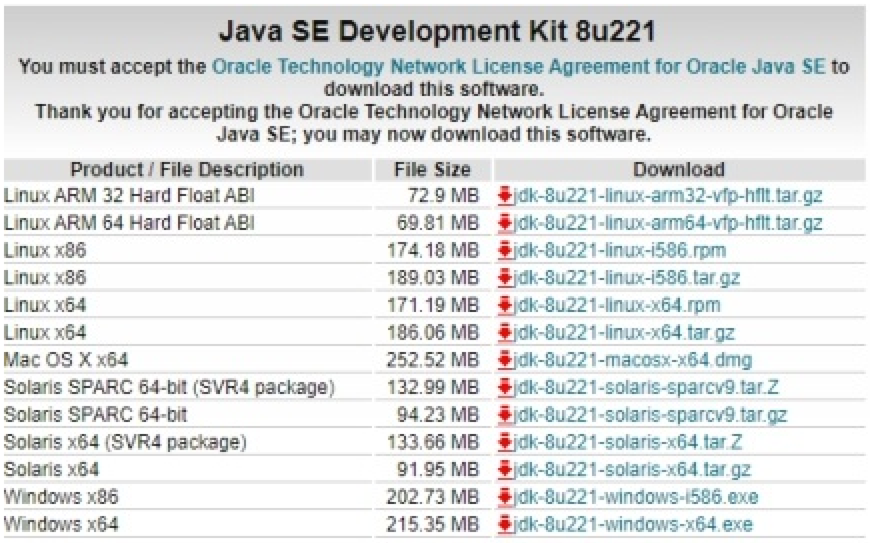
You can write a Java program in Notepad, save it as a file with the name "Hello.java", then compile it and run in on the JVM no problem.

Use your Googling skills to download the JDK from Oracle's website (choose whatever version makes you happy). Yes, Google, the search engine. Get comfortable with it.

Search "Download JDK" or something like that. You want the JDK, not the JRE, or any other thing.

When you get your hands on it, make sure you acknowledge their legal agreements, or they'll put you in jail for life.

You should see something *like* this:



Java SE Development Kit 8u221

You must accept the Oracle Technology Network License Agreement for Oracle Java SE to download this software.
Thank you for accepting the Oracle Technology Network License Agreement for Oracle Java SE; you may now download this software.

Product / File Description	File Size	Download
Linux ARM 32 Hard Float ABI	72.9 MB	jdk-8u221-linux-arm32-vfp-hflt.tar.gz
Linux ARM 64 Hard Float ABI	69.81 MB	jdk-8u221-linux-arm64-vfp-hflt.tar.gz
Linux x86	174.18 MB	jdk-8u221-linux-i586.rpm
Linux x86	189.03 MB	jdk-8u221-linux-i586.tar.gz
Linux x64	171.19 MB	jdk-8u221-linux-x64.rpm
Linux x64	186.06 MB	jdk-8u221-linux-x64.tar.gz
Mac OS X x64	252.52 MB	jdk-8u221-macosx-x64.dmg
Solaris SPARC 64-bit (SVR4 package)	132.99 MB	jdk-8u221-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	94.23 MB	jdk-8u221-solaris-sparcv9.tar.gz
Solaris x64 (SVR4 package)	133.66 MB	jdk-8u221-solaris-x64.tar.Z
Solaris x64	91.95 MB	jdk-8u221-solaris-x64.tar.gz
Windows x86	202.73 MB	jdk-8u221-windows-i586.exe
Windows x64	215.35 MB	jdk-8u221-windows-x64.exe

Yeah Java 8 is old boomer shit by now, but it suits me well enough right now.

Anyways if you’re on Windows, grab the Windows x64 one. If you’re on Mac, grab the Mac one. Run the program that is downloaded. Java is now installed; ba-da-bing, ba-da-boom.

Integrated development environments

Mechanics have toolkits. Carpenters have toolkits. Strippers have cool tits, I mean toolkits, too.

We programmers also have toolkits to help us do our work. The big shebang ones are called ***Integrated Development Environments (IDEs)*** that try to give us everything we need, all-in-one, to write our software.

Microsoft Word is like the IDE for writing papers, essays, novels, or mostly anything involving word processing. It’s got tools for tables, paragraphs, text color, text size, indents, lists, and holy shit so much that you can never find when you need it.

You do ***not*** use Word for writing code. IntelliJ and Eclipse are examples of IDEs for writing Java code.

IDEs like IntelliJ help us write the source code, compile it, run it, share it, test it, rewrite it, you name it. They’re vital to be a productive member of the programming society.

Which one should you use?

IntelliJ . Google it. Download the community edition. Move on with your life.

It’s *raw* !

While I do think it’s worthwhile to know how to compile and run Java code manually, I don’t think it’s a good fit for this book. But I’ll point you in the right direction.

From the command prompt…

> javac HelloWorld.java

Would create a bytecode file called HelloWorld.class

> java HelloWorld

Would then run that program using the generated bytecode.

Read more:

- Some guys' blog post on the command line
- Oracle's documentation
- WikiHow because why not

Or don’t. I don’t care. You really don’t need to stop and read more on this topic unless you’re paranoid about learning how to compile & run Java code at a very low, manual level.

Some would consider this part to be the “basics,” but I don’t fully agree. Sure, it’s basic, but does a beginner need to understand it to *start* coding in Java? To get their feet wet without hating their life?

Nope. IDEs.

CHAPTER 2: HELLO, WORLD!

For their first program, most people do something called “Hello, world!” Here’s how you’d do it.

```
print ( "Hello, world!" )
```

Done! Hah, just kidding. That’s Python. We’re in Java land, which means we need to do 2x the work to get the same fucking result.

Okay, here’s what you do. Create a file called *HelloWorld.java*

Put this inside of it:

```
public class HelloWorld {
    //Called when the program is run
    public static void main (String[] args) {
      System. out .println( "Hello, world!" );
   }
}
```

Compile it.

Run it.

The program spits out “Hello, world!”

Wow. Great. That was both uninteresting and didn’t teach you much. I kind of hate this program but it’s a rite of passage, in a sense.

There are a *few* things I want to highlight. Not with a highlighter, just with words.

First, in English, we put periods at the end of our sentences. You’re don’t want to do that in Java, as a period is *very* different. Instead, use a semicolon.

This is a semicolon:

;

Yeah, it’s that thing you never know how to use in English; does it go between two sentences, before a list, between some sentence fragments? Wait I just used one; was that correct? Oh God.

It’s easier in Java. Your Java “sentences” that do things are called ***statements*** and they all end with a semicolon. Don’t worry, if you forget one, Java will vomit everywhere, and you’ll know you fucked up.

Oh, and those double slashes *//* ?

Anything you put after those is ignored by the Java compiler. You can write anything you want at all. It’s called a ***comment*** . If it you want to type a novel on multiple lines, you use a single slash followed by an astric, like this: */* My Comment */* .

*/**

Boy this book is going swimmingly and there’s no way I’ll lose interest or the author will run out of steam at any point in the near future.

**/* .

No comment

Bad programmers write code that’s impossible to understand and explain it with these “comments.” Even worse programmers write horrible code but don’t even have the courtesy to leave some fucking comments.

Try to be the kind of programmer that makes your code readable without comments, but if you need to resort to them, go for it.

Chapter 2, over! Wow, that was easy.

CHAPTER 3: OBJECTS

“Ryan used me as an object.” – Kelly Kapoor

A lot of Java courses and tutorials wait quite a bit to talk about this, but I’m going to do it immediately.

You see, Java is so, *so* , ***so*** devoted to the idea of “classes” that you literally cannot have Java code unless it’s inside a class. So waiting any further will likely just confuse you. Here’s the skinny:

A ***class*** is a blueprint for something that can exist in the Java universe.

- It has properties, which describe what it *is* .
- It has methods, which describe what it can *do* .

An ***object*** is an *instance* of a class. Meaning, it’s a class brought to life.

For example, maybe there’s a class called **Douchebag** . It’s a template for how a Douchebag might behave.

You could create 4 different objects from that Douchebag class, each named after your middle school bullies.

I’m going to throw examples at you until you get a basic idea:

Número Uno

I write a class called **Airplane** .

- Its properties are *name* , *speed* , and *weight* .
- Its methods are *takeoff(), land(), crash(),* and *burn()* .

I create an instance (an Object) from the Airplane class.

- I give it a *name* of “Crash United”, set the *speed* to 420, and *weight* to 10000.

- I tell the plane to *takeoff()* and *land()* . I don’t use the crash and burn methods. You’re welcome, passengers.

Número dos

I write a class called **Monster** .

- Its properties are *name, damage* , and *health* .
- Its methods are *fight()* , and *die()* .

I create two Monster objects of the Monster class.

- The first is named “goblin”, with 10 *health* , and 2 *damage* .
- The second is named “troll”, with 50 *health* , and 6 *damage* .

I have them taking turns *fight()* ’ing each other, which lowers the other monster’s health.

When one of their health reaches 0, I tell it to *die()* . Oof.

Número Tres

I write a class called **Book** .

- The book class’s properties are *title* and *stars* .
- The book class has no methods, as it doesn’t *do* shit.

I write a class called Reader.

- The reader class’s only property is *currentBook* , which is a **Book** .
- The reader class’s methods are *read ( )* and *rate()* .

Whew. Lots of buildup. What can we do with these classes?

Well, maybe I’ll create an object from the **Book** class and set its *title* to “Java for Fucking Idiots”.

I’ll create a **Reader** object and set the name property to “Chad” and set its *currentBook* to the one I just created.

I’ll run the **Reader** object’s *read()* method, then its *rate() method* .

The *rate()* method sets the book’s *stars* property to 5, because Chad really seemed to enjoy it. Interesting.

Número we’re done

Java, and Object-Oriented Design (which it tries to follow like a religion), is a way to model the real world, but in code.

Let’s think of Amazon’s computer systems. The products that you buy every day are objects, I guarantee it. The packages shipped to your door are objects. Each stop the driver makes is an object. Your house is an object. The trip itself is an object.

So what does a program look like in Java? A bunch of objects interacting with each other in meaningful ways to make stuff happen.

In Java land that could be a desktop computer program (each screen is an object, the elements on the screen are objects), a web service (each network request made over the internet is an object), data returned from a database is an object, you name it.

And while I can’t claim that every Class you write in Java will be based off something in the *real* world, they will be *nouns* that have properties and methods.

Welcome to object-oriented programming! Enjoy your stay…bitch.

Hello, world?

Now, let’s back up a smidge see how this knowledge applies to the “Hello, World!” Program.

Here it is for those of you with short-term memory loss (probably most of you).

```java
public class HelloWorld {
    public static void main (String[] args) {
      System. out .println( "Hello, world!" );
   }
}
```

Let’s see…. there’s a class called *HelloWorld* . There’s a method called *main()* . And…well….

Uhh…

Um.

:thinking: emoji

Hm… what *does* "Hello, World!" have to do with object-oriented programming?

Spoiler: it really doesn't… no objects here.

It's demonstrating one of the few concepts in Java that are *not* object-oriented: *static* methods. Ones that exist outside the realm of objects. God dammit Java. Why can't your "Hello, world!" program be less stupid?

But I should calm down. It's bad for my health. Let's talk about what's really happening in the main method.

Main method

"Who do you think you are? What gives…what gives you the right?" – Michael Scott

The ***main method*** is the entry point to the program; when the program is run, the main method is run.

The line of code that has the words "Hello, World!" is inside the main method. *Anything* you put between those curly braces {} is inside the main method, actually.

```
public class HelloWorld {
    public static void main (String[] args) {
      System. out .println( "Hello, world!" );
   }
}
```

Why is there a "main" method? What makes it so special? The main method exists because *something* needs to run when the program starts.

Remember the whole "which came first, the chicken or the egg" dilemma?

- Some say the chicken came first, because all eggs come from chickens.
- Some say the egg came first, because all chickens come from eggs.

The answer in Java is that *neither* of those objects came first. The main method did.

Nothing can exist without the main method running first, outside of the realm of objects.

This is what that static bullshit means. ***Static*** properties and methods exist *without* objects.

"No object, no problem." – the static keyword

Alternatively…

"Don't make things static unless you have to because objects are good." – me

If you're not 100% on this yet, don't worry about it too much. Only a little bit.

Okay , you might be thinking. But what about that other stuff in front of the word *main* ? ***public*** ? Or ***void*** ? This is just confusing.

I will cover this more thoroughly later in the book. But I'll briefly answer this now as well because I hate when you want to know something, and the author is just like "hold on we'll get there." Bitch I'll tell you now.

- Putting ***public*** in front of something says that all the other classes, even in different folders, can see it. You could also describe some of your code as ***private*** if it is going to the bathroom and needs privacy or something, but not the main method! It's always out in public!

- Putting ***void*** in front of some code means that it does some cool shit but doesn't report back. No one asks the main method to say if things went well or not, or for the value of some math calculation. Main method doesn't care if anyone knows.

Got it? If not, read those explanations one more time.

…Okay, fine you're not going to. I'll summarize even more briefly:

The main method is how Java begins running your program.

The main method is ***static*** because *something* needs to exist outside of the realm of objects. And the developers of Java said so.

The main method is ***void*** because it doesn't need to provide a value to whoever is running it. And the developers of Java said so.

The main method is ***public*** because it will be called by someone outside of this folder, namely Java. And the developers of Java said so.

Declaring a class

Let's leave "Hello, World!" in the dust for now where it belongs, and I'll explain *in detail* how you declare a class:

```
class Book {

}
```

Neat! Mission accomplished.

In fact, you could even do this:

```
class Book {}
```

Java doesn't care about you pressing the enter key (creating a new line) or pressing the space bar (creating whitespace).

In general, though, you don't put curly braces on the same line like that, though. There's no real estate market in Java land, so you can take up the space you need to make your code readable.

But congratulations, you have a class. Proud of you!

Oh, one more thing: they are *always* capitalized like that. If it needs to be more than one word, you capitalize the first letter of those words and smush them together. For example:

```
class FantasyNovel {

}
```

Properties

Okay, so we have a basic class declared.

However, classes are pretty useless when they're empty like that. Let's add a ***property*** .

```
class Book {
   String title ;
}
```

What would properties be in the real world? No, not the Monopoly kind of properties. Let's ask the dictionary.

Property: “an essential or distinctive attribute or quality of a thing.”

Yep. That’s about right in Java too.

We call *title* a property because it pertains to a class. It’s the *book’s* name. It doesn’t exist without the book. We can add some more properties:

```
class Book {
   String title ;
   String content ;
   String[] chapterNames ;
    int bookmarkedPage ;
    double averageRating ;
    boolean hasSequel ;
   Author author ;
}
```

The book has a title, its actual content, chapter names… a number representing the page that the reader bookmarked… a rating to show in a store…

Anyway, starting to get the big idea? A class can have information associated with it called properties.

And just to remind you – those properties will be specific to each object created from this class. I could create 500 books, and they’d each have a different *title* property. I’d also probably write a good one, eventually.

Reading and writing 101

In Java, you can interact with those properties using a period. It’s called “dot notation.”

Let’s say I have an instance of the **Book** class called *bestBook* . This is an actual Book *object* , created from the Book class.

Here’s how I could play around with its properties.

```
bestBook. title = "How to Become a Fancy Waiter in Less Than 20 Minutes" ;
bestBook. content = "Forget everything that isn’t fine dining. And breathing."
bestBook. averageRating = 5.0 ;
System. out .println(bestBook. title );
```

The result? The program spits out the text:

“How to become a fancy waiter in less than 20 minutes”

The dot, or period thing, is like a window into the object’s soul. Put the dot after the name of your object, and access as much as you can see with it.

Methods

Remember the main method? Well, it’s called the “main” method because it’s the big important one that runs first. But you can have more than one.

In fact, it’s highly recommended to have more than one method. Splitting your code up into many smaller methods makes it easier to read and understand what a program is doing.

We just briefly touched on properties, which are pieces of data that are related to the object. The difference between properties and methods?

- ***Properties*** are what an object *is* .
- ***Methods*** are what an object *does* .

“But…but Steve. That doesn’t really explain what a method is in *Java* . What are you, an idiot?” – you

Okay maybe not you. You probably wouldn’t remember my name.

Look, we’re early on in the book. There’s a lot of material to cover, and I don’t want to get into the *weeds* (blaze it) of methods. I’ll give you *one* example of what they look might like in Java.

```
class Video {
    void resume() {…}
    void pause() {…}
    int getLikesCount() {…}
   String getTitle() {…}
   Video[] fetchRecommendations() {…}
    boolean isPaused() {…}
}
```

Just like with properties, you can use dot notation to touch its shit.

```
bestVideo.resume();
bestVideo.pause();
```

I just resumed and paused the video really quickly. Hopefully that didn´t break it and cause it to buffer for an eternity…

Anyways, I realize I left those methods empty. Maybe those methods are modifying properties (which I didn’t type out) of the Video object.

The great thing about objects and methods is that you’re dividing up the code’s responsibilities into manageable chunks.

The *pause()* method doesn’t need to know how to *resume()* .

The *fetchRecommendations()* method takes care of finding “related” videos that YouTube wants to shove down your throat; *getTitle()* don’t care about that!

