Zip Code Wilmington's Programming in Java

Kristofer Younger

Version 0.3, 2022-11-11

Table of Contents

Colophon

Zip Code Wilmington's Programming in Java by Kristofer Younger

Published in the U.S.A.

November 2022: First Edition

Preface

I'd like to thank the instructors of ZipCodeWilmington for inspiring my series of programming books: Chris Nobles, Roberto DeDeus, Tyrell Hoxter, L. Dolio Durant, and Mikaila Akeredolu. Without them, this book would have remained in "maybe some day" category. My thanks to Dan Stabb and Roberto, who made corrections to the text. I hope you get a chance to code someday, Dan! I especially wish to thank Mikaila: without his brilliant prep session slides as the starting point for this book, I would never have thought a small book on the basic fundamentals of programming would be possible or even useful.

Zip Code Wilmington is a non-profit coding boot-camp in Delaware for people who wish to change their lives by becoming proficient at coding. Find out more about us at https://zipcodewilmington.com

About this book

This book's aim is to provide you with the most basic of fundamentals regarding Java, the programming language we have been teaching at Zip Code Wilmington for more than 8 years. To someone who has spent some time with programming languages, this might be just a breezy intro to Java. If you have almost any serious coding experience, this book is probably too elementary for you. You might, however, find the ideas in the Appendices interesting. There I've added some material that have a few advanced ideas in them, plus there is a full code listing of a mars lander simulation.

Java: Pretty Easy to Understand

Java is a fairly easy programming language to learn, and we're going to use it in this book to describe the basic fundamentals of coding. Its form is modern and it is widely used. It can be used for many purposes, from web applications to back-end server processing. We're going to use it for learning simple, programming-in-the-small, object-orientation, methods and concepts; but make no mistake - Java is world-class language capable of amazing things.

Coding *The Hard Way*.

Zed A. Shaw is a popular author of several books where he describes learning a programming language *The Hard Way*. Zed suggests, and we at Zip Code agree with him whole-heartedly, that the best, most impactful, highest return for your investment when learning to code, is **type the code using your own fingers** [1]

That's right. Whether you are a "visual learner", a "video learner", or someone who can read textbooks like novels (are there any more of these out there?), the best way to learn to code is **to code** and **to code by typing out the code with your own fingers**. This

means you DO NOT do a lot of copy and paste of code blocks; you really put in the work, making your brain better wired to code by **coding with your own typing of the code.**

You're here, reading this, because you're thinking (or maybe you know) that you want to become a coder. It's pretty straight-forward.

You may have heard a friend wistfully dream of making a career at writing. "Oh," they say, "I wish I had time to write a great novel, I want to be a writer someday".

So you can ask them: Did you write *today*? *How many words?* And the excuses flow: "Oh, I have to pick up the kids" "Ran out of time, I'm so busy at work." "I had to cut the grass" and so on. Well, I'm here to tell you that all the excuses in the world don't stop a real writer from writing. They just sit down and do it. As often as they can, sometimes even when they can't (or shouldn't).

Coding, like writing, isn't something you can do when all your other chores, obligations, and entertainments are done. If you're serious about learning coding, you must make time for coding.

Watching hours of YouTube videos *will not* make you a coder.

Reading dozens of blog posts, Medium articles, and books *will not* make you a coder.

Following along with endless step-by-step tutorials *will not* make you a coder.

The only way you're going to learn to code is by doing it. Trying to solve a problem. Making mistakes, fixing them, learning from what worked and what didn't at the keyboard.

Many have heard my often-repeated admonition: **If you coded today, you're a coder. If not, you're not a coder.** It really is as simple as that.

Dedication to the mission

I happen to be among those who feel anyone can learn to code. It's a 21st century superpower. When you code, you can change the world. Being proficient at coding can be a life-changing skill that impacts your life, your family's life and your future forever. Time and time again, I've seen that the ability to learn to code is evenly distributed across the population, but the *opportunity to learn to code is not*. So, we run Zip Code to give people a shot at learning a 21st century superpower, no matter where you come from.

And fortune favors the prepared. Some day, you may be working at a great company, making a decent living, working with professionals in a great technical job. Your friends may say "You are so lucky!"

And you will think: **Nope. It wasn't luck.** You'll know that truly. You got there by preparing yourself to get there, and by working to get there, working *very hard*. Ain't no luck involved, just hard work. You make your own luck by working hard.

As many know, getting a spot in a Zip Code cohort is a hard thing to do. Many try but only a few manage it. I often get asked "what can I do to prepare to get into Zip Code?"

The best way is to start solving coding problems on sites like https://hackerrank.com - HackerRank (among others) has many programming assignments, from extremely simple to very advanced. You login, and just do exercise after exercise, relieving you of one of the hardest of coding frustrations, that of trying to figure out **what** to code. Solving programming assignments is a good way to start to cultivate a coding mindset. Such a mindset is based on your ability to pay very close attention to detail, a desire to continually learn, and being able to stay focused on problem solving even if it takes a lot of grit and dedication.

Spending even 20 minutes a day, making progress on a programming task can make all the difference. Day after day your skills will grow, and before long you'll look back on the early things you did and be astonished as to how simple the assignments were. You may even experience embarrassment at remembering how hard these simple exercises seemed at the time you did them. (It's okay, we've all felt it. It's part of the gig.)

Working on code every day makes you a coder. And coding everyday will help with your ability to eventually score high enough on the Zip Code admissions assessment that you get asked to group and potentially final interviews. And then, well, then you get to learn Java or Python and work yourself to exhaustion doing so. Lots and lots more hours.

Why?

You do that hard work, you put in those hours, you create lots of great code, you'll make your own luck, and someone will be impressed and they will offer you a job. And that is the point, right? A job, doing what you love, coding. Right? RIGHT?

You're Welcome,

-Kristofer

Ready?

Okay, let's go.

[1] check out his terrific work: https://learncodethehardway.org

Chapter 1. Output System.out.println()

Let's start with a really simple program. Perhaps the simplest Java program is:

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

This program just prints "Hello, World!".

System.out.println, in this case, means *outputting* or *printing* the "Hello, World!" somewhere. That somewhere is probably the terminal where the program was run.