Good programmers keep their methods small, well-named, and straightforward.

But that’s enough on methods for now – there’s a lot of very basic Java concepts we need to cover before we dig much more into this. Onwards and upwards!

CHAPTER 4: VARIABLES

"Not enough variables... hmm... not nearly enough variables." - Heimerdinger

Okay, it’s time to dig into the nitty gritty of learning the language. Get your big boy (or girl) pants on, cause shit is about to get real.

Let’s talk variables.

Ever take algebra in middle school? Seen something like this?

$7 = 3 + x$

Solve for x, right?

Variables are *kind of* like that.

Here’s an example

```
int myFavoriteNumber = 7 ;
```

Later if we use the *myFavoriteNumber* variable it would be equal to 7.

So***, variables*** are human-readable things that are given values. The short-term memory of your computer says “OK, I’ll store that value for you. Use that variable name again when you want to know that value, or change it.”

A person types “Please remove me from this distribution list” into Microsoft Outlook. The program would represent that data as a variable.

Another person opens the Calculator on their computer and presses the number 3. That’s another variable. A different kind of variable, but one nonetheless.

Your Veterinarian’s computer system checks if you have used them before; this yes/no, or true/false value can be represented by a variable, too.

Variables are *eeeeeeverywhere* . Create them, store them for later, change them, send them to your friends, get attached to them, cry over them. They’re wonderful.

Why do we need variables?

We cannot know the future, and neither can our programs (yet). Our code might need to respond to the user's entered account name, the time of day, the amount of items that are in inventory, the information scanned in a barcode, the breed of dog in a cute picture, the GPS coordinates of your phone, you name it.

Things that we do not know *now* , but our program can find out while running ("at runtime"), are good candidates for variables.

OK, now let's make some.

Making variables

You can ***declare*** a variable, basically telling the compiler "hey, this is something I'm going to use later, OK?" like this:

```
int numberOfStarsToAwardThisBook;
```

Variable declared. Hooray! However, it doesn't have a value.

We can fix this later with a new line of code that ***initializes*** the value:

```
numberOfStarsToAwardThisBook = 5 ;
```

Alternatively, we could have just done it from the start (declaring *and* initializing):

```
int numberOfStarsToAwardThisBook = 5 ;
```

Naturally, Amazon does not have code that makes *every review 5 stars* built into their software. We would likely show the user a screen to make a selection, then set this variable equal to the # of stars they selected afterwards.

Data types

What was the "int" part for?

int is short for integer. Integers are whole numbers. This is honestly elementary math but as I've said before… minimal assumptions.

These are whole numbers: 4, 8, 15, 23, 42, -15, -214

These are *not* whole numbers: ½, ¾, 0.86, -84.97, 0.01

You can put the word *int* before a variable to specify that it is a whole number. That's the ***type*** of variable it is.

Java is absolutely paranoid about these "types" and if you don't specify one, the compiler is going to be very upset with you. In fact, *every* variable in Java must have a declared type.

Here are some the different kinds of variables that you can make.

Note: *All* data types that have to do with numbers support positive *or* negative values.

byte: A tiny number (-128 to 127) typically used for dealing with files.

int: a whole number that can store quite large numbers. *Very* common.

short: a smaller int tha- who gives a shit, don't use these.

long: an integer that can be hyuuuuuuuge.

double: a number that can have decimal points, like 4.20, or 6.90.

float: a stupid and useless version of double.

char: a single character inside single quotes, like 'g' or 'f'.

boolean: true, or false. Nothing else. The data type of Siths.

You can Google "Java data types" or click this link to see all of your options.

What are thoooose?

They're called ***primitive*** data types; not because they're illiterate and don't bathe, but because they aren't objects.

Imagine the universe in Java is not composed of protons, neutrons, and electrons at its lowest level (hope you graduated from middle school!), but primitive variables.

A primitive variable does not have properties. It does not have methods. It is not a blueprint from which we can create an object. It cannot reproduce.

You cannot get more basic than a primitive. I mean, look at them. *char* is a single letter. *double* /*float* /*int/short* are numbers. *boolean* is true or false.

Note that all the examples I listed start with a lowercase letter; this is the tell-tale

Note that all the examples I listed start with a lowercase letter, this is the tell-tale sign of a primitive variable.

Every class or object that stores or processes data, somewhere deep inside, has data represented by primitives. I think. Probably.

Any piece of data that you see represented as an Object is using primitives under the hood. For example, there's a class in Java that's extraordinarily common called a String. See the capital S? Not a primitive!

A note on Strings

What's a ***String*** , you ask? Literally anything inside of two double quotes.

- "Hello, world!" is a String.
- " " is a String with a space in it
- "" is an empty String
- "123" is a String
- "598.23" is a String.
- This entire book within some double quotes is a String.

Secretly (okay not very secretly) it's a bunch of *char* variables tied together. The String class takes care of all that for you.

We have an entire section of this book dedicated to Strings, how to deal with them, and all kinds of garbage like that later in the book. But hopefully this will tide you over.

Experimenting

I recommend that you experiment with declaring variables, giving them values, and seeing what happens. You can even boot up your stupid "Hello, World!" program and put it inside the main method.

```
public class HelloWorld {
    public static void main(String[] args) {
        int myNumber = 420 ;
        boolean shouldBlazeIt = true ;
        char charmander = 'C' ;
        double moneyInTheBank = 500.36 ;
        long speedOfLight = 299792458L ;
        String bestAuthor = "Steve" ;
    }
}
```

Feel free to do something that won’t work. Push the limits of the Java compiler. For example, you could write this line of code:

```
int wellThisIsJustWrong = false ;
```

Then compile and run the code. It will throw a conniption, send a report to NASA, and inform your internet service provider to immediately cancel your plan.

CHAPTER 5: OPERATORS

“Hello operator? Get me the navy!

[You’ve reached the Navy’s automated phone service.]

Squidward! The robots have taken over the Navy!

Not the Navy!!!” – Spongebob Squarepants

OK, so now you know the basics of variables. Hopefully.

Now let’s look at ***operators*** , or ways that we can interact with our variables.

What’s an operator?

They’re special symbols that do special things. Maybe you want to add two values. Or see if two values are the same. Or check if a number is greater than or equal to another.

Or, you want to do all the above. Operators ahoy!

Note that operators are not variables. They’re not methods, they’re not objects, they’re not data types, and they’re just *not* that into you.

They’re like the buttons on your calculator that aren’t the numbers.

The math operators!

Let’s get it started!

- Add numbers with **+**
- Subtract numbers with **–**
- Divide numbers with **/**
- Multiply numbers with *****

This is just as easy it sounds for 99% of scenarios.

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

…x becomes 4

```
int y = 4 * 9 ;
```

…x becomes 36

But what happens where we start *combining* our operators? How do we know which ones are more important and go first? Do we multiply, then add, then subtract? If only there was some sort of pneumatic device to remember…

Well, guess what?

PEMDAS is back!!

Order of operations

An ***operation*** is one little chunk of math shit. Your waiter probably waits on tables (each one an operation) in a certain order, by the time in you arrived, how much they like you, how much they expect you to tip, where you're located in the restaurant, etc.

Math has its own order of operations: PEMDAS.

P arentheses () -> **E** xponents[3] -> **M** ultiplication & **D** ivision */ -> **A** ddition & **S** ubtraction +-

There is good news! You're beautiful.

No, but there are *no* exponents in Java. You literally just need to remember that stuff in parentheses happens first, followed by multiplying/dividing things, followed by adding/subtracting.

And if there's no combination between those categories you just go left to right.

Trust me, it's super easy.

Into the examples, here we go!

Example one!

```
int x = 5 + 3 - 7 ;
```

x is 1.

Why? No need to worry about the PEMDAS rules, it's just addition/subtraction.

5 + *3* is *8* . And if we subtract *7* from that, we get *1* .

Example two!

```
int y = 5 - 7 * 6 ;
```

x is -37.

Why? The multiplication must happen first, because PEMDAS says so.

7 * *6* is *42* . And if we subtract from 42 from 5, we get -37.

Example three!

```
int z = 28 / ( 3 + 4 );
```

x is 4.

Why? Even though it's addition, *3* and *4* must be tackled first because they're inside parentheses.

3 + *4* is *7* . Then we can divide *28* by *7* and get *4* .

Most math in programming, at least for 90% of developers, is extremely basic; add two values, multiply two values, etc. So don't expect to have to worry about this "order of operations" very often.

Modulo

Also… I debated not mentioning it, but fine, there's another one. It's called the ***modulo*** operator and it's stupid. It uses a percentage sign (because fuck you that's why).

```
int remainder = 5 % 2 ;
```

remainder is set to 1. Why? Because a modulo performs division, then gives you the ***remainder*** of that division operation.

The only scenario I've really seen for this is to check if a number is even/odd. For example, if a number "*modulo 2*" is equal to *0* , that means it's even. Because it evenly divides by 2 without a remainder.

```
boolean isEven = myNumber % 2 == 0 ;
```

That's all I'm going to say about it.

Your examples suck

Well, Java would not *actually* be used for the silly examples above, OK? Yeesh. Just introducing some stuff.

See the numbers 5, 3, and 7 in the first example?

```
int x = 5 + 3 - 7 ;
```

Because they're directly written into our source code, those numbers are ***hard-coded*** .

I mentioned this earlier, but no one would ever write Java code like this. If they want *x* to be 1, they'll set it equal to 1 to begin with.

We use these math operators (+ - / *) to calculate things we don't know the value of right now, but we will when our program is running!

Maybe you're making a shopping cart website. Your program might multiply (*) the current price of a product by the quantity the user selects to get the total price. Or, your program might decrease (-) the total by one if the user removes it from their cart.

The logical operators!

These ones are easy peezy.

They're basically the Java version of these words in English:

- and (&&)
- or (||)

No, that's not art of a bunch of buttocks. It's the new operators inside of parentheses.

Or

Let's create some *booleans* ! Remember that data type? A *boolean* value is either true or false. This data type goes hand-in-hand with these "logical" operators.

```
boolean imBored = true ;
boolean imHungry = false ;
```

```
//True
boolean imGoingToEatFood = imBored || imHungry;
```

See how the ***or*** operator **||** works? “I am going to eat food if I’m bored ***or*** if I’m hungry.”

If *either one* of those conditions are true, then *imGoingToEatFood* is set to true.

More examples…

- I’m making a text-based adventure game. If the user types in “exit” ***or*** “done”, I exit the program.
- I’m making a program to monitor the stock market (this is where money lives and dies). If the price of a stock I own goes severely up, ***or*** severely down, I send out an email alert.

In fact, Java won’t even look at the right side of the || operator if the left side is already true; there’s no point! Java’s like… why check *imHungry* if I already know *imBored* is enough to warrant eating food?

And

That’s not true when it comes to the ***and*** (&&) operator. *Both* need to be true for the expression to evaluate to true.

```
boolean wellRested = true ;
boolean motivated = false ;

//False
boolean goingToWorkOut = wellRested && motivated;
```

Even if I’m well rested, if I have no motivation at all, I won’t work out. And even if I were motivated to work out, if I have no energy due to lack of sleep, I’ll probably pass as well.

More examples…

- I’m writing an app where the user must log in using their account information. I only try to sign them in if they’re entered a username ***and*** they’re entered a password.
- I’m creating a lighting system that automatically turns off the lights if the lights are currently on ***and*** there has been no one in the room for 5

minutes.

Seem too easy? It's "logical", right? Get it? Logical? Because they're called the logica-

Oh, and just like with the math operators; no, you shouldn't create variables that are always true, or always false. They're variables. But I'm trying to get the concept across right now, kay? More on this when we get into conditionals and if statements.

The assignment operator!

"Now, are you men ready for your super…special…. secret…ASSIGNMENT!?!?

The two of you are to paint the inside of me house!!

Yayyyyy!!" – Spongebob SquarePants

Yeah, you've already seen this thing (it's the equals sign **=**), but we're going to cover it to clear up a misconception that beginners might make.

Think back to your algebra class in school. You probably don't remember. No, it's not the one with the triangles… does this ring a bell?

$x = 7$

How would you say that out loud? Probably something like this:

"x is equal to 7".

You can do something very similar in Java. Check out this line of code:

```
int x = 7 ;
```

It's the same thing as before, right? "x is equal to 7"??

FALSE!   INCORRECT !   NEGATIVE!

Wanna see why? What if I show you this?

```
int x = 1 ;
```

Okay, x is 1…

```
x = x + 1
```

Hmm

Hmmm…

Okay, now x is equal to…. x + 1?? Wait… if x is 1, then that means 1 = 1 + 1??

Does that mean that 1 = 2???

Hopefully your mind has awoken from its slumber and is screaming at the page: “THAT’S NOT POSSIBLE! THAT MAKES NO SENSE! WHY WOULD THEY MAKE *BRAN* KING?”

= is not “is equal to”

That’s because a single equals sign in Java is the ***assignment operator*** . Here’s how it works:

The thing on the left is given the value of the thing on the right. You can say it like this instead:

```
int x = 7 ;
```

“Assign to x the value of 7”

```
x = x + 1
```

“Assign to x the current value of x plus 1”

Guess what? The value of x is 8 now (shocked emoji).

It seems like a subtle distinction, doesn’t it? “Assigning something a value,” versus saying that “something is equal to something else.” But this concept is a mystery mouseketool; a surprise tool that will help us later.

The relational operators!

Remember the carrot symbols (< and >)? Maybe you were taught to think of them as alligators; the alligator’s mouth points to the bigger number? Maybe? Anyway…

Here’s some true statements in regular math:

2 < 3

2 is less than 3.

5 > 4

5 is greater than 4.

Cool. Now, *this* is wrong.

5 > 5

~~5 is greater than 5.~~

They're the same value; 5 is not bigger than 5. FALSE, you must be screaming right now!

Now, what if we wrote *this* down?

5 ≥ 5

This is true. 5 is indeed greater than *or equal to* the value of 5. The statement 5 ≥ 4 is also true.

Now that I've summarized the entirety of your 6th grade algebra class, let's see how this works in Java.

Relations!

```
//True
boolean itsSmaller = 2 < 3 ;

//False
boolean itsBigger = 3 > 4 ;
```

I know what you're thinking. I've got this in the bag. Those operators are simple! Comparing values to see if one is bigger or smaller than another? Hah! What a joke. This guy must think I'm a complete fucking idiot.

Well, yes, that's the point.

But there's some more curveballs coming your way.

Remember the "or equal to" bit I mentioned up above? In regular old math, that's when you underline the carrot thing, like this: ≥

You can combine an equals sign with one of the comparison operators to form the same thing; a more craaaazy kind of relational check:

```
//True
boolean itsBiggerOrTheSame = 5 >= 5 ;

//False
boolean itsSmallerOrTheSame = 3 <= 2 ;
```

Sure, putting the equals sign *after* the carrot isn't as sexy as underlining it like you do in algebra. But that's how you do it in Java.

You can keep experimenting with these but you're probably already getting the hang of it.

Yeah, I guess you could say that's a compliment. Keep up the good work buddy. Isn't it amazing how easy programming is?!?! You're so smart! And a wonderful reader! Also, did you award this book 5 stars yet?

The comparison operator!

What happens if you put two equals signs next to each other?

:thinking emoji:

The answer is that the mommy equals sign and daddy equal sign come together and form a new operator; the ***comparison operator*** !

This is where my tangent about how a single equals sign (=) means "assignment" and *not* "is equal to" really comes into play. So I hope you held on to that mystery mouseketool!

Okay, let's say that we don't want to *assign* a value to a variable. Instead, we want to check if a variable *is equal to* a value. This is super duper, *extremely* common by the way.

Enter the comparison operator; two equals signs, back to back.

```
//True
boolean isSameCharacter = 'c' == 'c' ;

//False
boolean isSameNumber = 5 == 4 ;
```

The Booleans are back, baby!

Are *'c'* and *'c'* the same character?

Are *5* and *4* the same number?

Those *booleans* are accordingly set to true or false.

Preeeeettttyyyy easy right? Just don't panic your little head over all the equals signs.

Yeah, there's still an equals sign to assign the value on the right to the variable on the left. Just like before. But that's not related to the == comparison operator on the right side.

If it really confuses you, use some parentheses.

```
boolean isCartEmpty = (numberOfItems == 0 );
```

The first one (=) is the assignment operator. The next two (==) form one comparison operator.

Changing requirements

I changed my mind. I want to check if a variable is *not* equal to something.

Here, we split our comparison operator == in half, and we're so shocked by the gruesome fatality that has befallen our beloved operator, that we put an exclamation point.

!=

Basically, "not equal to."

```
boolean isGoodCharacter = character. name != "Brandon Stark" ;
```

A character is a good one if its name is *not* equal to Brandon Stark.

```
boolean haveGumDropButtons = numGumDropButtons != 0 ;
```

We have gum drop buttons if the value is *not* equal to 0.

```
boolean fallingWithStyle = flying != true;
```

We're *not* flying, we're falling. With style.