We will use `System.out.println` to do a lot in the coming pages. It's a simple statement, one that in other languages might be `print` or `write` or `puts` (and why we all cannot just agree on one way to do this, well, beats me. Java's is `System.out.println`)

Here's your second Java program:

```
public class Main {
  public static void main(String[] args) {
    double milesPerHour = 55.0;
    System.out.println("Car's Speed: " + milesPerHour);
  }
}
```

Now, how do we `run` this program? Use a REPLI.it!!

Go to `https://replit.com/` and get a free account. Create a Repl and choose the language to be Java.

Whenever you see code that starts with the line `public class Main {`, that means you can copy and paste the entire snippet right into your Java `replit` and click the big green **Run** > button to run the code and see the output.

(I'll add some images here of what the replit looks like).

This is an example of the output of the replit.

```
> sh -c javac -classpath .:target/dependency/* -d . $(find . -
type f -name '*.java')
> java -classpath .:target/dependency/* Main
Car's Speed: 55.0
>
```

Cool, huh? The ability to communicate with you is one of Java's most fundamental capabilities. And you've run two Java programs. Congratulations, you're a coder. (Well, at least for today you are.)

Chapter 2. Comments

While you're not thinking about the long term, or about large Java programs, there is a powerful thing in Java that helps with tracking comments and notes about the code.

In your program, you can write stuff that Java will ignore, it's just there for you (or readers of your code). We use two slashes to start a comment, and the comment goes to the end of the line. Java will ignore anything on a line after two forward slashes. "//"

```
// this is a comment. it might describe something in the code.
int x = 53 - 11;

int y = x * 4; // this is also a comment.
```

Often, you'll see something like this in this book.

```
public class Main {
  public static void main(String[] args) {
    double flourAmount = 3.5;
    System.out.println(flourAmount); // -> 3.5
  }
}
```

That comment at the end of the System.out.println line is showing what you can expect to see in the output. Here it would be "3.5" printed by itself. Try it in your bookmarked Repl.

We can also add useful stuff to the .log call.

```
public class Main {
  public static void main(String[] args) {
    double flourAmount = 3.5;
    System.out.println("We need " + flourAmount + " cups of
flour."); // -> 3.5
  }
```

```
}
```

See how Java types it all out as a useful phrase? That proves to be very handy in a million-plus (or more) cases.

Comments can be used to explain tricky bits of code, or describe what you should see in output. Comments are your friend.

Chapter 3. BUT WAIT!!!

What does `int` and `double` signify? Well, see how they proceed another word?

```
public class Main {
  public static void main(String[] args) {
    int loavesToMake = 2;
    double flourAmount = 3.5;
//  ^^^^^^ ^^^^^^^^^^
    System.out.println("We need " + flourAmount
      + " cups of flour to make "+loavesToMake+" loaves."); // ->
3.5
  }
}
```

It is our way of telling Java what `flourAmount` is. We want `flourAmount` to be place holder for the number `3.5`, because makes the code easier to read and understand. See much more on this, up ahead, in `variables`.

3.1. Two kinds of Numbers in Java

There are times when I cannot believe I have to make this point. It just seems like, "well, if Java is so smart why do I have to care about this."

What am I complaining about?

Well, to make a strange distinction between **two** types of numbers, Java has a bunch different types of numbers.

In Java, we have to decide what **type** of number we are going use with a variables like *flourAmount* and *loavesToMake*.

If you're just going to *count* things (like say the number of birthdays a person has had), we will use *integers* for that. *integers*

in Java, are called **int**. A `int` means a number, but just an integer like 0, 1, 2, 42, 99, etc. It even means negative numbers -5 or -123487262. The math teacher might say that *integers* are *whole numbers*.

But a **lot** of things need a lot more flexibility. The value of your debit account could be $105.55, which is a number *between* 105 and 106. The amount of fuel in your car's gas tank isn't 1 or 6 gallons, it's probably something like 7.42 gallons, and changes by small amounts as you drive.

So Java has another **type** of number, a **floating point number** and in these early days in this book, we will use **double** as the kind of number we use when we need a number that might not be a *whole number*, but rather *real numbers*. A **double** can have decimal point and some number after, so I can write `3.5` or `42.0`. Or even more complicated numbers like:

```
3.14159
1000000.000001
7.5
55.0
14.0506
0.00678
```

We will use throughout this book, two *types* of numbers. **int**s which are whole numbers like

```
2
0
-16
4560072615728
```

And there are **double**s which are real numbers like

```
1.5
```

```
0.0
-7.005
822.98676253
```

And it turns out Java has a few more **types** of numbers, more than just **int** and **double**. I'll list them, but you'll need to go to a larger Java book or website to understand how they are different from *int* and *double*. They are four more **types** of numbers: **long**, **short**, **byte**, and **float**.

Chapter 4. Statements and Expressions

In Java, there are parts of a program and different parts have different names. Two of the most basic (and fundamental) are **statements** and **expressions**.

4.1. Expressions

An **expression** is something that needs to be *computed* to find out the answer. Here are a few simple ones.

```
2 + 2 * 65536

speed > 55.0

regularPrice * (1.0 - salePercentOff)
```

Each of these lines is something we'd like Java to **compute** for us. That computation is often referred to as "evaluation" or "evaluate the expression" to get to the answer. There are two kinds of expressions in Java, *arithmetic expressions* and *boolean expressions*.

Arithmetic expressions are, as their name implies, something that require arithmetic to get the answer. An expression like "5 + 8 - 3" gets *evaluated* to 10. So an arithmetic expression will end up being a *number*.

Boolean expressions result in either a True or a False value. Example: "maxSpeed > 500.0" - this is either true or false depending on the value of maxSpeed. Or "homeTeamScore > visitorTeamScore" which will be true is and only if the Home team's score is greater than the Visitor team's score. Both *scores* are numbers, the result of the **greater than** makes the result of

the epxression a *boolean expression* (either **true** or **false**).