Two more:

```
//True
boolean isDifferentCharacter = 'c' != 'b' ;

//True if it's not 5
boolean isValidInput = userInput != 5 ;
```

So there you have it. The "not equal to" comparison operator. Want to do something when something else is *false* ? Welp, there ya go.

Fancy talk

"Do you mean she puts on airs?

I guess so.

That's just fancy talk. If you wanna be fancy, hold your pinky up like this. The higher you hold it, the fancier you are!" – Spongebob Squarepants

Sometimes we are so lazy that we can't be bothered to type things out in Java. That's not surprising; programmers are inherently very lazy.

Let's say you run a zoo containing only giraffes (that's fine in my book, by the way. Haha, my book. That's funny cause we're in my book right now. Get it? Isn't that hilarious? Anyway, I hate when books have entire paragraphs inside parentheses. You forget what the original sentence was even saying); that could be represented with this line of code:

```
int numGiraffes = 23 ;
```

Now, let's say you just received a new giraffe at your zoo. Hooray!!! Maybe this line of code is executed:

```
numGiraffes = numGiraffes + 1 ;
```

Wonderful! But wait, papa Java is knocking at the door. There's a shorter way to write this for essentially no reason! How? Like this!!

```
numGiraffes++;
```

Wow! That's shorthand for the same line of code, saying "Take the current value of *numGiraffes* , increase it by 1, and set *numGiraffes* equal to that value."

How insanely useful is that?!?! Knocks your socks right off!

Is that only for addition?

Unsurprisingly to those of us with functioning neural passageways in our craniums, there are other fancy talk operators like the above addition one.

A giraffe passes away, decrease by 1…

```
numGiraffes--;
```

Triplets are born, increase by 3…

```
numGiraffes += 3 ;
```

Half of the giraffe population is culled, divide by 2…

```
numGiraffes /= 2 ;
```

The giraffes are cloned, multiply by 2…

```
numGiraffes *= 2 ;
```

This is how we'd say that last one out loud:
"assign to numGiraffes the current value of numGiraffes times 2."

In all honesty, I typically do not recommend using these, except maybe the ++ and -- ones.

I PERSONALLY (*opinion alert!!*) find writing out the actual mathematical operation to be easier to understand. Remember that you should primarily write your code to understood by a human; your future self and other team members will appreciate it.

King boolean

Remember when I went over the exclamation point (!) and how it basically is saying "not"?

So "!=" is checking if a value is "not equal to" another value?

There's an easier way to write this in Java if you're simply trying to flip something from true to false (boolean values), or vice versa.

For example, instead of something like this:

boolean isBad = isGood != true ;

You could simply say

boolean isBad = !isGood ;

The exclamation point goes directly in front of the boolean. Wow again!!

The other operators…

Not even gonna go over them. If you want to learn about bitwise, shift, or the

Not even gonna go over them. If you want to learn about bitwise, shift, or the ternary operators, look for a more advanced piece of material. Or encourage me to write another book. Winky face.

My brain hurts

All those fancy talks are optional, so don't fuss. Plus, if you use an IDE like IntelliJ, it will correct you if you're doing something in a stupid way.

CHAPTER 6: CONDITIONALS

“It’s time to get funky.” – DJ Casper

Most times your code can’t simply run top to bottom, statement after statement, and do anything useful. You need your code to do something different based on what the user typed in, or what products you have in stock, or what version of Android you’re running on, etc.

Behold, the might ***if*** statement:

```
if (numGiraffes > 500 ) {
   System. out .println( "Please send help." );
}
```

That line of code that prints to the system only happens if we have more than 500 giraffes.

Another example:

```
Scanner userInputReader = new Scanner(System. in );
int userInputNumGiraffes = userInputReader.nextInt();
if (userInputNumGiraffes <= 0 ) {
   System. out .println( "Where they at?" );
}
```

That line of code prints to the system only if the user typed in a 0. Or a negative number. I’m going to skip explaining how to get input via a Scanner , OK? Really, really dull stuff.

Anyway, does the *if* part make sense? It’s pretty similar to English, really. *If* this, do that.

Now, what if we *also* want to do something when the user types in an actual value? If you remember the comparison operators, you could add something like this below that conditional:

```
if (userInputNumGiraffes <= 0 ) {
   System. out .println( "Where they at?" );
}
if (userInputNumGiraffes > 0 ) {
   System. out .println( "I shall add them at once." );
}
```

Two *if* statements, back-to-back. Could work, right? But that's stupid. Why check the value of userInputNumGiraffes in two places? Don't do that.

Instead, add an ***else*** block to the original *if* statement, as follows:

```
if (userInputNumGiraffes <= 0 ) {
   System. out .println( "Where they at?" );
} else {
   System. out .println( "I shall add them at once." );
}
```

If the first check (the user inputs a value less than or equal to 0) is *true* , we go inside the *if* block. Otherwise (else!) we go to the *else* block.

Programming is EASY dude.

Oh, and you don't need to put the word *else* on the same line as that curly brace {.

You could press enter and push the *else {* to the next line, like this:

```
if (userInputNumGiraffes == 0 ) {
   System. out .println( "Where they at?" );
}
else {
   System. out .println( "I shall add them at once." );
}
```

You could also put each curly brace on its own line. You'd be committing a Java code style atrocity, but you can do it if you really want to piss me off.

OK, I hope you're with me so far; you can check if something is true or false, then act appropriately using if/else (conditional) logic. This is called ***control***

flow . You’re controlling the flow of the program.

However, there are situations where it’s not simply “do this if *X*, *else* do something different”. You might want to do different things based on how many giraffes there are, for instance.

How do we do that? Well, there’s a few ways to skin this cat….

Multiple if/else checks

How about something like this? Inside the else block, we add another if/else check. Inside that else block, we add another if/else check, and so on. That works, right?

Note that if you’re on a small screen this bit might get messy, but bear with me…

```
if (userInputNumGiraffes < 0 ) {
   System.out .println("Get your hands the fuck off my giraffes." );
} else {
   if (userInputNumGiraffes == 0 ) {
      System.out .println("Sorry, you can cannot add 0 giraffes to the zoo." );
   } else {
      if (userInputNumGiraffes < 3 ) {
         System.out .println("Thanks for donating 1-2 giraffes." );
      } else {
         if (userInputNumGiraffes < 10 ) {
            System.out .println("Thanks for donating 3-9 giraffes." );
         } else {
            System.out .println("How did you have 10+ giraffes?");
         }
      }
   }
}
```

See what’s happening? We print a different sentence out to the console based on the number the user typed in.

For example, that last *if* check that looks for if the value is <10? That only happens if the value is *not* negative, is *not* zero, is *not* less than 3, but *is* less than 10. Read through it top to bottom a few times to logically follow the logic.

That code looks good right? Pretty? Easy to read?

That code looks good right? Pretty? Easy to read?

No, it looks like absolute shit. It's an eyesore. Hard to read, hard to understand, and unpleasant to look at. Here's the good news; you can make this **1000x** easier if you simply say, "*else if* ". HUH??

How is this possible? Here's why (if you don't care, it doesn't really matter); the curly braces between those chained "*elses* " and "*ifs* " aren't doing anything, nor are putting them on different lines.

It functions *exactly the same* as doing this:

```
if (userInputNumGiraffes < 0 ) {
   System.out .println("Get your hands the fuck off my giraffes." );
} else if (userInputNumGiraffes == 0 ) {
   System.out .println("Sorry, you can cannot add 0 giraffes to the zoo." );
} else if (userInputNumGiraffes < 3 ) {
   System.out .println("Thanks for donating 1-2 giraffes." );
} else if (userInputNumGiraffes < 10 ) {
   System.out .println("Thanks for donating 3-9 giraffes." );
} else {
   System.out .println("How did you have 10+ giraffes?" );
}
```

Doesn't that look better than your nested if/else statements stretching out to the right, riding off into the sunset?

The 'switch' statement

Let's change up the example, because while I love long-necked creatures that primarily exist to feed lions, I could benefit from a change of imagery.

We're creating a video game. The user presses a button on their keyboard.

If they pressed W, move the player forward.

Else if they pressed A, move the player left.

Else if they pressed S, move the player backward.

Else if they pressed D, move the player right.

Else if….

See? It's like a multiple-choice test!

While we could continue this pattern of doing if/else if, else if, else if, else if checks until our heart dies out, there's an easier way; the ***switch*** statement.

Well, maybe not easier. I personally find it a bit overrated and think it opens you up to making mistakes but screw it, you should know it.

I'll show you both. The way we'd do it with what we know so far:

```
if (keyPressed == 'W' ) {
   player.moveForward();
} else if (keyPressed == 'A' ) {
   player.moveLeft();
} else if (keyPressed == 'S' ) {
   player.moveBackward();
} else if (keyPressed == 'D' ) {
   player.moveRight();
} else {
   player.stopMoving();
}
```

And behold the new and totally correct way to do it, with no mistakes whatsoever:

```
switch (keyPressed) {
    case 'W' :
      player.moveForward();
    case 'A' :
      player.moveLeft();
    case 'S' :
      player.moveBackward();
    case 'D' :
      player.moveRight();
    default :
      player.stopMoving();
}
```

Whoa, crazy! We create a "***switch*** " with the value of the user input, then do something different for each "***case*** ". And no, "case" has nothing do with uppercase or lowercase letters; it's when the switch equals a particular value.

If they pressed in 'W', then the code under the 'W' case is run.

If they pressed ‘D’, then the code under the ‘D’ case is run.

The “***default*** ” is what happens if the user input isn’t *any* of our cases. Maybe they pressed space bar. Or some letter we haven’t accounted for.

Make sense? Compare it with the if/else examples further back.

However, there’s one fatal flaw in the ***switch*** example code (IT’S ACTUALLY NOT WITHOUT MISTAKES, CURVEBALL!!!). What happens if the user presses ‘A’? What direction does the player move?

Think about it. Scroll up/flip back/take a taxi back and look; what could be wrong with a user pressing ‘A’? Or don’t. I’ll tell you anyway.

The player starts to *moveLeft()* but is then immediately told to *stopMoving()* .

“That’s what’s happened?! But why?? The user pressed ‘A’, so it matched the ‘A’ case, right? Move left?” – you, maybe.

Well, the “*default* ” line of code will actually happen every single time, ***in addition to*** *the* “case”…

Because unless you specify otherwise, a switch will look at every single ‘case’, then hit the ‘default’ block of code. That switch won’t stop!

You need to give a very special word to Java here for each ‘case’. It’s called a ***break*** .

```
switch (keyPressed) {
    case 'W' :
      player.moveForward();
       break ;
    case 'A' :
      player.moveLeft();
       break ;
    case 'S' :
      player.moveBackward();
       break ;
    case 'D' :
      player.moveRight();
       break ;
    default :
```

```
    player.stopMoving();
     break ;
}
```

Now when one of the cases is matched, Java will hit the kill *switch* (hehe) when it sees *break* and prevent anything else in the switch statement from running. Java jumps down to any code below the entire switch statement (outside the curly braces {})

Why do we need to do this?

"Because fuck you, that's why." - Java

Now tell me, which is easier to understand, some **if/else** 's, or a **switch/case/break/default** cluster? It's totally your choice. Want to do something different based upon a certain variable's value? Pick the one you like better.

But if you choose a ***switch*** you'd better hope you don't forget your ***break*** statements (and it does happen).

CHAPTER 7: ARRAYS

“Isn’t there anything on TV that isn’t about BOXES?!?!

We now return to Championship boxing.” – Spongebob Squarepants

Pop quiz: What’s the first letter of the alphabet?

Answer: (doo doo, doo doo, doo doo) The Daily Double

Okay, no. It’s actually **A** .

How would we represent that letter in Java? Which data type?

Tick.

Tock.

Tick.

Tock.

Did you guess a ***char*** ? If so you’re right! If not… shucks! I was really rooting for you, dumbo. Here it is!

```java
char firstLetter = 'A' ;
```

Congratulations again.

More than one value??

Cool. Now, what if I asked you, “what are the letters of the alphabet?”

You’d probably start singing the song: “A B C D E F G, H I J K LMNOP…”

So, how would we represent *that* in Java? What kind of variable can represent the entire alphabet?

The answer… is… an ***array*** ! Did you get it? If you did, WAT? HOW? I haven’t talked about arrays yet. You’re a goddamn cheater.

Anyway, Java has this idea of an “array,” which is pretty much a box of variables. Watch, I’ll *spell* it out for you (literally, if you read aloud! Hah! Ahaha! Ahem…)

```
char [] alphabet;
```

Look at that! Those square brackets *[]* are saying that the variable *alphabet* is an array (or box) of characters. Meaning it can be more than one letter. Absolutely mesmerizing. Thanks Java!

I *should* mention that you could also put the square brackets to the *right* of the variable name…

```
char alphabet[];
```

But no one does that anymore and neither should you if you wanna be cool.

Filling the box in one go

But let’s keep going. How do we put things *inside* the box? Well, let’s initialize that variable that we just declared.

```
alphabet = { 'A' , 'B' , 'C' , 'D' , 'E' , 'F' , … };
```

Yes, I got tired of typing. Those three dots (…) don’t mean anything. But do you see what I did there?

Inside curly brackets *{}* (or braces, whatever you call them) I put all the chars, separated by commas. This is how arrays work, basically.

First, you put your variable type (*int, char, double* , etc.), followed by some square brackets [].

Then, if you know the values that go in your array when it’s created, then you can put them in the curly brackets *{}* .

More examples:

```
int [] favoriteNumbers = { 6 , 9 , 420 };
double [] doubleDoubles = { 1.7 , 340.4 };
float [] whyUseFloats = { 50.8f , 129.3f , 603.432f };
```

Cool? Cool.

Making an empty box

That works when we know right away what goes in the array.

However, what if we *don't know* right away all the values that go in the array? Say that three times fast….

Scenario: I want to collect the user's top 10 favorite letters of the alphabet. They're going to type them in one at a time, and I want to keep track of everything they've entered. What do I do?!

Do I call it quits? Do I go home? Well, I'm already at home… so, no.

Nonsense! I can do this:

```
char [] userFavoriteLetters = new char [ 10 ];
```

I know what you're thinking.

"What the hell is that on the right side?

You're supposed to use those… curly brace things when you put the variables in the box!

And they're letters, not numbers? What's the 10 for?? And what does 'new' mean????"

Calm down, Jennifer. I can explain:

When an array is created, Java must know the exact size it is and always will be.

In the previous examples where we gave the array all its values when it was created, Java knew the size, right? Remember this guy?

```
int [] favoriteNumbers = {6 , 9 , 420 };
```

We told Java right off the bat there's 3 things in this array, and here they are.

But when we can't give all the values at the beginning, then Java at least needs to know the size!

That's what the *10* says; there will be *10* things, or "elements" in this array. The "new" part is saying you're making a new array, and is just a required word by Java.

We’ll get more into what ***new*** actually *means* later, but DON’T WORRY ABOUT IT OKAY? CALM DOWN. REMAIN CALM.

Just know that with this line of code, we create a box with 10 empty slots in it. Rejoice, and let’s keep going:

```
char [] userFavoriteLetters = new char [ 10 ];
```

OH, one more thing. Very important.

You can tell how big your Array is using a very special “property” on the array object, ***length*** . Remember properties? Variables that describe an object? Well, arrays are objects so they can have properties, too.

For the *userFavoriteLetters* array, what would the length be? Well, it would be *10* . Try it.

```
System. out .println(userFavoriteLetters. length );
```

It prints out 10. Very useful for later when we do loops, keep this in mind!

Putting shit in the box

“Well, the square brackets start comin’ and they don’t stop comin’ and they don’t stop comin’…” – Smash Mouth

So, we’ve got an array with 10 slots. How do we put something in those slots? No, that’s not a dirty innuendo.

The answer is… we use the assignment operator (that’s the equals sign) combined with square brackets to specify which slot in the box will be assigned a value.

Check this out:

```
char [] userFavoriteLetters = new char [ 10 ];
userFavoriteLetters[ 1 ] = 'Q' ;
```

See what I did there? I said “Put the letter Q in the first slot.”

And you can keep doing this shit:

```
userFavoriteLetters[ 2 ] = 'C' ;
```

[illegible]

Hell, you can even jump right to this:

```
userFavoriteLetters[ 7 ] = 'Z' ;
```

Your array, under the hood, looks like this now:

[, Q, C, , , , , Z, ,]

So we put the user’s favorite letter, Q, in the first slot, then we put….wait. Hold the phone.

Is that a comma *before* the Q? Is Q not in the first slot? The first slot appears to be empty! WHAT’S HAPPENING?!

I’ll tell you: ***arrays start at 0 in Java*** . That’s right, slot 0 is the first position, not slot 1. The correct way to put something in the first slot is:

```
userFavoriteLetters[ 0 ] = 'Q' ;
```

This is incredibly important! Arrays are used all throughout Java, even if you’re not aware of it at times. And by the way, it’s not just papa Java coming up with some weird “start at 0” rule, nearly every programming language does this.

So yes, once your array is created, you can put things into it with some square brackets *[]* .

I’ll show a few more examples for your benefit.

```
String[] words = { "Hello" , "world" };
```

- This is an array of length 2 because there are 2 things in there.
- “*Hello* ” is at position 0 and “*world* ” is at position 1.

```
words[ 1 ] = "universe" ;
```

- Now, “*Hello* ” is at position 0, but “*universe* ” is at position 1.