4.2. Statements

A **statement** is just a line of Java. It ends with a ';' (semi-colon).

```
public class Main {
  public static void main(String[] args) {
    // at the Grocery

    double salesTaxRate = 0.06;
    double totalGroceries = 38.99;
    double salesTax = totalGroceries * salesTaxRate;
    double chargeToCard = totalGroceries + salesTax;
  }
}
```

And this is what a Java program looks like. It's just a list of statements, one after the other, that get computed from the top down.

Some of the statements have expressions in them (like totalGroceries * salesTaxRate), while some are just simple **assignment** statements (like totalGroceries = 38.99, where we assign the variable 'totalGroceries' the value 38.99). Don't panic. These are just some simple examples of Java to give you a feel for it. We'll go thru each of these kinds of things slowly in sections ahead.

4.3. Multi-line Statements

In this book, you may see that the code used in examples is longer than can fit on one line in the code boxes. Well, Java doesn't care. That's why it has **semi-colons** ';' at the end of the statements. So to be clear, a statement with long variable names is the same as one with a short name.

```
k = h * kph - (rest / 60);

kilometersCycled = numberOfHoursPedalled * kilometersPerHour -
(totalMinutesOfRest / 60);
```

When you come across code that goes onto multiple lines, do like Java does, read until you find the ';'. It's like a period in an English sentence.

4.4. Block Statement

Very often in Java, we will see a **block** of statements. It is a list of statements inside of a pair of curly-braces "{ }". It acts like a container to make clear what statements are included in the block.

```
if (magePower > 120.0) {
    maxMagic = 500.0;
    lifeSpan = 800.0;
    maxWeapons = magePower / maxPowerPerWeapon;
}

// some more code
```

See those curly-braces? They start and stop the *block*, and contain the statements within. You can also see how the code is indented, but the real key are those braces. You'll see lots of blocks when you're looking at Java code.

And, it's a bad idea, but this code:

```
if (wizardStrength > 120.0) { maxMagic = Wizard.maxpower();
lifeSpan = Wizard.maxlife(); maxWeapons = wizardStrength *
numberOfWands; }
```

is identical to this:

```
if (wizardStrength > 120.0) {
    maxMagic = Wizard.maxpower();
    lifeSpan = Wizard.maxlife();
    maxWeapons = wizardStrength * numberOfWands;
}
```

But the SECOND example is formatted much cleaner, making it more readable and therefore easier to understand. So I encourage you to your code easy to read by keeping it tidy.

Chapter 5. Variables and Data Types

5.1. Variables

In Java, variables are containers used to store data while the program is running. Variables can be named just about anything, and often are named things that make you think about what you store there.

Think of them as little boxes you can put data into, to keep it there and not lose it. If you wanted to track a number, an age of someone, you can create a variable to keep it in.

```
int age = 3;
```

Now, if we wanted to indicate that someone had a birthday, we might add 1 to the variable.

```
int age = age + 1;
```

At this point, the *age* variable as 4 in it.

There are some rules about variable names.

- All Variables must be named.
- Names can contain letters, digits, underscores, and dollar signs
- Names must begin with a letter
- Names can also begin with $ and _ but are often used in special cases
- Names are case sensitive (y and Y are different variables)

- Reserved words (like Java keywords) cannot be used as names
- All variable names must be unique (no two alike)

So this means the following are fine for variable names in Java:

```
x
area
height
width
currentScore
playerOne
playerTwo
$sumOfCredits
_lastPlay
isPlayerAlive
```

And uppercase and lowercase letters are different. So each of these are DIFFERENT variables even though they are based on the same word(s).

```
currentSpeed
current_speed
currentspeed
CURRENT_SPEED
```

So be careful with UPPERCASE and lowercase letters in variable names.

5.1.1. Declaring a Variable

```
int x;
```

This creates a variable **x** that can hold an integer (int) primitive type. But it has NO value assigned to it, so if we…

```
System.out.println(x);  // -> undefined, although it will
probably print zero.
```

If we were to declare a variable named 'cats' and assign it the value 3 (because we have 3 cats?):

```
int cats = 3;
System.out.println(cats);  // -> 3
```

5.1.2. Assign a value to a variable

```
int age = 19;
String name = "James";
System.out.println(name, "is", age, "years old"); // -> James is
19 years old
age = 21;
name = "Gina";
System.out.println(name, "is", age, "years old"); // -> Gina is
21 years old
```

NOTICE how the **first time** we use *age* and *name* we must *declare* those variables. But the *second time* (and any use thereafter) we don't have to put *int* or *String* in front of the name.

This is how we make sure we don't use variables we have not declared (which in Java is a *syntax* error).

5.1.3. Reassigning a value to a variable

```
String x = "five";
System.out.println(x); // -> five
x = "nineteen";
System.out.println(x); // -> nineteen
```

Notice how we don't use "String" again, when we assign "nineteen" to x. We can assign a variable as many times as we might need to.

```
int age = 3;
// have a birthday
age = age + 1;
// have another birthday
age = age + 1;
System.out.println(age); // -> 5
```

Notice here how age's current value is used, added one to it, and re-assigned *back into the variable *age**.

Java, once you declare a variable as an *int*

```
int height = 62.0; // inches maybe?
System.out.println(height); // -> 62

height = "very tall"; // Java WILL NOT LET you do this.
System.out.println(height);
```

Java demands that once you tell it height is a int, you cannot change it to a String. You might think, but hey, *I'm the programmer*, but in fact, Java is protecting you from a common logic error. And in doing so, is shows what we mean by Java being a **strongly-typed language**. Once you declare something to be of a given type, you cannot change the type while the program is running.

5.2. Constants

Constants are like let variables but they contain values that do NOT change such as a person's date of birth. Convention is to capitalize constant variable names.

You make a double or int or String a constant by marking it **final**.

```
final String DATE_OF_BIRTH = "04-02-2005";
DATE_OF_BIRTH = "04-10-2005"; // <-error

final double SPEED_OF_LIGHT = 670_616_629.0; // miles per hour

SPEED_OF_LIGHT = 700_000_000.0; // ERROR, cannot changed things
marked final
```

An attempt to re-assign a value to a constant (something marked with *final*) will fail.

5.3. Data Types

Java can keep track of a number of different kinds of data, and these are known as "primitive data types". There are 5 of them.