Getting shit from the box

Getting shit from the box

Pretty much the same way you put things in, with some square brackets [].

```
char firstLetter = userFavoriteLetters[ 0 ];
```

The number provided in those square brackets is called the ***index*** .

In any array where you want to put things in or take things out, you specify the index at which you’re doing your shit.

Let’s check your knowledge here.

If my *char* array has 10 slots in it, how do we get the *last* one in the array? What “index” do we put in the square brackets? Think about it before reading on. Open a window if you need to.

The answer:

```
userFavoriteLetters[ 10 ]
```

…would be absolutely ***INCORRECT*** .

You actually need to put a *9* there to get the last element in the box!! Why? Because arrays start at 0! And if there’s 10 slots, then the last character is at index 9! One before the size of the array.

10 – 1 = **9** .

```
userFavoriteLetters[ 9 ]
```

Returns the last item in the array.

This is a concept that your mind might have a hard time adjusting to. Well, pretty much any new thought or idea is probably a bit puzzling to you. But this one especially.

Just remember, when dealing with arrays you need to mentally subtract 1 from how you’d say it in English.

The item in the 1st position is at index *0* .

The item in the 2nd position is at index *1* .

The item in the 5th position is at index *4 .*

…

BRAIN TEASER TIME

What happens if you ask the array for an item in a slot that doesn't exist? For example, in our 10-slot array, what if we asked for the user's 15h favorite letter?

```
char fifteenthFavoriteLetter = userFavoriteLetters[ 14 ];
```

The answer is your program "crashes." An "exception" occurs. Everything goes to absolute shit.

Ever see a program on your computer or phone or game console just completely close and tell you to press a button to report the error?

There are many ways to prevent this. You could only look inside the box once you've made sure that the box is *at least* that big (using an *if* check). Or do something fancy called "exception handling " which I won't be getting to in this book.

Lists

Java has some extremely helpful classes that we can use in place of arrays.

While I won't be covering them in this book, feel free to take a gander .

CHAPTER 8: LOOPS

“You make a loop de loop and pull
And your shoes are looking cool
You go over and back, left to right
Loop de loop and you pull them tight”

– *Spongebob Squarepants*

Welcome to the section on loops.

Java has 3 kinds of loops. One of them is used all the time. Another is used every once in a while. The other is almost never used. I’ll let you guess them as we go.

You: Hold, on the hell is a loop?

Life. Life is a loop.

You wake up. You get ready for the day. You go to school or work. You come home exhausted. You try to relax if you have time. You sleep. Repeat.

We can do the same thing, slightly less depressingly, in Java code!

Let’s back up to when I introduced arrays with the alphabet.

```
char alphabet = { 'A' , 'B' , 'C' , 'D' , 'E' , 'F' , … };
```

Let’s print them out to the system one by one.

```
System. out .println(alphabet[ 0 ]);
System. out .println(alphabet[ 1 ]);
System. out .println(alphabet[ 2 ]);
System. out .println(alphabet[ 3 ]);
System. out .println(alphabet[ 4 ]);
```

…

Well this is EXHAUSTING. Just like life, am I right my fellow Gen Z-ers?

Wouldn't it be much easier to write this line of code *once* , then run it for each letter of the alphabet? Java is here to save you.

Well, every programming language could save you. But Java can too.

The ***while*** loop: to infinity… and beyond!

Let's say you want to do something *forever* . Here you go:

```
while ( true ) {
   System.out.println( "This is the loop that never ends." );
}
```

The ***while*** loop looks at the condition inside the parentheses () and runs the code inside the curly braces {} if it's true.

Then it goes back to the top, sees if the condition is true, and runs the code inside the curly braces {} if it is…

Then it goes back to the top, sees if the condition is true, and runs the code inside the curly braces {} if it is…

Each pass through those curly braces to run some code is called an ***iteration*** . This loop runs for an infinite number of iterations.

Yes, it's just like when we learned about ***if*** statements and conditionals, except repeating.

```
if ( true ) {
   System. out .println( "This line of code runs once." );
}

while ( true ) {
   System. out .println( "This line of code runs forever." );
}
```

Does it make sense?

The *while* loop, just like an *if* statement, relies on a conditional, boolean value (true or false) to know if the shit inside the curly braces should be run.

And guess what? *true* is always a *true* value, unsurprisingly. So it repeatedly evaluates to true, and runs until the heat death of the universe. Or your computer catches fire.

A less infinite while loop

I have better examples than just an infinite loop in store for you. But, before we start on these better examples, let's create a variable. This variable will be the current index of the array as we "loop through it".

```
int index = 0 ;
```

Remember, in an array of every character in the alphabet, the letter at index 0 is A, index 1 is B, etc.

Now, how many letters are in the alphabet? Don't worry, once you create the alphabet array, you can just use alphabet.***length*** to get the size remember? But it's 26…

So, how can we write a *while* loop that goes through every letter of the alphabet, and prints each one out to the screen?

Think about it:

- What would be inside the parentheses *while (...)* to determine how long we loop?
- How do we know what letter to print to the screen?

Can you use the index variable we just made to address these 2 points?

Feel free to take some time (if you have any initiative at all), to see if you can get this on your own.

If not, here's the answer. Yes, I had to write out all the letters this time.

```
char [] alphabet = { 'A' , 'B' , 'C' , 'D' , 'E' , 'F' , 'G' , 'H' , 'I' , 'J' , 'K' , 'L' , 'M' , 'N' , 'O' , 'P' ,
'Q' , 'R' , 'S' , 'T' , 'U' , 'V' , 'W' , 'X' , 'Y' , 'Z' };

int index = 0 ;
while (index < alphabet. length ) {
   System. out .println(alphabet[index]);
   index = index + 1 ;
}
```

What comes out? This:

```
A
```

```
B
C
D
E
F
…
```

Whoa, would you look at that!? Here it is in plain English:

- Set the index to 0.
- Run some code while the index is less than the length (26).
    - Print the letter in the alphabet positioned at the current index.
    - Increase the index by 1.

This approach of doing something in a loop, then increasing the index by 1 (***incrementing***) is very common. You're basically moving one by one through the array.

Note: If you wanted to you could increase the index by 2 and print out *every other* letter! Wouldn't that be *exciting* ?!?! Wow!!

BREAKING AND ENTERING

Picture your Netflix library (let's see how quickly we can age the book!)

You look at the first entry in Netflix' list of recommendations. You think about it, but don't feel like watching it. You move onto the next one.

You repeat this until you see a TV show or movie where you *know* you'll enjoy it, because you hate trying new things. Once you see an entry that you decide to watch, you stop (and ***break*** the loop).

```
String[] recommendations = { "ShowA" , "ShowB" , "The Office" , "ShowC" };
int index = 0 ;
while (index < recommendations. length ) {
  String recommendation = recommendations[index];
   if (recommendation == "The Office" ) {
    System. out .println( "I'll watch this one:" );
    System. out .println(recommendation);
     break ;
  }
  System. out .println( "Not sure, incrementing..." );
```

```
    index++;
}
```

What do you think happens here?

Do we loop through every single recommendation? Do we stop at a certain point? Why? Here’s the output of the program:

Not sure, incrementing...

Not sure, incrementing...

I'll watch this one:

The Office

See how we didn’t even get to *ShowC* ? That’s because of the magical ***break;*** statement. It says that the loop is done *right now* , regardless of anything else!

So basically, the ***break*** statement lets us kill our loop at any time for any reason. No need to wait until we get back to the top and re-evaluate the condition. This iteration is done, *and* we won’t go onto the next one. Any code we wrote after the while loop can run now.

Continuing on

We’ve seen a while loop that goes forever. Then one that goes through every letter of the alphabet. Then one that stops when it wants to watch “The Office” (unsurprisingly).

There’s another Java keyword I feel I must mention. It’s called ***continue*** .

You know how ***break*** basically kills the loop? Dead, over?

continue says “don’t kill the loop, but skip this iteration.”

Move on to the next iteration. Nothing to see here.

Let’s say you want to print every letter of the alphabet, but not ‘F’. You just hate that letter.

```
while (index < alphabet. length ) {
    char letter = alphabet[index];
    if (letter == 'F' ) {
        continue ;
    }
    System. out .println(letter);
    index = index + 1 ;
}
```

We print out everything except ‘F’. The code after a continue statement is never reached, instead we return to the top of the loop.

It’s like a ***break*** statement’s little brother. It “breaks” this iteration and goes to the next, but it doesn’t kill the entire loop. Lil boi.

If you hated ‘F’ *so much* that you wanted to immediately stop printing out letters, you’d go back to using ***continue*** .

A BUG? IN *MY* CODE?

There’s a problem with my last example. A “bug” as a programmer might say, where something happens that you didn’t expect.

Did you notice it? Probably not. But can you see it now? Why might this beautifully written code not do exactly what I described?

Hint: look at where the *continue;* statement is. What *doesn’t* happen when the letter is F?

Did you get it? Take a second.

Answer: the loop runs forever! Look what happens when the letter is an F; we immediately skip the iteration using a *continue;* and go to the next iteration of the loop.

But what letter are we gonna look at next iteration? F. And the next one? F. We never increased the value of the index!!! We’ll keep pulling F out of the array since the index doesn’t change. Don’t make this mistake.

Here’s the fixed example. We increase the index immediately after pulling the letter from the array:

```
while (index < alphabet. length ) {
     char letter = alphabet[index];
     index = index + 1 ;
     if (letter == 'F' ) {
         continue ;
    }
    System. out .println(letter);
}
```

It might be a contrived example, but it’s a reminder that we need to be careful inside loops that we don’t infinitely spin our wheels.

The doo doo while loop

Note that a loop doesn’t *have* to run at all. What if before we did the loop on our alphabet, we set the index to a value other than 0? A value outside the size of the alphabet?

```
int index = 87 ;
while (index < alphabet.length )
…
```

Well that’s simply too high! The while loop sees the index isn’t less than 26, so it doesn’t run even once! Not a single letter is printed out.

That’s good though, right? No problem with that. The *while* statement is simple and does what it’s supposed to: runs code over and over again while a certain condition is true.

Here’s comes the fun, doo doo doo doo

However, at some point the Java developers thought: why don’t we fix this? Oh, it makes sense as-is, but why don’t we add another kind of loop for no reason?

(Note: this actual conversation didn’t happen)

That’s when we get the “do-while” loop, or “doo doo” while loop, as I call it. Because it’s stupid and no one uses it, except the one guy reading this that is infuriated that I’m insulting his favorite language construct.

Here’s how it works:

I write some code. It’s run once. Then, we start a while loop. It might be run again.

```
do {
   System. out .println( "I hate that I exist." );
} while ( false );
```

What do you think would happen if we run this code? Take a guess inside your little noggin.

"I hate that I exist" is print out to the screen once. Then, the condition is evaluated at the *end* .

Since I literally wrote ***false*** (the opposite of ***true***), the loop doesn't keep going. One could argue the loop never started. This is the opposite of the while loop, which looks at the condition at the *beginning* .

I'll give you some more:

```
boolean havingFun = false ;
do {
  //A game of League of Legends sets havingFun to true or false.
} while (havingFun);
```

In place of actual code, I left a comment there that describes how such code would work.

In this code, we always play one game of League of Legends, even if we aren't yet having fun today. Then, if we are having fun, we keep playing until it's no longer fun.

But really. I have never once encountered a scenario where I found I wanted to use a do while loop. Maybe I'm just an idiot.

Like with anything there will be someone who argues otherwise; maybe using a do-while loop 0.00001% of the time it makes the code easier to read. But I can't be bothered to care.

You can always write it to fit into the other 2 primary kinds of loops, ***while*** loops and ***for*** loops.

Did someone say ***FOR*** loop??

First of all, why another kind of loop? We already have a way to do things repeatedly with the ***while*** loop. And we have a stupid way to do it with the ***do while*** loop.

What else could we possibly need to loop through data, do things over and over until something changes, etc.?

Let's summarize how we wrote a *while* loop for looping through an array.

1. Declare an “index” variable that is used while looping.
2. Write the condition that determines when the loop should keep going or stop.
3. Increase the index at the end of the loop.

The **for** loop (promo code steve)

Well, scientists have invented a type of loop that does all of that much easier! Check out how you’d write a loop the old way, versus with this new type of loop!

```
//While loop
char [] myArray = {'A' , 'B' , 'C' , 'D' };
int i = 0 ;
while (i < myArray.length ) {
   System.out .println(myArray[i]);
   i++;
}
```

```
//For loop
char [] myArray = { 'A' , 'B' , 'C' , 'D' };
for ( int i = 0 ; i < myArray. length ; i++) {
   System. out .println(myArray[i]);
}
```

“Holy shit!” – you, the reader

I mean look at it! It’s majestic! In that one line of code (the one that starts with ***for***) we did the exact same 3 things that while loop did in 3 different locations.

1. Declare an “index” variable that is used while looping.
2. Write the condition that determines when the loop should keep going or stop.
3. Increase the index at the end of the loop.

I’ll admit it. A ***for*** loop is more confusing for beginners, especially so for idiots, buffoons, and debutants. And no, I don’t know what that last word means.

But you get very concise and easy-to-understand code once you get used to it. And guess what? This is the type of loop you see 95% of the time in Java.

That’s because the most common reason to loop is to go through every element in a data set, like an array, and do something for each one. This is simply the most concise way to do that.

Why and when to for loop?

99% of the time, you use a *for* loop when you have an array (or some list) of data. You need to do something *to* each of the items in the array, *or* you need to do something *using* each item in the array.

For each product in the cart, apply the coupon code to it.

For each horse in the race, calculate the odds of it winning.

For each depressing memory, cry for 1 hour.

Reverse, reverse!

It's also easy to do different kinds of loops by adjusting the three sections of the for loop. If you notice, those sections are indeed separated by semicolons.

For example, maybe you want to go through the array in *reverse* . Here's what those 3 sections would specify:

1. Set the index's initial value to the *end* of the array
   - `int i = myArray. length - 1 ;`
2. Run the loop until we reach the beginning of the array (including position 0)
   - `i >= 0 ;`
3. Decrease the value of the index at the end of each loop iteration
   - `i--`

```
char [] myArray = { 'A' , 'B' , 'C' , 'D' };
for ( int i = myArray. length - 1 ; i >= 0 ; i--) {
   System. out .println(myArray[i]);
}
```

Try to read through that yourself a few times to understand what each part is doing. Then I'll spill the beans. I hate beans.

Here's how what we're asking Java to do in English…

1. Create an array with 4 elements, A B C D.
2. Create a variable ***i*** (the index) and set it equal to 3 to start.
3. Specify that the loop should keep going as long as ***i*** is greater or equal to 0.

4. Specify that at the end of each loop, we should decrease the index by 1.

And here's how it actually runs, step by step. If you're confused right now, this part should help.

```
char [] myArray = { 'A' , 'B' , 'C' , 'D' };
for ( int i = myArray. length - 1 ; i >= 0 ; i--) {
   System. out .println(myArray[i]);
}
```

1. Set ***i*** to 3. This is greater than or equal to 0. Run code inside loop (Print 'D'). Decrease ***i*** by 1.
2. ***i*** is now 2. This is greater than or equal to 0. Run code inside loop (Print 'C'). Decrease ***i*** by 1.
3. ***i*** is now 1. This is greater than or equal to 0. Run code inside loop (Print 'B').. Decrease ***i*** by 1.
4. ***i*** is now 0. This is greater than or equal to 0. Run code inside loop (Print 'a').. Decrease ***i*** by 1.
5. ***i*** is now -1. **This is not greater than or equal to 0. Loop is donezo.**

Enhanced for loop

Java makes ***for*** loops even easier. In the simple case where you're simply going through some sort of data collection, you can do this:

```
Meme[] memes;

for (Meme currentMeme : memes) {
   //do shit with the meme
}
```

You don't even need to create and initialize an index variable.

No need to get the item at that position in the array.

No need to increase the index at the end of the loop.

None of that shit.

It's the perfect way to loop through some items in a list. **Always use one of these unless you have a reason not to.**

As another example, you could replace our earlier example like so:

```
~~char [] myArray = { 'A' , 'B' , 'C' , 'D' };~~
~~for ( int i = 0 ; i < myArray. length ; i++) {~~
~~    System. out .println(myArray[i]);~~
~~}~~
```

```
char [] myArray = { 'A' , 'B' , 'C' , 'D' };
for ( char character : myArray) {
    System. out .println(character);
}
```

CHAPTER 9: INSTANTIATION

“Ow! What the heck was that!?

[Instantiation]! That was part one of your ceremony.

Ceremony for what?

We’re going to put you to rest!” – Spongebob Squarepants

WE’RE BACK BABY! OBJECTS ARE BACK ON THE MENU!!!

IT’S TIME TO MAKE AN OBJECT OUT OF A CLASS. READY??

Creating an object

Book myBook = new Book();

“I DID IT! I DID IT, DAD! I MADE AN OBJECT!” - you

Alright, you made an object. Yes, you did. But slow down my child. There’s a ton of complexity in that simple line of code. Let’s do it bit by it.

I’ll label each part of that line.

Book^{1} *myBook*^{2} *= new*^{3} *Book()*^{4} *;*

Feel free to flex your brain and guess what each word is doing before reading on. Now…

1. <u>Book</u> ^{1}

This first part says our variable is going to be a Book. Just like we’d make an int, or a double, or a char, we’re saying that we’re making a variable of a specific type. In this case, it’s a Book.

2. myBook

This bit is saying the name of our variable is myBook. We can name it whatever we want. We could have called it "foo," or "bar," but I absolutely loathe variables with those stupid fucking names.

"So the name of the object we just made is myBook?"

Ehhhhhhhhh…. for now, yes, you can say this is the name of our object. Some sticklers would say "*ackshually* that's an object reference variable, not the object itself, let me explain why you're wrong-"

But for now, you're fine to say the object is called *myBook* .

3. new

This is the absolute magical keyword in that line of code. You are saying **I am making a *new* object out of this class. Bitch.**

You didn't need to say ***new*** when you were using the primitive variable types like ***int*** and ***double*** . This is for the object-oriented shit, baby! Wooh! New Book!

4. Book() [4]

Huh? Didn't we already cover Book? It's the type of variable we're making right?

Why is it at the beginning and then here, at the end? What's the point?

Because you can do fancy shit with those parentheses right there.

When you make a new Book you could say "hey Java, when you make the new Book object, make its *title* this, its *author* this, etc.

And that's the wayyyyyy the news goes…

So yeah.

Book myBook = new Book();

More than meets the eye. Just like you, beautiful reader. There's a big, full, beautiful brain behind those dead-set and lifeless eyes. We'll make it through this together, I promise.

Let's get back to the two parentheses () there. That's a super special part that I'll

[illegible]
cover with this next section. Coming up…now!

Constructors

“Bob, the Builder!

Can we build it?

Bob, the builder!

Yes we can!”

You know, when I went to write down this quote I had to look it up. It turns out they changed “Can we fix it?” to “Can we build it?” And you just ***know*** I’ve gotta be accurate with my quotations, so that’s what I used.

Anyways, what’s a constructor?

Bob

A constructor is like Bob, the Builder. You know, the carpenter with a yellow hardhat. Cool guy.

Bob has a hammer, saw, screwdriver, and all kind of tools to build houses.

In Java, Bob builds the object (the actual house) from your blueprint (the class). Here’s a sample class:

```
class House {
   String color = "Gray" ;
}
```

The blueprint has a default value here.

When the House is *created* , it will have that value. Yes, the line of code that makes the color property “Gray” will not run *until* you create a House object.

Speaking of which, I can ask Bob to actually build the House in a line of code like this:

```
House stevesHouse = new House();
```

The right side is the important bit; it says make a ***new*** House!...and assign it to the variable on the left.

See those two parentheses ()? With nothing inside of ‘em? That’s us telling Bob to build a default, normal, vanilla, bland, unimportant house. The house will be Gray.

```
System. out .println(stevesHouse. color );
```

Yep, that will say “Gray”! Also, don’t get me wrong; gray houses are totally fine. Very adequate.

Default Constructors

Every class has a hidden Bob the Builder by default that looks like this:

```
class House {
   String color = "Gray" ;
   House() {}
}
```

A constructor is always the name of the class followed by parentheses . Then some curly braces {}.

See the constructor in that example?

When a House object is created, the *color* property is given the value “Gray”. Then, the constructor gets to work, and …. well, this one doesn’t do shit. No need to even write it out. But it’s always there!!

You might sometimes see people declare the House like this instead:

```
class House {
   String color ;
   House() {
       color = "Gray" ;
   }
}
```

Switch between this example and the previous one. Note the difference. Well, there’s *effectively* no difference, but you can choose which way you prefer. I like setting the property values inside the constructor (this one).

Getting it right from the start

Anyway, sure; Bob could build the house, paint it gray, and *then* you could come and repaint the place yourself.

```
House stevesHouse = new House();
System. out .println(stevesHouse. color );
stevesHouse. color = "Blue" ;
System. out .println(stevesHouse. color );
```

```
> Gray
> Blue
```

That's fine…

Or, you could get it the color you want right from the start.

Check it out!

```
class House {
   String color ;
   House(String customColor) {
       color = customColor;
   }
}
```

The parentheses are no longer empty! Can you guess what's happening here?

Even if you figured it out, the vocabulary is always just memorization. My least favorite part of any test. Or life.

customColor is called a ***parameter*** .

By putting this parameter in our constructor, if someone wants to build a house, they must now provide a color!

~~House stevesHouse = new House();~~

That doesn't work anymore, dumbass!!! Sorry that was a bit harsh. Especially since you didn't write it, I did.

But since you made a constructor that requires the *customColor* parameter, Java will murder your firstborn child if you try to build a House without specifying a color.

How 'bout this?

```
House stevesHouse = new House( "Blue" );
System. out .println(stevesHouse. color );
```

Yep, that works! We build the house, and it’s blue. “Blue” is printed to the console.

Brain teaser!

What do you think happens here?

```
class House {
   String color = "Gray" ;
   House() {
       color = "White" ;
   }
}
```

What color is printed if I do

```
House coloredHouse = new House();
System. out .println(coloredHouse. color );
```

The answer? It’s…. the … color…. White! Yep, the constructor’s code runs *after* the properties’ values are assigned.

You can verify this by adding some Sysouts.

```
class House {
   String color = "Gray" ;
   House() {
      System. out .println( color );
       color = "White" ;
      System. out .println( color );
   }
}
```

The result?

```
> Gray
> White
```

Bob had Gray on his blueprint, but when he went to start building, he decided on White instead.

Brain teaser over. Your brain is no longer being teased.

Brain teaser #2!

Haha, gotcha! We’re back with another one.

What if we create a constructor that looks like this?

```
class House {
  String color ;

    House(String color) {
     color = color;
  }
}
```

What the hell even happens on a line of code where we say that “color” should be assigned the value of “color”??? color = color???

The answer is nothing happens. The house’s color property is left alone. The custom color value that we’re providing Bob is re-assigned the same value. That’s fucking stupid.

We can fix this by specifying that we want to assign the custom *color* to our property ***color*** .

```
class House {
  String color ;

    House(String color) {
      this . color = color;
  }
}
```

Just like you can access an object’s properties and methods with “dot notation”, a.k.a. a period, an object can refer to its own properties and methods using ***this*** followed by the dot.

Which is great in this case where we need to differentiate between our property and the parameter, both with the same name.

¿Porque no los dos?

¿Porque no los dos?

It’s time to introduce you to a concept called engineer overkill.

Wait, that’s not it….

Constructor strain?

No, no…. constructor…overload? Yep, that’s it! ***Constructor overloading*** !

Fortunately for me, the Java developer, I don’t have to have just one constructor. I can have as many, or as few, as I want!

Unfortunately for you, it’s another thing you should learn.

Treat your eyes to this: *two* constructors!!

```
class House {
   String color ;
   House() {
       color = "White" ;
   }
   House(String customColor) {
       color = customColor;
   }
}
```

A class with two different ways to create an object from it!

Specify the color if you want. If you don’t, it will be White! Java is so smart (even smarter than me) that it will use the constructor that matches whatever you throw into the parentheses ().

Here’s how we’d provide Bob with the specifications for two different houses. The first one we don’t give a shit what color he paints it. The second one needs to be a beautiful magenta.

```
//Use the empty constructor
House houseOne = new House();

//Use the String customColor constructor
House houseTwo = new House( "Magenta" );

System. out .println(houseOne. color );
System. out .println(houseTwo. color );
```

Here’s the output.

```
> White
> Magenta
```

The parentheses can be empty *or* have a value for the customColor parameter!

This is a little bit more valuable when you have a real class more than one property. But this book is designed for morons so what the hell do you want from me? You can have more than one constructor. Lesson over.

null

Ever heard of a NullPointerException? No? Well if you take up Java, you’re going to be seeing them for the rest of your life.

But before we get there, let’s talk about ***null*** . You can nullify an object like you’d nullify your third marriage.

```
Marriage marriage = null ;
```

That variable is set to null. What is null?

It’s a special, secret sauce. There’s nothing else like it in Java; it basically means “the absence of anything.”

And here’s the unfortunate thing; with all these cool object-oriented concepts we’ve learned: properties, methods, and the like, ***null*** was never invited to the party. And it’s got a vengeance.

Null means your object has absolutely *no* value and if you try use its properties or methods, there will be hell to pay. Meaning an application crash if you’re not careful.

Let’s revive the House example for a second. We know it has a property called *color* . Why don’t’ we try to give that variable a value when the House hasn’t even been built yet?

```
House stevesHouse = null ;
stevesHouse. color = "Blue" ;
```

You broke it. NullPointerException baby! Whoo!! Application crash ahoy!

```
java.lang.NullPointerException
```

There’s a practice in Java called null-checking to avoid this sort of thing.

There's a practice in Java called null-checking to avoid this sort of thing.

Basically, **unless you are *absolutely positive* that your variable is not null, you'd better check yourself before you Shrek yourself** .

```
if (stevesHouse != null ) {
   stevesHouse. color = "Blue" ;
}
```

Does it seem contrived? Silly? Unnecessary?

It's obvious that stevesHouse is null; I just set it to null! Why do a "null check"?

Programs can become big and scary. Lots of classes, with lots of properties and methods. Your program might talk to another program. Or communicate over the internet. Or expect the user to type in some values.

Maybe at some point I'd want to do something like this:

```
stevesHouse.upstairsFloor.bathroom.sink.faucet = new Faucet();
```

You'd better pray to god that the faucet you want to replace is on the sink, in the bathroom upstairs, in Steve's house, and that everything is perfectly in place, and no one has fucked with the House while you weren't looking.

Since even one *NullPointerException* can kill your program, it's a good idea to practice defensive coding. That means ensuring your variables have reasonable values before trying to use them.

CHAPTER 10: METHODS

“Do it.” – Senator Palpatine

Methods do things.

Airplanes fly(). CoffeeMakers brew(). People fly like paper, get high like planes. Cats meow() for no fucking reason when I’m trying to work god DAMMIT JASPER I JUST FED YOU SHU-.

```
class Cat {
   String name ;
   Cat(String name) {
       this . name = name;
   }
    void meow() {
      System. out .println( "Meow!" );
   }
}
```

name is a property.

meow() is a method.

Why do Cats meow? In real life, who fucking knows. In Java, it’s when we call the appropriate method on the Cat object.

```
Cat jasper = new Cat( "Jasper" );
jasper.meow();
```

What did we just do? Well first we instantiated a new Cat, which I covered in the last chapter, you plumbus.

But after that we used a method called *meow* . It printed out “Meow!” to the screen.

We could call what we just did a a whole variety of things. ***Invoking*** the method.

Calling the method. ***Running*** the method. ***Executing*** the method. “Doing the needful.”

By and large, I see and use the word ***calling*** . We called the meow() method. When the meow() method was called, the cat meowed.

Lost in Translation

```
class Translator {
  String languageFrom ;
  String languageTo ;

    Translator(String languageFrom, String languageTo) {
     this . languageFrom = languageFrom;
     this . languageTo = languageTo;
  }

    void translate(String textToTranslate) {}
}
```

Wew lad, a new class! This one is responsible for translating text between two different languages. I’ll hide the code that would actually do the translating. But it’s great, I assure you.

Let’s create a new translator. I’ll call him Eugene.

```
Translator eugene = new Translator( "English" , "Spanish" );
```

Right now we have him wired up to go from English to Spanish.

We can call the translate method to learn how we find a bathroom in Spanish.

```
eugene.translate( "Where is the bathroom?" );
```

We did it. Right? I think we did. But where’s the Spanish translation? Did Eugene translate it in his head but not even tell us the translation? EUGENE? WHAT’S THE TRANSLATION? HELLO? USE YOUR WORDS.

The problem is our translate method is ***void*** .

void methods do not give us any data when they’re done executing. They do something, and that’s it. It’s like sending a text message and hearing nothing back, no matter how hard you wish they’d just fucking reply.

Eugene translated it in his head but didn’t share with us. We can fix this by making his method ***return*** a String.

```
String translate(String textToTranslate) {
   textToTranslate = textToTranslate.trim();
   String translation = //write this code later
    return translation;
}
```

This special keyword *return* ends the method right there; no code can exist after a return; statement. Method’s over, result is returned.

Methods can *return* any of the data types we’ve seen, whether primitive variables like int, char, boolean, or objects like String, Cat, or Meme.

In this case, we replaced *void* with *String* . Now we can learn the translation from Eugene without having to fire him.

```
Translator eugene = new Translator( "English" , "Spanish" );
String translatedString = eugene.translate( "Where is the bathroom?" );
System. out .println(translatedString);
```

> “¿Dónde está el baño?”

Thanks, Eugene!!

Conditional returns!

“What is this, a crossover episode?” – Mr. Peanutbutter

In this little section, conditionals return. Hah. They return. Literally. Cause they’re back and….alright.

Anywho, you can actually have *multiple* return statements in your methods. The rule is that nothing can exist in a method after a return statement; but what if your return statement is inside a conditional? You’ll need multiple return statements then!

For non-void methods, Java needs an absolute 100% guarantee that the method will return a value. This would be unacceptable:

```
boolean isPositive( int number) {
    if (number > 0 ) {
        return true ;
    }
}
```

What if the number is less than or equal to 0? You need to explicitly give your method something to return in *all* scenarios! Java isn't smart enough to infer that anything else would be *false* , for example.

You'd have to do something like this:

```
boolean isPositive( int number) {
    if (number > 0 ) {
        return true ;
    } else {
        return false ;
    }
}
```

Or, you could provide a "catch-all" return value at the bottom of your method, which in this example acts the same way:

```
boolean isPositive( int number) {
    if (number > 0 ) {
        return true ;
    }
    return false ;
}
```

Static methods

Typically, you call methods on *objects* . The idea is that the methods will interact with the properties on that object; change the value, retrieve the value, initialize a value, all that shit.

But you can create methods that don't need to be called on an object. Just the class. Those methods are ***static*** .

Here's a few examples

```
class StringHelpers {
    public static boolean startsWithA(String str) {
        return str.charAt( 0 ) == 'A' ;
    }
    public static boolean isNullOrEmpty(String str) {
        return str == null || str.isEmpty();
    }
}
```

We don’t need to create a *new StringHelpers()* object, then use a dot “.” to call the method on that object. We can straight up just use the StringHelpers class itself.

```
String myString = "Alpha" ;
if (StringHelpers. startsWithA (myString)) {
    System. out .println( "It starts with A." );
}
```

In general, you don’t want to make a method static unless it’s some simple helper methods like these. You’re in an object-oriented language, it’s a good idea to use objects and their methods, not static methods on classes.

Note: some people would call these static methods “***functions*** .” The difference between methods and functions is mostly a pissing match between programmers. Typically, I’m pretty sure methods are just functions that are attached to objects. And since Java is highly object-oriented, we typically call them methods.

Parameters and Arguments

Methods can have data passed to them when they’re called. We’ve already seen this a few times, but I should probably cover this explicitly, too. Call it a moral obligation…

```
class Airplane {
    void fly() {

      }
}
```

That method *fly* has 0 ***parameters*** . There’s nothing inside the parentheses () of this method definition. You could simply call it like this:

```
Airplane airplane = new Airplane();
airplane.fly();
```

Let’s add a parameter so we can specify what altitude we fly at.

```
void fly( int altitude) {

}
```

Now the method has 1 parameter: *altitude* . When you call this method, you can “pass” the altitude to the method, and the method can use it to fly to the desired altitude.

```
airplane.fly( 15000 );
```

Make sense?? Remember when we could pass things to a constructor? Same thing with methods!

Okay…so that’s a parameter. Something we can put in our method definition so that we get data passed to it when it’s called.

But what’s an ***argument*** ?

Well, it’s the other side of the coin.

When you’re typing out your method and specify what can be passed to it, you are specifying a method’s ***parameters*** .

When you actually call the method and pass something to it, you pass an ***argument*** .

`int altitude` is a parameter
`15000` is an argument.

Parameters are fulfilled by ***arguments*** . Like your gaping maw (the parameter) is fulfilled by 100 McNuggets from McDonalds drenched in Big Mac sauce (the argument).

I want more….more parameters

You can have as many parameters as you want. Maybe your Airplane needs to know the desired altitude, but also the desired speed.

```
void fly( int altitude, double speed) {

}
```

Yup that’s fine. when you call the method, pass the two arguments like so:

```
airplane.fly( 15000, 700 );
```

Java is fucking EASY.

Overloading

Remember constructor overloading? No? Well we can do something similar with methods that makes our objects easier to interact with! It’s called ***method overloading*** and lets us declare multiple versions of that method, just like an object can have different constructors.