- **Number** - there are two kinds of numbers: integers and floating point
  - **Integers** (or **int**) are like 0, -4, 5, 6, 1234
  - **Floating Point** (or **double**) are numbers where the decimals matter like 0.005, 1.7, 3.14159, -1600300.4329
- **String** - an array of characters -
  - like 'text' or "Hello, World!"
- **Boolean** - is either **true** or **false**
  - often used to decide things like isPlayer(1).alive() [true or false?]

It is common for a computer language to want to know if data is a bunch numbers or text. Tracking what **type** a piece of data is is very important. And it is the programmer's job to make sure all the data get handled in the right ways.

So Java has a few fundamental **data types** that it can handle. And we will cover each one in turn.

Create variables for each primitive data type:

- double
- int
- string
- boolean

Store a value in each.

```
// Here are some samples.

// integer
int x = 0;

// boolean
boolean playerOneAlive = true;

// float
double currentSpeed = 55.0;

// string
String playerOneName = "Rocco";

// constant integer

final int maxPainScore = 150000;
```

Now, you try it. Write down a variable name and assign a normal value to it.

```
// Here are some samples.

// integer
int applesPerBushel =     ;

// boolean
boolean isEngineRunning =          ;
```

```
// float
double myGradePointAverage =       ;

// string
String myDogName =         ;

// constant integer

final int floorsInMyBuilding =         ;
```

Chapter 6. Arithmetic Operators

Java can do math. And many early programming exercises you will come across deal with doing fairly easy math. There are ways to do lots of useful things with numbers.

6.1. Basics

Operator	Name	Description
+	Addition	Add two values
-	Subtraction	Subtract one from another
*	Multiplication	Multiply 2 values
/	Division	Divide 2 numbers
%	Modulus	Remainder after division
++	Increment	Increase value by 1
- -	Decrement	Decrease value by 1

Say we needed to multiply two numbers. Maybe 2 times 3. Now we could easily write a program that printed that result.

```
public class Main {
  public static void main(String[] args) {
    System.out.println(2 * 3);
  }
}
```

And that will print 6 on the console. But maybe we'd like to make it a little more complete.

```
public class Main {
```

```
  public static void main(String[] args) {
    int a = 2;
    int b = 3;
    // Multiply a times b
    int answer = a * b;
    System.out.println(answer); // -> 6
  }
}
```

Here we have set up two variables, a and b, for our *operands* and a final *answer* variable. But, it's pretty much the same as the more simple example above.

Using this as a model, how would you write programs to solve these problems?

- Exercise 1: Subtract A from B and print the result
- Exercise 2: Divide A by B and print the result
- Exercise 3: Use an operator to increase A by 1 and print result
- Exercise 4: Find remainder of A of b

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

    int a = 9;
    int b = 3;

    int L1 = b - a;
    int L2 = a / b;
    int L3 = a + 1;
    //or using increment
    L3 = a++;
    int L4 = a % b;
    System.out.println(L1); // -> -6
    System.out.println(L2); // -> 3
```

```
    System.out.println(L3); // -> 10
    System.out.println(L4); // -> 0
    L4 = 10 % b;
    System.out.println(L4); // now -> 1
  }
}
```

6.2. Division and Remainder

We know that we can do regular division. If have a simple program like this, we know what to expect:

```
int a = 6 / 3; // -> 2
int a = 12 / 3; // -> 4
int a = 15 / 3; // -> 5
int a = 10 / 4; // -> 2 (what??)
```

When we are dividing *int_s, we always get an _int* as a result. So, somewhat confusingly, when we evaluate `10 / 4` we get `2`. (and remember in school when the teacher would say, "10 / 4 is 2 with a remainder of 2"?? yes, remainders aare still with us.)

But sometimes, we have a need to know what the *remainder* of an integer division is. The remainder operator % (the percent sign), despite its appearance, is not related to percents.

The result of a % b is the remainder of the integer division of a by b. So as above, if we evaluate `10 % 4` we will get the remainder of `2`.

```
System.out.println( 5 % 2 ); // 1, a remainder of 5 divided by 2
System.out.println( 8 % 3 ); // 2, a remainder of 8 divided by 3
```

Now what's this about '%' (the remainder) operator?

```java
public class Main {
  public static void main(String[] args) {
    int a = 3;
    int b = 2;
    // Modulus (Remainder)
    int answer = a % b;
    System.out.println(answer); // -> 1
  }
}
```

```java
public class Main {
  public static void main(String[] args) {
    int a = 19;
    int b = 4;
    // Remainder
    int answer = a % b;
    System.out.println(answer); // -> 3
  }
}
```

```java
public class Main {
  public static void main(String[] args) {
    int a = 4;
    int b = 4;
    // Remainder

    for (a = 4; a < 13; a++) {
        System.out.println(a, a % b); // would produce
    }
    // output of
    // 4 0
    // 5 1
    // 6 2
    // 7 3
    // 8 0
    // 9 1
    // 10 2
    // 11 3
    // 12 0
  }
```

```
}
```

6.3. Order is Important

A strange thing about these operators is the order in which they are evaluated. Let's take a look at this expression.

```
6 × 2 + 30
```

We can do this one of two ways:

- Say we like to do multiplication *(I know, who is that?)*
  - we would then do the "6 times 2" part first, giving us 12.
  - then we'd add the 30 to 12 giving us 42 [1]
- But say we don't like multiplication, and want to save it for later…
  - we would add 2+30 first, giving us 32
  - and then we multiply it by 6, and, whoa, we get 192!

Wait! Which is right? How can the answers be so different, depending on the order we do the math in? Well, this shows us that the Order of Operations is important. And people have decided upon that order so that this kind of confusion goes away.

Basically, multiply and divide are higher priority than add and subtract. And exponents (powers) are highest priority of all.

There is a simple way to remember this.

6.3.1. P.E.M.D.A.S

Use this phrase to memorize the default order of operations in Java.

Please Excuse My Dear Aunt Sally

- Parenthesis ( )
- Exponents 2^3
- Multiplication * and Division /
- Addition + and Subtraction -

Divide and Multiply rank equally (and go left to right) So, if we have "6 * 3 / 2", we would multiply first and then divide. "6 * 3 / 2" is 9

Add and Subtract rank equally (and go left to right) So if we have "9 - 6 + 5", we subtract first and then add. "9 - 6 + 5" is 8

30 + 6 × 2 How should this be solved?

Right way to solve 30 + 6 × 2 is first multiply, 6 × 2 = 12, then add 30 + 12 = 42

This is because the multiplication is *higher priority* than the addition, *even though the addition is before the multiplication* in the expression. Let's check it in Java:

```
public class Main {
  public static void main(String[] args) {
    int result = 30 + 6 * 2;
    System.out.println(result);
  }
}
```

This gives us 42.

Now there is another way to force Java to do things "out of order"

with parenthesis.

(30 + 6) × 2

What happens now?