With this technique (TECHNIQUE, TECHNIQUE!), whoever is interacting with your object can choose the version they like best.

```
class Airplane {
    void fly() {

      }
    void fly( int altitude) {

      }
    void fly( int altitude, double speed) {

      }
}
```

Now anyone using an Airplane object can change the altitude without specifying the speed, or begin flying without specifying *anything* .

```
airplane.fly();

airplane.fly( 15000 );

airplane.fly( 0 , 12552.854 );
```

Yes, that last line will crash the airplane at an astonishing rate. Don't worry, the fly method *should* make sure the argument passed is reasonable.

It's typically a good idea to overload your methods if you want to *allow* more customized behavior when the method is called, but also want to provide an easy way to call the method without passing so many arguments.

Under the hood, you might even do something like this to make your code very reusable:

```
class Airplane {
    void fly() {
      fly(currentAltitude);
   }
    void fly( int altitude) {
      fly(altitude, currentSpeed);
   }
    void fly( int altitude, double speed) {
       //The actual code to fly
    }
}
```

See how they're calling each other? This is an example of how you can reuse your code without copy pasting!!!! It's like a chain or something.

Just want to start flying? Call *fly* and it will use the current altitude. Want to change the altitude? It will *fly* to that altitude with the current speed. Want to change both? Call the full fly method.

Method to the Madness

Objects in Java do shit all the time. That's why we write code after all; the program needs to *do* something valuable for us humans.

Methods are helpful for more than *just* doing things on behalf of the object; they also let us programmers turn a confusing mess of spaghetti into a reasonable chunk of code.

For example, remember Eugene? Do you think Eugene has just one method to perform his translation?

Under the hood, that *translate* method could have been doing a whole litany of things:

```
String translate(String textToTranslate) {
   cleanseInput(textToTranslate);
   parseWords();
   translateWords();
   adjustPunctuation();
   …
    return translation ;
}
```

Much better than a 1000-line method of the noodle variety.

Divide your methods up to prevent method madness.

CHAPTER 11: STRINGS

String theory.

A theoretical framework in which the point-like particles of particle physics are replaced by one-dimensional objects called strings.

My string theory is that Strings in Java are the best variable type of all time. ALL TIME!

What’s a string?

You’ve seen ‘em. I’ve talked about ‘em. You probably have a general idea of what they are by now.

It’s just some text, right? “abc” or “23890chusoao”? Aren’t those Strings?

Yep. The double quotes “ “ are what identify Strings. That or, you know…the word String. Duh.

But Strings are super special in Java; not only are they extremely common, they have a ton of built-in methods that make them super powerful. And you oughtta know why Strings are the best. The kind of “oughta know” that Alanis Morisette would advocate.

Anything that does not require math (addition, multiplication, that stuff) and is not a boolean (true or false) value should be a string 99.9% of the time, ISO (In Steve’s opinion)

Text

Want to represent a single character? You could use a ***char*** , right?

```
char firstLetter = 'A' ;
```

Just use a String.

```
String firstLetter = "A" ;
```

Why bother using a primitive when you can get the real deal? Needless to say, if you need a string of characters, you'd use a String, too.

```
String myName = "Steve" ;
```

Did…did you just say a ***String*** of characters?

Yep, that's what a String is, at least under the hood. Somewhere inside how a String is made in Java there exists a char[]. In fact I just looked up the source code for String. Lookie!

```
public final class String … {
    //The value is used for character storage.
    private final char value [];
    …
}
```

Picture a bunch of characters lined up, with a piece of string tying them together. Not all that firmly; you could snip the string if you wanted to. But they're a unit!

Yay, Strings!

Numbers

And numbers? Just because a value looks numerical doesn't mean it should be an integer. Don't get me wrong, lots of times you do need numbers to be *ints* or *doubles* (fuck floats). But not always!

For example, phone numbers…

```
long jennysNumber = 5558675309L ;
```

You ain't gonna be adding phone numbers together! Make it a string!

```
String jennysNumber = "5558675309" ;
```

Making a String

"You just put some stuff inside two double quotes, I get it."

Well…I should tell you that theoretically you *could* do this. Don't.

```
String jennysNumber = new String( "5558675309" );
```

Yep, **Strings are objects.** They have constructors.

This constructor is absolutely useless, though. Seriously, more useless than nipples on a man.

Java lets us use shorthand to simply put the value inside double quotes and lets that be the end of it.

There *are* other ways you might need to create a String, though. For example, maybe you have an *int* but just absolutely need it to be a String.

```
int foundThisInADatabase = 34897320 ;
String strRepresentation = String. valueOf (foundThisInADatabase);
```

You can use the static *String.valueOf* method to convert the *int* to a String. Same goes for pretty much any other type of variable. Neato!!

String Comparisons

Remember the comparison operator? It's two equals signs. Here, I'll remind you.

```
boolean isSeven( int number) {
   return number == 7 ;
}
System. out .println(isSeven( 5 ));
```

We print out ***false*** because 5 ==7 is simply not true.

Strings, though, should *not* be compared like that. Bad! Bad comparison operator!

Let me show you why. For the purposes of this example, I will be using the String constructor instead of just some double quotes. Please, forgive me.

```
String strOne = new String( "Ayy lmao." );
String strTwo = new String( "Ayy lmao." );
boolean sameString = strOne == strTwo;
```

You're on Who Wants to be a Millionaire? What is the value of *sameString* ? Just like the show, you can phone a friend if you'd like.

A. true
B. false
C. null
D. The program crashes

First of all, a primitive cannot be *null* , so jot that down.

But the answer is B: *false* ! Yep, those strings are not the same *according to the comparison operator* (double equals sign).

We know they're the same, however. So what's the deal, why is it saying they're not the same?

When it comes to objects, the comparison operator == is literally checking "are these the same object?"

Well, these aren't. You created two different objects, right? *strOne* and *strTwo* . You failed.

The lesson? Don't use the comparison operator for Strings, or any object for that matter, if you're trying to see if they're the same value.

How do we do it, then? Like this!

```
boolean sameString = strOne .equals (strTwo);
```

Bam! *sameString* is now true.

Superpowers

Alright, maybe not superpowers, per se. But Strings are objects, as I said. You just saw the *equals* method that checks if two Strings have the same value.

Unlike the trite and overly simplified examples we've used so far, the String's superpowers are actually useful methods that accomplish actual things! Let's check out some of them!!

equals

We just covered it, but just to reiterate: when you want to compare two strings, never use ==. Always use the *equals* method that all Strings have.

```
boolean sameString = strOne.equals(strTwo);
```

This is something you actually do a lot in Java. Did the user type in a certain word or sentence? Does it match a value we expected? Now we can do that with Strings!

Thanks Java!@! Programming is fun.

equalsignorecase

For when you want to use *equals* but don't care if the characters in the string are uppercase (A) or lowercase (a).

```
//true
"MEMES" .equalsIgnoreCase( "memes" );
```

contains

For when you want to see if there's a particular String inside your String. Maybe you're Mark Zuckerberg and want to see if a user's Facebook post contains "vaccines".

```
//true
"Boy am I glad that vaccines were invented" .contains( "vaccines" );
```

Note that *contains* is case sensitive. That means it cares about if the characters are uppercase or lowercase. i.e.

```
//false
"BOy I HatE HoW VACccINeS CauSe AUTisM!" .contains( "vaccines" );
```

Hmm. How can we see if "vaccines" is inside there, no matter what case the letters are in? I know! We'll use *containsIgnoreCase* !

Just kidding – doesn't exist! Dammit Java.

There's one easy way to do to this for 99.9% of scenarios. It involves our next superpower, so we'll go over it first.

toLowerCase and toUpperCase

```
String bestGame = "Old School Runescape" ;

//"OLD SCHOOL RUNESCAPE"
String bigBoy = bestGame.toUpperCase();
```

```
//"old school runescape"
String lilBoi = bestGame.toLowerCase();
```

Extremely straightforward. Just changes letters to either the upper- or lower-case versions.

The people who wrote those methods? Not as straightforward for them .

containsIgnoreCase? Remember? Hello?

Ah, right! Okay, first, you just standardize what case the letters are in. Then you can do *contains* .

```
String testString = "BOy I HatE HoW VACccINeS CauSe AUTisM" ;

//"boy i hate how vaccines cause autism"
testString = testString.toLowerCase();

//true
testString.contains( "vaccines" );
```

If you really wanted to, you could instead change it toUpperCase and look for “VACCINES”.

charAt

Remember “indexes” from arrays? You know, [0] gets the first element, [1] gets the second element, and so on? Well that knowledge, unfortunately for you, is coming back from the dead for this exercise.

```
"we live in a society" .charAt( 1 );
```

This returns the *‘e’* character.

```
"I hate sand" .charAt( 7 );
```

This returns the *‘s’* character.

Yes, spaces from your spacebar “ “ are characters, so you need to include them when you’re dealing with the indexes of characters in a String.

Not too bad. Now for the next method. :)

substring

Let’s say we want to get the first 5 letters of a String.

```
String helloWorld = "Hello, world!" ;
```

Oh god, it’s back.

Anyway, we can pull out a “substring” inside this String by giving Java the starting index and ending index of the characters we want.

```
String firstFive = helloWorld.substring( 0 , 5 );
```

This is saying “get me the 0th character up to the 5th character”. The result of which is “Hello”.

Wait, 0 to 5? Isn’t that 6 characters? 0, 1, 2, 3, 4, 5….that *is* 6! What the hell is happening here?! Why did we only get “Hello” and not the comma too?

Here’s the tricky bit (you knew it was coming): Java don’t work like that.

You are asking to get the 0th character ***UP TO, NOT INCLUDING*** the 5th character. This is known as “inclusive” and “exclusive”.

```
String firstFive = helloWorld.substring( 0 , 5 );
```

“Get the substring starting at index 0 INCLUSIVE and ending at index 5 EXCLUSIVE.”

0, 1, 2, 3, 4, 5 (here, 5 is the friend without a group in class and is left out)
H, E, L, L, O

Let’s do another.

Look who’s purging now

```
"Ah geez I don’t know Rick" .substring( 4 , 7 );
```

Let’s say this String is written out on a thin sheet of paper.

Picture taking some scissors and cutting the string at position 4 and 7. Everything we don’t want is thrown into the wastepaper bin.

Cut one

The first cut we make is at the lowercase g in “geez”. It’s 4 over from the left.

Because the rules state that we *include* the first index in our result, we cut to its left so we don’t lose it. “Ah “ is then thrown in the trash.

Cut two

The second cut we make is at the blank space “ “ after “I”. It’s 7 over from the

The second cut we make is at the blank space “ ” after [illegible] over from the left.

Remember; we don’t want the character at the 7th index; it’s ***excluded*** . We also don’t want anything to the right of it.

Because the rules state that we *exclude* the last index in our result, we cut to its left so it’s excluded. Everything to the right of, *and including the blank space “ ”* is then thrown out.

Snipping complete! The paper remaining says **“geez”.**

In the trash, we have *“Ah “* and *“ I don’t know Rick”* . Visualize the String being cut up and follow these instructions once or twice again.

More sugar

I know. I know. You’re a~~n idiot~~ newbie and you’re still confused. But this is one of those things that really just needs drilled into your head.

Here, I’ll give you some examples to help you out, friend. Use your finger to count from left to right and make sense of why the results are what they are:

```
 //"NOP"
"LMNOPQ" .substring( 2 , 5 );

//"B"
"ABCDEFG" .substring( 1 , 2 );

//"WH"
"WHAT" .substring( 0 , 2 );

//"" (empty string)
"GAWRSH" .substring( 1 , 1 );
```

Hopefully you’re beginning to comprehend this.

One more thing to mention here. You know how we give the start index ***and*** end index?

Well, Java lets you only give the start index, if you want. The result? You only make one cut, and Java gives you the entire tail end of the String.

```
//"Cool"
"Lame Cool" .substring( 5 );
```

```
//"BCDEFGHIJKLMNOP"
"ABCDEFGHIJKLMNOP" .substring( 1 );
```

RIP in pieces “A”

split

I like this method. It creates a bunch of smaller Strings out of your String.

You give Java a character to “split” on. The string is karate chopped at each occurrence of that character. You’re left with an array of the results.

```
String sentence = "I just think they’re neat" ;
String[] words = sentence.split( " " );
```

Guess what? We now have an array of the words! That’s 5 items in the array!

```
{ "I" , "just" , "think" , "they're" , "neat" }
```

When Java saw an empty space, it dropped a tactical missile and split the String up. The bits to the left and right ended up in the array.

Freaking nice right there.

trim

Okay, last one. That I’m going to mention…there’s a few more. But this one’s easy.

It removes all the empty whitespace and newlines from beginning and end of your String.

```
"      This book is great.  " .trim();
```

We trim the fat and are left with “This book is great.”

I trim most of the strings I deal with if a human might have provided it. No one wants that shit at the beginning and end of the text.

that’s all, folks

There’s a lesson to be taken from this, in addition to all the String method themselves.

[illegible]

Don’t write your own method for doing something, even if it seems easy, if someone else already devoted time and energy to creating a quality one.

If you think it’s confusing to start using these methods, I can assure you it would be even more confusing to try and write them yourself.

Chances are if you need to manipulate a String, someone else has needed to do the same thing. In these cases, the Java Standard Library itself is giving you the methods you need. If Java doesn’t have it built-in, use the internet to find a good way to do it. You might be able to make your life a lot easier.

There are also ways to download “libraries” full of helpful code for you to use, written by other members of the Java community, but that’s outside the scope of this lil book. For now, stick to Googling and copy-pasting like the rest us.

Concatenation

Still with me? Good.

Here’s another big confusing word. Concatenation.

Remember the plus sign? You can use it to add numbers together.

```
int result = 3 + 2 ;
```

Answer: 5

Java *also* uses the plus sign to let us *“add”, or “concatenate”* (kun-kat-en-ate) things to Strings.

```
String helloWorld = "Hello, " + "world!" ;
```
--> ”Hello, world!”

```
String cashRegisterDisplayStr = "DISCHARGE " + change + " IN CHANGE TO CUSTOMER" ;
```
--> “DISCHARGE 7.09 IN CHANGE TO CUSTOMER”

```
String thisIsNotTwo = "1" + 1
```
--> “11”

Note: with that last example… if you try to add two things together and one of them is a String, they will be concatenated, *not* added. It’s “11”, not 2.

Okay, not too bad! One more example.

Let’s say you’re a shopping website on the interwebs. The user purchases an item, and you want to thank them for purchasing it. Maybe you'd show them a message like this:

```
String itemName = "Everybody Poops" ;
double price = 2.99 ;
String messageToUser =
"Thanks for purchasing the " + itemName + " for $" + price + "." ;
```

You might see “Thanks for purchasing Everybody Poops for $2.99.” when we eventually show that String to the user. No, *System.out.println()* doesn’t cut it for a website…but that’s a different book.

Also, you bet your ass I added an extra String concatenation to put a period at the end of my sentence, even for an example. Shit pisses me off.

Anyway, String concatenation is used eeeeeverywhere. I mean really. Combining Strings with Strings, or with numbers, or objects, or all kinds of shit. **+** is an important little dude.

Formatting

That last example was a lot of plus signs to make a big ol’ String. Those variables really belonged *inside* the String there, didn’t they? We had to manually build the String ourselves; cutting up the string into parts, and inserting the variables where they should fit in.

There’s an easier way. It’s a static method available on the String class called *format* . Yes, it’s static, meaning you can literally call String.*format* (note that capital S) without needing an object to use the method.

Here’s how you might do the exact same message with this approach.

```
String messageToUser =
String. format ( "Thanks for purchasing %s for %f." , itemName, price);
```

See those letters with a percentage sign % before them? No, it’s not a percentage sign like you’re used to. This isn’t a value out of 100. This isn’t the modulo operator either.

They’re like little placeholders. Then, when you use String.format, you give Java the real values! It will insert them where they belong.

There is a method (hehe) to the madness of which letter to use. Basically, each type of variable has its own letter. And you know what?

type of variable has its own letter. And you know what?

I’m not pasting a bit chart here. If you find yourself needing to make a String that contains variables in it, use your Google-fu to find the table online . Or just use %s to treat everything as a String. :)

Strings are Immutable

What does immutable mean? Does it describe your remote when you’re watching TV and need to mute it, but it just won’t work?

No. It means Strings ***cannot be changed.***

Concatenation? That created a new String.

Any time you use one of the String methods we’ve seen? They created new Strings.

You need to make sure you’re not expecting the String to *change* by using one these methods. Instead, you must expect Java to create a new String from the shit you’re asking it do.

```
String initialString = "Ah, hello there." ;
initialString.toUpperCase();
System. out .println(initialString);
```

--> “Ah, hello there.”

See? The string didn’t change to be in all upper-case. Were you expecting Obi-Wan to be screaming “AH, HELLO THERE.”? Too bad.

Sure, we called the *toUpperCase()* method. But that does *not* modify the existing String we made; it makes a *new* one with those modifications.

And what did we do with the String that was created when we called *toUpperCase()* ? Nothing! It vanished into the ether. We didn’t assign it to a variable or use it inside a conditional.

Another failure:

```
String sillyString = "haha dank memerinos!" ;
sillyString.substring( 5 );
System. out .println(sillyString);
```

> “haha dank memorinos!”

```
> haha dank memerinos!
```

Please kill me.

But again, that String method did not *change* the value of our String, it created a new one that we ignored. We did not change sillyString to be “dank memerinos!”