```
public class Main {
  public static void main(String[] args) {
    int result = (30 + 6) * 2;
    System.out.println(result);
  }
}
```

What's going to happen? Will the answer be 42 or 72? Paste it into the REPL.it and find out!

6.4. Java Math Class

There is a useful thing in Java called the Math class which allows you to perform mathematical tasks on numbers.

- Math.PI; - returns 3.141592653589793
- Math.round(4.7); // returns 5
- Math.round(4.4); // returns 4
- Math.pow(x, y) - the value of x to the power of y - x^y
- Math.pow(8, 2); // returns 64
- Math.sqrt(x) - returns the square root of x
- Math.sqrt(64); // returns 8

What does "returns" mean?

When we ask a 'function' like sqrt to do some work for us, we have to code something like:

```
public class Main {
  public static void main(String[] args) {
    double squareRootTwo = Math.sqrt(2.0);
    System.out.println(squareRootTwo);
  }
}
```

We will get "1.4142135623730951" in the output. That number (squareRootTwo) is the square root of 2, and it is the result of the function and *what the function sqrt "returns"'*.

Math.pow() Example

Say we need to compute "6^2 + 5"

```
public class Main {
  public static void main(String[] args) {
    double result = Math.pow(6,2) + 5;
    System.out.println(result); // -> 41.0
  }
}
```

What will the answer be? 279936.0 or 41.0?

How did Java solve it?

Well, 6^2 (6 squared) is the same as 6 * 6. And 6 * 6 = 36, then add 36 + 5 = 41.

You'll learn a lot more about working with numbers in your career as a coder. This is really just the very basic of beginnings.

[1] The answer to life, the universe and Everything.

Chapter 7. Algebraic Equations

Some of the most fundamental of computer programs have been ones that take the drudgery of doing math by a person, and making the computer do the math. These kinds of *computations* rely on the fact that the computer won't do the wrong thing if it's programed carefully.

Given a simple math equation like:

a = b3 - 6 and if b equals 3, then a equals ?

In math class, your teacher would have said "How do we solve for a?" The best way to solve for `a = b3 - 6` is to

- figure out what b times 3 is (well, if b equals 3, then 3 times 3 is 9)
- subtract 6 from b times 3 (and then 9 minus 6 is 3)

```
public class Main {
  public static void main(String[] args) {
    // And in Java:
    // a = b3 - 6

    int b = 3;
    int a = b * 3 - 6;
    System.out.println(a); // -> 3
  }
}
```

Now you try it.

Solve the equation with Java...

q = 2j + 20

if j = 5, then q = ?

Take a moment and write down your solution before reading on.

```
public class Main {
  public static void main(String[] args) {
    int q = 0;
    int j = 5;
    q = 2 * j + 20;
    System.out.println(q); // -> 30
  }
}
```

Let's try another...

Solve the equation with Java...

$x = 5y + y^3 - 7$

if y=2, x = ?

and print out x.

My solution is pretty simple.

```
public class Main {
  public static void main(String[] args) {
    double y = 2;
    double x = 5 * y + Math.pow(y, 3) - 7;
    System.out.println(x); // -> 11
  }
}
```

7.1. *Trigonometry*

The word trigonometry comes from the Greek words, trigonon ("triangle") + metron ("measure"). We use trigonometry to find angles, distances and areas.

For example, if we wanted to find the area of a triangular piece of land, we could use the equation **AreaOfaTriangle** = **height * base / 2**

Therefore we just need to create variables for each and use the operators to calculate the area.

Calculate Area of a Triangle in Java Height is 20 Base is 10 Formula: A = h * b / 2

```
public class Main {
  public static void main(String[] args) {
    double height = 20;
    double base = 10;
    double areaOfaTriangle  =  height * base / 2;
    System.out.println(areaOfaTriangle); // -> 100
  }
}
```

Calculate the area of a circle whose radius is 7.7 Formula: area = Pi * radius^2

Hint: You'll need to use a constant Math property!

```
public class Main {
  public static void main(String[] args) {
    double radius = 7.7;
    double area  =  Math.PI * Math.pow(radius, 2);
    System.out.println(area);
    // -> 186.26502843133886 (wow)
  }
}
```

Chapter 8. Simple Calculation Programs

8.1. How far can we go in the car?

Let's create a simple problem to solve with Java.

> Our car's gas tank can hold 12.0 gallons of gas. It gets 22.0 miles per gallon (mpg) when driving at 55.0 miles per hour (mph). If we start with the tank full and carefully drive at 55.0 mph, how many miles can we drive (total miles driven) using the whole tank of gas?
>
> BONUS:
>
> How long will it take us to drive all those miles?

What do we need figure out? We need a variable for our result: totalMilesDriven. So we start our program this way...

```
public class Main {
  public static void main(String[] args) {
    double totalMilesDriven = 0;

    // print result of computation
    System.out.println(totalMilesDriven);
  }
}
```

It's often good to start with a very simple template. If we run that, we will see 0 (zero) as the result, right?

Next step, let's add the variables we know.

```
public class Main {
  public static void main(String[] args) {
    double totalMilesDriven = 0;
    double totalHoursTaken = 0;

    double totalGasGallons = 12.0;
    double milesPerGallon = 22.0;
    double milesPerHour = 55.0;

    // print result of computation
    System.out.println(totalMilesDriven);
  }
}
```

Okay, good. We've added all the stuff we know about the computation. Well, except the part of the *actual computation*.

You probably know that if you multiply the milesPerGallon by the totalGasGallons, that will give you totalMilesDriven. And if you divide the totalMilesDriven by the milesPerHour, you will get the totalHoursTaken.

So let's add those as Java statements.

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

    double totalMilesDriven = 0;
    double totalHoursTaken = 0;

    double totalGasGallons = 12.0;
    double milesPerGallon = 22.0;
    double milesPerHour = 55.0;

    totalMilesDriven = milesPerGallon * totalGasGallons;
    totalHoursTaken = totalMilesDriven / milesPerHour;

    // print result of computation
    System.out.println(totalMilesDriven +" "+ totalHoursTaken);
  }
```

```
}
```

We get as a result 264 miles driven in 4.8 hours. And that's how a simple Java program can get written.

Let's do another.

8.2. The Cost of a "Free" Cat

A friend of ours is offering you a "free cat". You're not allergic to cats but before you say yes, you want to know how much it'll cost to feed the cat for a year (and then, approximately how much much each month).

> We find out that cat food costs $2 for 3 cans. Each can will feed the cat for 1 day. (Half the can in the morning, the rest in the evening.) We know there are 365 days in a year. We also know that there are 12 months in the year. So how much will it cost to feed the cat for a year?

Looking at it, this may be quite simple. If we know each can feeds the cat for a day, we then know that we need 365 cans of food. So we can describe that as

```
public class Main {
  public static void main(String[] args) {
    double totalCost = 0;
    double cansNeeded = 365;
    double costPerCan = 2.0 / 3.0;

    totalCost = cansNeeded * costPerCan; // right?

    double monthsPerYear = 12;
    double costPerMonth = totalCost / monthsPerYear;
    // print result of computation
    System.out.println(totalCost +" "+ costPerMonth);
  }
```

```
}
```

What's going to be the answer? [1] Run it in your Repl window to work it all out.

And let's do one more.

8.3. You Used Too Much Data!

A cell phone company charges a monthly rate of $12.95 and $1.00 a gigabyte of data. The bill for this month is $21.20. How many gigabytes of data did we use? Again, let's use a simple template to get started.

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

    double dataUsed = 0.0;

    // print result of computation
    System.out.println("total data used (GB)" + dataUsed);
  }
}
```

Let's add what we know: that the monthly base charge (for calls, and so on) is $12.95 and that data costs 1 dollar per gigabyte. We also know the monthly bill is $21.20. Let's get all that written down.

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

    double dataUsed = 0.0;
    double costPerGB = 1.0;
    double monthlyRate = 12.95;

    double thisBill = 21.20;
```

```
    // print result of computation
    System.out.println("total data used (GB)" + dataUsed);
  }
}
```

Now we're ready to do the computation. If we subtract the monthlyRate from thisBill, we get the total cost of data. Then, if we divide the total cost of data by the cost per gigabyte, we will get the dataUsed.

```
public class Main {
  public static void main(String[] args) {
    double dataUsed = 0.0;
    double costPerGB = 1.0;
    double monthlyRate = 12.95;

    double thisBill = 21.20;
    double totalDataCost = thisBill - monthlyRate;

    dataUsed = totalDataCost / costPerGB;

    // print result of computation
    System.out.println();
    System.out.println("total data used (GB)" + dataUsed);
  }
}
```

How many GBs of data did we use? Turns out to be 8.25 gigabytes.

Now if the bill was $24.00? How many GBs then? (go ahead, I'll wait...) [2]

[1] totalCost will be $243.33 and $20.28 per month.

[2] total data used (GB) 11.05

Chapter 9. Boolean Expressions

When starting out in programming, the idea of a boolean variable, something that is either just true or false, seems like an overly simple thing… something that feels rather useless.

In fact, booleans in computer code are **everywhere**. They are simple, but also useful in many ways. You've probably heard about how everything in computers is ones and zeros at the lowest level - and that's true. But on this super simple base of `0 and 1` is built all of the power of the internet, and all the apps you've ever used.

When you are coding, you often have to make a choice about what to do next based on some kind of condition or to do something repetitively (over and over) based on some condition. Something like `is there gas in the car?` or `are we moving faster than 100mph?` In real life, these are considered to be YES (or **true**) or NO (or **false**) kinds of questions. If the gas tank is empty, the question results in a FALSE condition. If there is some gas in the tank, then the question's result is TRUE, "yes, there is some gas in the tank."

And while this may seem super simple to you, and it is, it is also very powerful when used in a program.

This idea of a condition that is either TRUE or FALSE, is known as a **boolean expression**. And in Java, they crop up everywhere. They are in *conditional statements* and they are part of *loops*.

Boolean expressions can be very complex, or very simple:

```
playerOne.isAlive() == true
```

might be a key thing to know inside of a game. But it might be more complicated:

```
player[1].isAlive() == true && player[2].isAlive() == true &&
spaceStation.hasAir() == true
```

(here, *&&* means `and`)

All three things need to be true to continue the game. Using boolean expressions, we can build very powerful tests to make sure everything is just as we need it to be.

We also need more kinds of boolean expressions when we are programming. Things like **less than** or **greater than or equal to**, and other **comparison operators** so we can compare things to work out the relationships within our data.

9.1. Comparison Operators

```
int healthScore = 5;
```

We need a way to ask about expressions like "is healthScore less than 7?? (very healthy)" or "is healthScore greater than or equal to 3?? (maybe barely alive?)"

To do that we need a bunch of **comparison operators**.