In fact, if you look at the methods available on the String class, none of the ones that “change” or “modify” the String are ***void*** . Instead, Java creates a new String based off the original one and returns that for you to use.

Basically, you don’t need to worry about *why* strings are “immutable”; just ensure you don’t find yourself trying to change a String instead of creating new ones based off the original.

Oh, and here’s how you’d properly write those examples:

```java
String initialString = "Ah, hello there." ;
String initialStringUpper = initialString.toUpperCase();

String sillyString = "haha dank memerinos!" ;
//Reusing the same variable
sillyString = sillyString.substring( 5 );
```

CHAPTER 12: THE JAVA CINEMATIC UNIVERSE

"I know what it's like to lose. To feel so desperately that you're right, yet to fail nonetheless. It's frightening. Turns the legs to jelly. I ask you, to what end? Dread it. Run from it. Destiny arrives all the same. And now, it's here. Or should I say, I am." – Thanos

In Java, the big bad isn't Thanos. It's complexity.

Complex code is the enemy of all programmers. At least the good ones. The more complex your source code, the harder it is to fix issues, to add new functionality to it, to hand it off to someone else to work on, you name it. And it only makes it easier to introduce bugs, confuse yourself, and regret ever starting the project at all.

There are a lot of ways to battle complexity. You can have small, well named methods. Straightforward and meaningful property and variable names. Comments on particularly tricky portions of code.

But I'm moreso talking about complexity at *scale* .

We've really only seen one or two classes at a time so far in my little examples. But other than superman (yawn), no superhero can or should try do *everything* .

That's why it's important for Java superheroes to band together and take on complexity as a team. Many times you can recruit classes from Java to help you out, rather than writing that code in your current classes, or writing new classes on your own.

- Need to store data in a file?
- Need to connect to a database
- Need to generate a random number?
- Need to make a network call?
- Need to display a user interface with buttons, input fields, widgets, and

that kinda jazz?

Java has a lot of premade classes available to us to help with common tasks like these.

These are all part of the “Java standard library” that I mentioned at the beginning of the book. Basically, they’re ready-to-go when you need them, as part of Java itself.

They might not be the sexiest superheroes around, but they are there for you. Unlike your father that left for milk in the middle of the night and never returned. I miss you, dad.

Packages

One way to reduce complexity and keep things tidy is to organize your classes into “packages.”

If you don’t specify a package at the top of your class, you’re considered to be in the “default” package. Never do this unless you’re just messing around. Even then, stop it. Get some help.

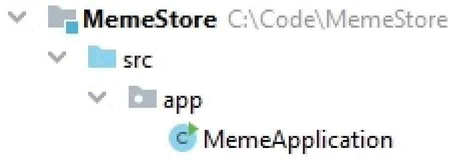
What is a package, though?

You can think of it like a literal brown box package that contains your classes. When boxing things up for storage or for a move in real life, you probably have some rhyme or reason to how they’re grouped together. That’s what a package is for.

Logically, it’s a way to group together related classes; bundling like classes with other like classes.

In practice, it’s the folder that your Java file is in, combined with a package declaration at the top of your Java file.

For example, here’s an example, so you can view an example:



```
MemeStore C:\Code\MemeStore
  src
    app
      MemeApplication
```

Note that I’m using IntelliJ because I’m not a serial killer.

I have a big daddy folder for my project called MemeStore. That’s just what I wanted to call it, the name doesn't matter in the slightest.

I then have a folder called "src" that will contain alllllll my Java stuff. This is just a standard thing for most Java developers to do, but it's not mandatory.

Then my package! I just called it "app" because I'm creating an application. Not very creative, I realize.

Note: A package is just a folder! You could make the directories manually inside Windows Explorer, or from your command prompt. Doesn't matter. I used IntelliJ because I'm not a sociopath.

Then, like I mentioned, the class declares its package at the very top of the file.

MemeApplication.java

```
package app;

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


     }
}
```

It's literally just the word "package" followed by the directory the Java file is inside of. Oh, and a semicolon.

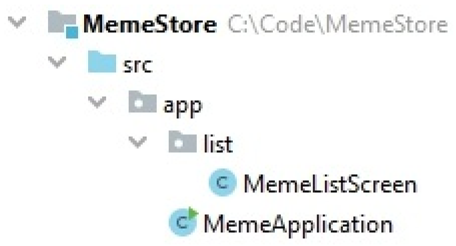
And yeah, this class doesn't do anything right now. It's just a main method.

Maybe if we actually developed this thing, we'd open a new window and show the user a list of memes. I'm really just focusing on the packages for right now, thank you.

Matryoshka dolls

You can keep creating folders inside your folders. And therefore packages inside your packages. Like those Russian dolls that have a Russian doll inside the Russian doll inside the Russian doll.

I did mention that we might show the user a list of memes, since it's a meme store. Let's add a new package called "list". And put a MemeListScreen class in there.



OK, so we created that new package (literally a folder) and created a new java file called MemeListScreen.

MemeListScreen.java

```
package app.list;

public class MemeListScreen {

}
```

And surprise, it literally is nothing. But notice the package line. See the period there?

This is saying that this class belongs to the “app.list” package, which is the folder structure it’s inside of.

For example, if we were to keep doing this, inside that “list” folder we could create an “options” folder. And then inside that “options” folder we could create an “about” folder. We could then have a Class inside the *package* app.list.options.about.

Maybe it’s called AboutMemeStoreScreen or some shit. I don’t know.

That class would put this at the top of the file: `package app.list.options.about;`

Getting confused? The big idea is this: We organize our code to keep things tidy and easy to understand.

Just like you might put your Word documents in one folder, and your PowerPoints in another folder, it’s a good idea to put related Java code in the same folder (package). And you can keep breaking it down further and further.

Naming your packages

Like with many areas of Java, there are conventions (or common ways to do things) when it comes to creating packages. I’ll just show you a few example packages first, as the conventions will probably become obvious:

- com.google.gson
- com.google.dagger
- com.google.copybara.templatetoken
- com.google.common.math
- com.google.common.base.internal

Notice every part of a package is lowercase. And that even if a package is multiple words (see the example “templatetoken”?) we still combine it into one word.

The most confusing part, by far, is the “com.google” part…why is there a website name before all the packages? And why is it backwards?? The fuck is going on?

People, at least in America, have first names, middle names, and last names to more uniquely identify us, yeah? It’s like that, except way more specific. Maybe like a social security number.

It makes it straightforward to share code between developers without package names clashing.

What if Google, Java, Facebook, Amazon, and Steve all had a package called “math”. And we *all* wanted to share it with the world, so that no one has to write code to do math ever gain? We’d have a bad time.

Instead we prefix all of our packages with our website name.

“Okay, but why is it in reverse?”

IDK. People just do that. Also, if you don’t have a domain name, make one up.

com.stevebrown.memestore.app

Seems good!

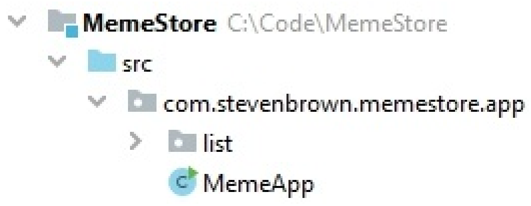
So… that’s kind of confusing

Is it? It probably is. It was much more intuitive (that means easy to understand) when we just had a folder called “app”, with some folders inside there.

Does this mean we now have a “com” folder? Then a “stevebrown” folder? Then inside *that* all my actual packages with my Java code?

Well, yes. That’s correct. But since we’re sane human beings and not psychopaths, we’re using a program like IntelliJ to write code in. And it will handle creating all those folders for us, based off the package name.

For example, I can use IntelliJ to make a package structure like this:



See how "com.stevenbrown.memestore.app" is displayed as *one* big package?

Sure, it's actually a folder within a folder within a folder within a folder under the hood. But IntelliJ (or god forbid, Eclipse) creates the folders on your behalf.

In this instance, I absolutely have that entire folder hierarchy on my computer:

C:\Code\MemeStore\src\com\stevenbrown\memestore\app

Oh, and to be clear, your package declarations also reflect this full "reverse domain name":

```
package com.stevenbrown.memestore.app;

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


     }
}
```

This will come to you naturally in time. I swear. Trust fall?

How should I package my classes?

It's totally up to you to decide the best way to organize all your classes. But here's some popular options people use:

1. Package by feature
2. Package by layer
3. Package by whatever the hell you feel like

Package by feature

If you like option 3 then you can skip the rest of this section for now. Wimp.

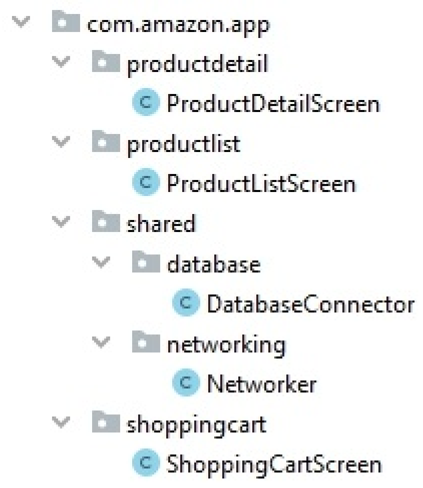
What does this mean? A feature is, well, a feature of your software. Something you provide that is valuable. For a piece of software that the user actually sees, a feature might be a screen. Or collection of screens.

Imagine you’re creating a mobile application for Amazon, the future 4th branch of the U.S. government.

In a package by feature approach, we might have one top-level package for each screen (and its related classes). Then, a package for everything else.

- Product List
- Product Detail
- Shopping Cart
- Account Settings
- Shared
    - Database
    - Networking

That’s the structure in plain English (I like English). Here’s what it might look like using all those packaging conventions we just went over:



```
com.amazon.app
    productdetail
        ProductDetailScreen
    productlist
        ProductListScreen
    shared
        database
            DatabaseConnector
        networking
            Networker
    shoppingcart
        ShoppingCartScreen
```

Note that Amazon’s app would be wayyyyyyyy more complex than this. Also I don’t work for them, *and* I don’t know how they structure their code. But this is a simplistic example of a “package by feature” approach.

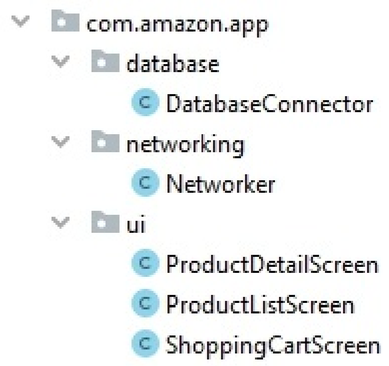
Each feature gets its own big boy package, and anything that isn’t quite a feature is thrown together somewhere else.

Package by layer

Sure, package by “layer” … or whatever you want to call it. By functionality, by purpose, etc.

In the last approach, “package by feature”, we had a package for each main screen of the app…for each “feature” that we provide to the user.

In this approach, “package by layer”, we group by the *kind* of class it is. For example, is the class part of the user interface? It goes in the userinterface (ui) package.



```
com.amazon.app
    database
        DatabaseConnector
    networking
        Networker
    ui
        ProductDetailScreen
        ProductListScreen
        ShoppingCartScreen
```

Do you feel me? Are you feeling me?

“package by feature” – group things together by the section of your app or whatever you would consider a “feature” of your program

“package by layer” – group things together solely by their function, whether that be user interface, networking, helper classes, creating files, math, that kind of stuff.

Let’s say there’s some networking related code I want to add that only affects the ProductDetailScreen. Where would it go?

In Package by Feature, it goes in the *productdetail* package.

In Package by Layer, it goes into the *networking* package.

Imports

“It is clear our nation is reliant upon big foreign oil. More and more of our imports come from overseas.”

– *George W. Bush*

This quote has nothing to do with anything.

But that last section took fucking forever and I want to reset our minds here

But that last section took fucking forever and I want to reset our minds here. Whew. Okay…

How do we let our classes interact with each other? What if the ProductListScreen doesn't know what products to show without first making a network call to Amazon? Surely it needs to ask Amazon for recommendations seeing as they know everything about every aspect of our lives.

Can we just do something like this inside our ProductListScreen?

```
Networker networker = new Networker();
Response networkResponse = networker.getProductsToDisplay();
…
```

It seems reasonable, right? We know that we have a Networker class in our project. Why wouldn't we be able to create an object from that class? Why can't we just "instantiate" it willy nilly?

Because Java needs you to "import" it. Unless the class you want to use is in the exact same package as yours, you must put an import statement at the top of your Java file.

Remember package declarations? It's the simple one-liner where you specify what package your class belongs to.

Right below that, still at essentially the top of the file, you simply import all the classes that you'll need from outside this file.

```
package com.amazon.app.productlist;
import com.amazon.app.shared.networking.Networker;
```

Now, for the rest of the class you can create new Networkers! Go crazy! Connect to every web site that has ever existed! WEoooohhhhhh!

Ahem.

So yeah, that's how you would import classes from the rest of your project. But this same rule applies to using classes from Java.

Just for shits and giggles, suppose I want to generate a random number. Rather than write my own class or function to do this, I can use the **Random** class from the Java standard library.

But I can't just create a new *Random(),* just like that! I have to import it. At the top of my file…

```
import java.util.Random;
```

Then in one of my functions, I’m good to go!

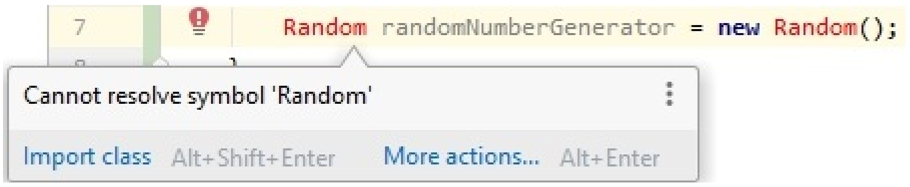
```
Random randomNumberGenerator = new Random();
int randomInt = randomNumberGenerator.nextInt();
```

Hells yea!

I have the memory of a goldfish

So I need to remember the full package name for every class I want to use?

NOPE! You literally just need to know the name of the class, start typing it in your code, then use your super duper IDE like IntelliJ (promo code Steve) to automatically import the class you need.



```
Random randomNumberGenerator = new Random();
```

Cannot resolve symbol 'Random'

Import class. Adds the import to the top of your class. Boom, done.

Importing it all

See above how we imported the Random class? We could import every single class inside the java.util package using an astric *.

```
import java.util.*;
```

Yep. Now this class could use any class inside the java.util package.

There’s pretty much no reason to do this thanks to the goldfish-proof ways of managing your imports that IntelliJ and other software provides. But it’s certainly *possible* to import like this.

The exception to the rule?

There’s a class we’ve been using a lot throughout this book that we’ve never had to import.

Can you think of what it is?

It rhymes with “I can’t remember a goddamn thing.”

and with “Ding ding ding!”

That’s right, it’s a **String** !! Why haven’t we had to import Strings?

I just said a bit ago that if a class isn’t inside your package, you need to import it before you can use it. And we know that a String is a class, because we went over its most important methods. What gives? Am I a liar?

Java pulled a little sneaky on us. They quietly import one special package into *every* Java file. That package’s name?

Albert Einstein.

No, it’s ***java.lang*** ;

Strings belong to the java.lang package. That means you don’t need to import it.

Fun fact

You could create a String like this if you wanted to make everyone around you cry:

```
String oofOuchOwie = new java.lang. String( "Why?" );
```

These are the rules

Just to reiterate, for every Java file:

1. The name of the file should match the Java class, with “.java” at the end.
2. The file should be placed in a folder structure that matches that package name (just the folders separated by a period).
3. The top of the file should state the package the class belongs to.
4. Underneath the package statement, the file should import every class it will need to use.

CHAPTER 13: OBJECT-ORIENTED CONCEPTS

To wrap up this little book, I'll give you a brief rundown of some of the software "design" principles that make a language "Object Oriented," just because I like you that much.

Abstraction

You go to a restaurant. You ask for a very nice, expensive meal. The waiter leaves and returns 20-30 minutes later with your meal.

Where did the waiter get the food from? Did he make it himself? Did he go to the store and buy it? Did he run to the restaurant across the street and take it from there?

When you ask for a meal, you don't walk into the kitchen, tell the chefs how to prepare it, micromanage how they interact with each other, put the food into the oven yourself, or swing by Walmart to buy the buns the restaurant has run out of, etc.

You simply ask the waiter for the food; with the expectation they will provide it to you.

The waiter has ***abstracted away*** the details of how the food is prepared.

Good programmers create classes like this. You might have a very large, powerful program with dozens or hundreds of classes to do something super complicated. Maybe you have a program that calculates the quickest way to travel between Point A and Point B on a map.

But does your user interface need to directly talk to all these different classes to calculate the route? Or can we create a class that ***abstracts*** away all that complexity? Maybe one class that simply takes address 1 as a String, address 2 as a String, then returns an array of the steps you need to take.

This is what abstraction is all about: simplicity of the design that is exposed to other classes or systems.

Encapsulation

We can *enforce* abstraction. Classes can simply refuse to expose themselves out in public. And thank god for that.