Operator	Description	Example
==	Equal to	x == 5
===	Equal value and type	x === '5'
!=	Not equal to	x != 55
!==	Not equal to value and type	x !== '5'
>	Greater than	x > 1

Operator	Description	Example
<	Less than	x < 10
>=	Greater than or equal to	x >= 5
<=	Less than or equal to	x <= 5

Each of these can be used to make it very clear to someone reading your code what you meant. Imagine a flight simulator, where you're flying a big, old fashioned airplane. The code that keeps track of the status of the plane might need to be able to make decisions on boolean expressions like:

```
altitude > 500.0   // high enough to not hit any trees!
airspeed >= 85.0   // fast enough to stay in the air.

fuelAvailable <= 5.0   // need to land to refuel!

totalCargoWeight < 6.0  // more than 6 tons and we can't take
off!

pilot.isAlive() && copilot.isAlive()  // both are alive, keep
flying.
```

In Java, ` == ` is used for *equals* in numbers (both *int* and *double* AND *boolean*). But when we compare two Stings, we use `.equals()`, like this `player1name.equals("Roberto")`. In Java you almost **never** want to use `player1name == "Roberto"`, always use `.equals()` instead.

Like the && (and) in the examples above or this last boolean expression with the "&&" in it, we have in Java the ability to combine expressions into larger more complex expressions using `&&` (and) and `||` (or).

9.2. Logical Operators

The **logical operators** are AND and OR, except in Java we use **&&** for AND and || for OR.

Operator	Description	
&&	Logical AND	playerOneStatus == 'alive' && spacecraft.hasAir()
||	Logical OR	room.Temp > 70 || room.Temp < 75

The AND operator, '&&', is an operator where BOTH sides have to be true for the expression to be true.

```
(5 < 9) && (6-3 === 3)  // true
```

See how both expressions on either side of the '&&' are true? That makes the entire line true.

The OR operator, '||' [1], is an operator where if ONE or the OTHER or BOTH boolean expressions are true, the entire expression is true.

```
(5 < 9) || (6-3 === 3) // true

(5 === 4) || (7 > 3)   // true!

(5 === 4) || (6 === 2) // false (both are false)
```

Both sides of a logical operator need to be Boolean expressions. So it's all right to use lots of different comparisons, and combine them with && and ||.

```
// deep in a cash machine application...
(customer.balance() <= 20.00
&&
customer.hasOverDraftProtection() == true)
||
(customer.savings.balance() > 20.0
&&
customer.canTransferFromSavings() )
```

See how these conditions could line up to allow a customer to get cash from the cash machine? Again, this is **why** boolean expressions are important and powerful and why coders need to be able to use them to get the software **just right**.

- Create 2 variables to use for a comparison
- Use at least two comparison operators in Java
- And print them "System.out.println(2 > 1)"

Here is an example:

```
public class Main {
  public static void main(String[] args) {
    double houseTemp = 67.0;
    double thermostatSetting = 70.0;

    System.out.println(houseTemp >= 55.0);
    System.out.println(houseTemp <= thermostatSetting);
    System.out.println(thermostatSetting != 72.0);
    System.out.println(houseTemp > 65.0 && thermostatSetting ==
68.0);
  }
}
```

These log statements should produce TRUE, TRUE, TRUE and FALSE.

[1] shift-backslash '\' on most keyboards

Chapter 10. Strings

Strings are perhaps the most important data type in Java. Many other computer languages have strings, and they are used in almost ALL modern programs. Knowing how to manage them, create and modify them to do what you need them to do, is a "sub-superpower" within Java.

Pay close attention; this stuff is VERY important.

10.1. What is a String?

Think about the words on this page. The text here is made up of a bunch of letters, and spaces. Now, when we write by hand, we don't really think about the space between the words, do we? If we truly ignored the notion of space between the words, wewouldendupwithtextlikethis. And while it is possible to read, our modern eyes are trained on well-edited texts; having no spaces tires us pretty quickly.

So yes, what we see as text in this book is really a series of letters and spaces strung together in a line - line after line, paragraph after paragraph. In modern computing, that kind of data is often called a **String**. It is one of the most fundamental aspects of coding: the manipulation of strings by programs to transform, present or store text in some fashion.

Many programming languages use some kind of quote or double quote to show where strings start and end. There is really no difference between using single or double quotes in Java. So a string like "the quick brown fox" would be a string from the 't' to the 'x'. And notice the three spaces within the string. If they were not there, the string would be "thequickbrownfox". And that's important, because to the computer, if it keeps these two strings around, it doesn't really understand that 'the' and 'quick' are just two common English words. The spaces are there to retain more

of what the human meant.

No, to the computer, each letter, including the space 'letter', is just a piece of data and very important.

String - a string of letters and numbers and spaces and punctuation, kept altogether for some use. Here are some strings for you to consider.

```
"the quick brown fox"
"The New York Times"
"And lo, like wave was he..."
"oops"
"Hello, World!"
"supercalifraglisticexpealadocious"
"On sale for $123.99!!"
"Pi is approximately 3.14159"
"Merge left at the ramp to the right, the restaurant is on the
right"
"He said, \"Wait there is more!\""
```

Think of strings as a tightly packed list. Each item and letter is numbered. The entire string can be "indexed", meaning I can reach in and copy out, say, the fifth letter easily. String indexes are zero-based; therefore, the first character (element) is in index position 0.

```
Index  -> 012345
String -> Hello
```

So here, "H" is at 0, "e" is at 1, 'l' is at 2 & 3, and 'o' is at index position 4. Computers often start numbering things like strings, lists, and arrays at 0, not at one. It's just one of those things: all strings and arrays (which are coming up) start at zero.

10.2. Declaring a string

```
String name = "Wacka Flocka";
```

Now we have a string variable named **name** and it's value is "Wacka Flocka".

10.3. String Properties

A common and often used string property is **length()**.

We can use `.length` to find the length of a string

```
String str = "Wakanda Forever!";
int answer = str.length;
System.out.println(answer); // -> 16
```

10.4. Accessing Characters in a String

As mentioned before, we can reach into a string and copy out the stuff we find there.

```
String word = "Hello";
// Access the the first character (first by index, second by
function)
System.out.println(word[0]); // H
System.out.println(word.charAt(0)); // H
// the last character
System.out.println(word[word.length - 1]); // o
System.out.println(word.charAt(word.length - 1)); // o
```

When you see something like **word[0]**, it is pronounced like "word sub zero". If you have **word[5]**, you would say "word sub five". This is just verbal shorthand for the expression.