You could be a class with 500 properties, 1500 methods, and the worst spaghetti code in the entire world. But you know what? Other classes don’t need to see that. They don’t have to care. Encapsulation is a way of abstracting away our code from other classes.

Let’s say we run a pizza business that lets the customer track the progress of their order. That customer uses an app installed on their phone.

The app needs to make some network calls over the internet to get updates. No, you don’t need to worry about that. And that’s part of the point.

Assume that every class used in this stupid example is in a totally different package.

```
class StatusScreen {
    private Order order ;
    private Networker networker ;

      public void showOrderStatus() {
       String currentStatus = networker .getOrderStatus( order .getId());
       updateProgressBar(currentStatus);
   }

      private void updateProgressBar(String status) {
        //update order progress bar
    }
}
```

private is saying that no other class knows how to update the progress bar on the screen. Only the StatusScreen can call this method. It’s invisible to the rest of them. Consider private properties and methods to be what is “abstracted away” from the classes that would interact with it.

public is saying that any other class can call this method. It's possible we'd have another part of the user interface that wants to tell this screen to show the order status. And luckily for us, it would appear that Networker's *getOrderStatus* method is public, since we're able to use it.

And that's correct; this is what Networker might look like:

```
public class Networker {
    public String getOrderStatus(String id) {
      establishNetworkConnection();
      makeRequest();
      processResult();
       return getStatusFromResult();
   }

      private void establishNetworkConnection() {}
    private void makeRequest(String id) {}
    private void processResult() {}
    private String getStatusFromResult() {}
}
```

We get to call the ***public*** method *getOrderStatus* from our user interface class. And we have no idea that all this complicated networking shit is happening under the hood.

This is the beauty of encapsulation; we're forcing abstraction. Other classes can't fuck up our shit if they aren't even allowed access to it.

private and ***public*** are access modifiers. There's two other ones.

If you *don't specify one* , your class, method, or property is considered ***package private*** . It's basically the default. Other classes in the same package can see you.

protected is a specialized version of that. Except a child class can see that method or property as well.

Inheritance

Which "inheritance" would you rather have?

A. Several million dollars in savings from your beloved parents who were responsible with their money and left it to their children.

B. Some of their traits, like hair color, height, personality… hyperactive depression…ability to ride a unicycle, wiggle their ears….

Option **B** , right? Obviously! Who needs money, not me right? Ha. Ha haha.

Please rate the book 5 stars and like, follow, and subscribe for more content.

What is inheritance?

Defining a class that is a more specific version of another class. Defining a class that is the “child” of a “parent” class. A way to reuse code without copy-pasting it. Another way to model our code in an easy-to-understand way for humans.

In Java, a class does not find another class, hook up, and just *have* a child. Unfortunately enough for them…

The child *is* a specialized version of the parent. The child *inherits* the properties and methods of the parent. The child is a “subclass” of the parent.

When you’re thinking through how you want to write a program, whether that be a website, app, database utility, video game, or college assignment, you often will find many parts of the program will share similar traits.

Maybe your app will have a bunch of different screens that share the same color, or toolbar, or will display information in a similar fashion. No need to do the same thing in multiple places; declare a class called Screen, then extend that class for each screen in your app.

Maybe your video game will have a bunch of different types of heroes and enemies, all of which can deal damage, take damage, die, revive, party up, use abilities, etc. You can model your code carefully using classes and inheritance to minimize how huge and confusing your codebase can become.

How do inheritance?

I’ll start really simple for our first example.

```
class Meme {


}
```

Inheritance, we did it!

Wait, what… who is inheriting what? That looks like an empty class

Wait, what…who is inheriting what? That looks like an empty class.

Plot twist! *Every single class* we've made so far has inherited from another class: ***java.lang.Object*** !

You could explicitly inherit from the Object class for no reason, if you wanted to.

```
class Meme extends Object {

}
```

But don't do that. Java does that for us.

Also, yes, Java says ***extends*** rather than "inherits from." Don't try to extend your parents, whatever that means.

Coitus

Let's create some actual inheritance of our own:

```
class Monster {
    int health = 10 ;
    int damage = 2 ;

      void attack(Monster monsterToAttack) {
      monsterToAttack. health -= damage ;
   }
}
```

Here, we have a simple Monster class. Any Monster created from this class starts off with 10 *health* , 2 *damage* , and the ability to attack another monster.

But we're striving to make the greatest monster-themed video game of the 21st century. One to rival the likes of Pokemon or The Witcher!! Not all of them will be the same!

Let's make this more interesting; specialized, child Monster classes!

First, a *Goblin* . It will have lower health, and lower damage, because fucking Goblins are always the weakest of the bunch.

```
class Goblin extends Monster {
   Goblin() {
       health = 5 ;
```

```
        damage = 1 ;
    }
}
```

What we did here was create a new class called *Goblin* that inherits from *Monster* . It has its own constructor that specifies its unique values; 5 *health* , and 1 *damage* .

By the magic of inheritance, it also can *attack* , because it is a Monster. And the Monster class is the one who defined that method.

Yep, that means you could do this:

```
Goblin warts = new Goblin();
warts.attack(someOtherMonster);
```

Just like that! The Goblin can fight! Woo!!!

Let’s make another one! This one will be stronger; a big tough golem.

```
class Golem extends Monster {
    Golem() {
        health = 100 ;
        damage = 9 ;
    }
}
```

Just like before, with the Goblin, we don’t need to define those *health* and *damage* properties again; the parent already did. We’re just assigning them values of our own.

OK, one more. This one will do more than just have special health and damage; it will do something different when it attacks!

```
class Zombie extends Monster {
    Zombie() {
        damage = 3 ;
    }

      void attack(Monster monsterToAttack) {
        super .attack(monsterToAttack);
        health += damage ;
    }
}
```

Look at its attack method! Not only does it attack another monster, but it heals itself when it does damage!! It's a zombie, get it? It's consuming brains or something? Hah!

Super…

You probably noticed something a bit strange in the Zombie's attack method. Instead of just healing itself, as zombies might do, it says *super.attack()* first. What's that business?

Child classes that want to specialize how they behave (their methods) may want to do some stuff *in addition* to what their parent does. Those child classes must tell the parent to do its thing first.

When a method first calls a ***super*** method, that is the way of saying "I'm first going to delegate to my parent here, they know what to do." You know, that thing you never do in real life.

On the other hand, child classes that want to completely *replace* the parent functionality can simply *never* call their parents using *super* . If we did that in the Zombie scenario, the zombie would heal itself but deal no damage (because the parent Monster method's code was never ran).

Fight to the death

MORTAL KOMBAT!!!

```
Goblin goblin = new Goblin();
Golem golem = new Golem();


while (goblin. health > 0 && golem. health > 0 ) {
    goblin.attack(golem);
    golem.attack(goblin);
}
```

Honorably, these two will fight until either of their *health* values reaches or falls below 0.

Do note that child classes can add new methods of their own; they don’t only have to modify the parent’s methods; they can add entirely new suites of functionality.

Real world examples

I’m a bit biased as an Android developer, but I think these are some relevant examples.

- One part of a screen in Android is a **Fragment** . We can *extend* Fragment to create one of our own, maybe a ListFragment, and wire that user interface up to our list of reasons to hate ourselves.
- The entire screen itself is often an **Activity** . We can extend Activity, maybe with a MainActivity, to start adding our own fragments to the app, using the parent Activity’s methods!
- An input field that the user can edit, an **EditText** , extends from a **TextView** , which just displays text. An EditText can do everything a TextView can *and more* …mostly editing that text.
- We can create a database on our Android device by extending the **RoomDatabase** class. No need to embarrass ourselves trying to write a local database from scratch, we just have to inherit from this class then follow the rules.

That last example? Look at how much code your special database class automatically has , just from inheriting it. Hell yeah! Inheritance!

There’s a lot more to uncover here with inheritance honestly. Want to learn more about it from me? We’re gonna need a bigger book.

Polymorphism

Probably the biggest and scariest word for people learning Java. And honestly, I would not be surprised if a lot of people that program in Java *full time* don’t even know what it means.

And that’s not a knock against them. It’s both a tricky concept and a confusing word. Okay, maybe slightly a knock against them.

But not you. You will know. I will force you to learn... if you voluntarily read.

What is polymorphism?

Java is smart sometimes. One thing it's smart at is knowing which method to call.

It probably sounds easy. I have a class, and it has a method. I create an instance of that class, and call that method. How could a programming language get confused?

When inheritance is involved, that's when.

Let's harken back to our Monster example. We wrote some code that looked like this:

```
Goblin warts = new Goblin();
```

But we can throw a massive curveball here. See how the data type of the *goblin* variable is **Goblin** ? And the constructor we're using is *Goblin()* ?

Since a Goblin is a Monster, we can instead do this:

```
Monster warts = new Goblin ();
```

Whoooaaaa! They don't have to be the same? Nope!

The data type on the left does not have to be the same as the constructor you call on the right. The one on the right can be a child's constructor.

Why polymorphism?

What does this get us? It's a bit like abstraction; we can abstract away what is *behind* the mask.

Remember the fight method on Monster?

```
public void attack(Monster monsterToAttack) {
   monsterToAttack. health -= damage ;
}
```

There's a Monster being passed to this method. Is it a Goblin? Golem? Zombie? Some other kind of Monster?

We don't need to give a single shit. We can treat them all like Monsters; Java will call the appropriate methods on the children without us needing to care.

Can you imagine how annoying it would be if there were different types of Lists, Monsters, Heroes, Screens, Coffee, etc. and you had to be able to account for all of them individually?

Java uses polymorphism to allow us to code against the parent data type of a class, while it takes care of dynamically running the methods of the child implementation of that class.

Splash

```
class Magikarp extends Monster {
   Magikarp() {
        damage = 5000 0 ;
   }
    void attack(Monster monsterToAttack) {
       System. out .println( "Magikarp used SPLASH! But nothing happened!" );
   }
}
```

Check that out. It’s a new kind of monster that does something totally different when it attacks; nothing!

Now, what happens if we run this code?

```
Monster fishy = new Magikarp();
fishy.attack(goblin);
```

- Do we fail to compile?
- Do we splash and do 0 damage?
- Do we attack and deal 50000 damage?
- Does Java call the “attack” method on the parent class Monster, or the “attack” method on the child class Magikarp?

My head. My poor, poor head.

The answer is we splash around and nothing happens, of course! Note that the *data type* of fishy is **Monster** . But we created a *new Magikarp()* .

Again, this ability for Java to call the child class’s method, when the variable is created using the data type of the parent, is called ***polymorphism*** .

Real examples of polymorphism?

One incredibly common use of polymorphism in Java is **Lists** . I briefly mentioned these in the array section…to say I wouldn't be covering them. But for this section, I will if only for a moment.

```
List firstList = new ArrayList ();
List secondList = new LinkedList ();
firstList.add( new Goblin());
secondList.add( new Goblin());
```

These two "implementations" of a List, ArrayList and LinkedList , manage their contents *very* differently; one has an array under the hood, the other has a confusing series of nodes.

But it doesn't matter; once you've declared those variables as a Lists, you can interact with them in the same way.

Another one, this one Android related:

I create several screens in my Android app that extend from the **Activity** class. Android itself might store these as an array of Activities:

```
Activity[] currentActivities ;
```

When the app is closed, Android loops through those activities, and lets us know the screen is being destroyed.

```
for (Activity activity : currentActivities ) {
   activity.onDestroy();
}
```

Java and Android don't care if it's a screen for translating text, or a screen that shows your current GPS coordinates. It doesn't care if it has a million lines of code, or 3. It doesn't care if your subclass even does anything special in its *onDestroy* method.

Does your child class override the *onDestroy* method? It will be called. If it doesn't, the parent class's *onDestroy* method will be called.

The ability to refer to the parent class, but create the object using the child, is super cool and very common. And very polymorphic.

CHAPTER 14: BRINGING IT TOGETHER

What have we learned? Did we hold all this information in our noggins? Well, I did. I wrote it. I already knew this. I did 0 original research and cited 0 sources.

Anyways, if this were a high school English paper with a 1-3-1 format, it would be the final "1" conclusion part.

What's the big idea?

~~Java is a piece of shi~~

Java lets us write programs that often feel like more of an art project than any sort of mathematical process.

The goal is to create a piece of software that accomplish what it sets out to do, whether that be a mobile app, website, or piece of shit "Hello, World" app that is as straightforward, easy to read, *and* functional as possible.

I believe that Java makes this easy with how it absolutely enforces you use classes, and therefore Objects that keep the code clean and maintainable… at least when in the hands of a good developer.

Even though I flew through some of the low-level parts of Java, I hope you got a general idea of what each basic part of the language is, why it exists, and how you might get started with it.

Remember that Google is your friend, and anything you want to know is a series of frustrating web searches away.

Okay, with that little tidbit out of the way…let's revisit what we've learned…

From the top!

Can I summarize an entire introductory Java book with a numbered list?

"Let's find out." – Mr. Owl, of the Tootsie Roll Tootsie Pop

1. Computers understand "machine language". We don't write this ourselves; we use a programming language to write in a format us mere mortals can understand.

2. Java as a programming language is not directly compiled or converted to machine language like most languages. It compiles to "bytecode" which is run on the Java Virtual Machine, which itself runs on the actual computer. "Write once, run anywhere."

3. Java developers use a Java Development Kit (JDK) to compile their code, and to use stuff provided inside the Java Standard Library, like Strings, networking, math, UI, etc.

4. Java is fiercely devoted to Object Oriented Programming, where code is divided into logical chunks called classes. These classes act as blueprints, from which we create Objects. The objects interact with each other in intuitive ways to create maintainable, easy-to-understand, and valuable software. Hopefully.

5. Objects have properties; nouns like numbers, words, lists, or other data that pertain to the object and describe what it *is* . An Airplane could have a speed, a Book could have a price, a Person could have a Head.

6. In general, variables are things that have values, like a number, piece of text, list, or even an entire Object.

7. You can declare a variable, like int x; then assign it a value later with an equals sign like x=5;. Or you can declare and assign at the same time, like int x = 5;

8. All variables in Java are "strongly typed" meaning they absolutely, positively, must have a data type (like int, char, String, etc) and it can never change.

9. Classes have methods; or things that the object *does* … they're verbs

that can calculate, compare, display, or anything else that is an action. A MemeListScreen might refreshMemes(), a RemoteController might sendButtonPressToTV(), a Driver might checkBlindSpot().

10. The main method is a special method every Java program must have, which is the first method run in every Java program (automatically by the Java Virtual Machine).

11. Strings are classes built-in to Java that are used all the time, with methods like toLowerCase, equals, contains, substring, and trim.

12. There are special characters called "operators" built into Java that perform basic functions, like add (+), subtract (-), OR (||), AND (&&) , compare(> >= < <= ==), and more. From these we can build powerful functions.

13. When a program needs to change what it does based off some condition, like user input, value of a variable, etc. it uses conditionals, or *if* statements. Inside those conditionals, the code only runs if the condition is considered to be true.

14. Conditionals can be made even more special with "else if" block which execute under some secondary condition, or "else" blocks which execute if none of the other conditions are true.

15. Arrays are boxes that can contain multiple values, like an array of Strings, or an array of numbers. They must have the same size forever and ever. The first item in the array is at index 0.

16. Loops are ways to run the same code repeatedly, at long as some condition is true. A while loop just keeps running under that condition, and a for loop is a souped-up version with a built-in way to keep track of how many times you've looped, and what to do after a loop.

17. A class becomes an object through "instantiation" or by creating an "instance" of that class. You use the ***new*** keyword to accomplish this. Note: an instance and an object are the same thing; a class brought to life.

18. When a class is being “instantiated”, its constructor is used to build the class. A custom constructor can be made that is given instructions in the form of “parameters”, or variables passed into the constructor’s parentheses ().

19. Classes are grouped together into packages, such as com.example.mypackage. A class in that package would say “package com.example.mypackage;” at the top of the file.

20. Classes that want to use classes from other packages (very common) must import them first, with a line like “import java.util.ArrayList”.

21. A class can “extend” another class to reduce code duplication and provide a more specific piece of functionality than the parent. This is called inheritance. Dog might extend Animal.

22. We can create a variable with the data type of the parent, but the constructor of the child class. Java will still know to call the child’s methods when the parent’s methods are called. This is called polymorphism.

23. Abstraction is the tactic of exposing only what needs to be exposed about what a class does and how it does it. Every class does not need to see how the other class’s sausage is made.

24. Encapsulation is a way of enforcing abstraction with access modifiers, public/protected/private/default. Most things should be kept private to prevent other classes from seeing them.

25. You are no longer a fucking idiot in Java programming. You might still be a beginner, but you are now prepared to get your feet deeper in the Java swimming pool.

FINAL THOUGHTS

Thanks for reading.

I hope you had a good chuckle here and there and learned something along the way.

If you wouldn't mind leaving a review, I'd greatly appreciate it (positive *or* negative). You can also email me your feedback at scbWriterGuy@gmail.com .

This is my first attempt at writing anything past the required page count for a college essay, so I'll take all the help I can get.

Papa bless, and good luck.