10.5. String Concatenation (Joining strings)

This simply means joining strings together using the + operator or the concat( ) method. Either one is commonly used.

```
int price = 20;
String dollarSign = "$";
String priceTag = dollarSign + price; // $20
//or
String priceTag = dollarSign.concat(price); // $20
System.out.println(priceTag); // -> $20
```

Or perhaps a little more useful example:

```
String name = "Mikaila";
int hoursWorked = 12;

String workReport = "Today, " + name + " worked a total of " +
hoursWorked + " hours."
System.out.println(workReport);
```

The output would be:

> Today, Mikaila worked a total of 12 hours.

10.6. SubStrings

Getting a substring is a common operation. This is how we extract the characters from a string, between two specified indices. (Which is why it's important to remember the indexes start at 0.)

*someString.**substring**(start, end)*

A start position is required, where to begin the extraction.

Remember, first character is at position 0. Characters are extracted from a string between "start" and "end", not including "end" itself.

```
String firstName = "Christopher";
```

Now let's use the 3 substring methods on firstName and extract and print out "Chris"

```
String firstName = "Christopher";
System.out.println(firstName.substring(0,5)); // "Chris"
```

Yep. They all print "Chris". (Act impressed... thanks!) BUT, let's try to extract the string "stop" from the name.

```
String firstName = "Christopher";
System.out.println(firstName.substring(4,8)); // "stop"
```

Notice how the arguments to the functions are **slightly** different. This is why it might be best to pick to memorize and use that one.

Let's try a little harder idea...

```
String fName = "Christopher";
```

- Your turn to use the substring/substr/slice method on firstName
- Extract and print out "STOP" from inside the string above
- And make it uppercase! ("stop" to "STOP") [1]

Well?

```
String fName = "Christopher";
System.out.println(fName.substring(4,8).toUpperCase());
```

Want to bet there is also a "toLowerCase()" method as well?

10.7. Summary of substring methods

Take a look at these various ways to copy out a substring from the source string named 'rapper', which contains the string 'mikaila'.

```
String rapper = "mikaila";

System.out.println(rapper.substring(0,4));  // mika
System.out.println(rapper.substring(1,4));  // ika

System.out.println(rapper.substring(4,7));  // ila
System.out.println(rapper.substring(3,7));  // aila
```

We're using each of the three different substring methods to copy out some smaller piece of the 'rapper' string.

[1] You could google how to do this, try "Java string make upper case"

Chapter 11. Arrays

Arrays are a very powerful idea in many programming languages. Let's start with **why** we need them.

Imagine you have a small number of things you want to track. Let's use our vague computer game we've been using for an example. The game has 5 players, friends that get together over the internet to play a dungeon game.

Now, if you're the coder of this game you could keep track of each player's healthScore by have 5 different variables. (for players Zero to Four)

```
int playerZeroHealthScore = 100;
int playerOneHealthScore = 100;
int playerTwoHealthScore = 100;
int playerThreeHealthScore = 100;
int playerFourHealthScore = 100;
```

If we setup these 5 variables, our game can track 5 players! But **we'd have to change the game's code to track SIX players.** Well, that's not good. Kind of silly actually.

To get around this kind of problem we use an **array**. We could ask, "how many players are playing?", and then make an array that size. We know we need to track each player's healthScore, so we create an array:

```
int[] playerHealthScores = new int[]{100, 100, 100, 100, 100};
```

Now, like a *string*, array indexes start at zero.

```
                          //  0   1   2   3   4
```

```
int[] playerHealthScores = new int[]{100, 100, 100, 100, 100};
```

This array is a **data structure** - a way for us to keep track of lots of data in a controlled fashion. (We can make arrays any size we'd like.) If we need to deduct health points from one of the players, we can do something like this:

```
int majorHit = 50;

playerHealthScores[2] = 67;  // player 2 just took a hit!

playerHealthScores[1] = 105;  // player one is getting stronger.

playerHealthScores[3] = playerHealthScores[3] - majorHit;
```

The best way to think about arrays is like all those postal boxes at the post office. Each box has a number on it, and things get put in the box depending on the box number.

Arrays are **indexed** like that. Each array slot has an index number, starting at zero. See how we use the number 2 to *index* into the playerHealthScore array?

Arrays:

- Can store multiple values in a single variable
- Start counting from index position zero
- Elements can be primitive data types or/and Objects

So let's think about an array of donuts for the following examples.

11.1. Declaring Arrays

Declaring and initializing some arrays in Java:

```
String[] donuts = new String[]{"chocolate", "glazed", "jelly"};
```

11.2. Accessing elements of an Array

We use square brackets to get elements by their index. We'll use an array of strings to identify our donuts. Sometimes, we say something like "donuts sub 2" to mean *donuts[2]*.

```
String[] donuts = new String[]{"chocolate", "glazed", "jelly"};

System.out.println(donuts[0]);  // "chocolate" (we could say
"donuts sub zero")

System.out.println(donuts[2]);  // "jelly"
```

11.3. Get the size of an Array

We can use the **length** property to find the size of an array.

```
String[] donuts = new String[]{"chocolate", "glazed", "jelly"};

System.out.println(donuts.length);  // it'll print 3
```

Note: A string is kind of like ARRAY of single characters.

11.4. Get the last element of an Array

If we use the *length* property carefully, we can always get the last element in an array.

```
String[] donuts = new String[]{"chocolate", "glazed", "jelly",
"strawberry"};
```

```
System.out.println(donuts[donuts.length - 1]); // strawberry
```

11.5. Change an element of an Array

We can change the value within an array to anything we'd like, as long as it is the same **type** as the array started with.

```
System.out.println(donuts[2]); // jelly

                          // .    0            1         2
3
donuts[2] = "powdered";   // -> ["chocolate", "glazed",
"powdered", "strawberry"]

System.out.println(donuts[2]); // powdered
```

Chapter 12. Changing the Control Flow

In many of these examples so far, we see a very simple **control flow**. The program starts at the first line, and just goes line by line until runs out of statements.

```
int q = 0;
int j = 5;
q = j * 4 - 20;
System.out.println(q); // -> 0
```

When programs start to get more sophisticated, the *control flow* can be changed. There are various conditional statements, loop statements, and functions that can cause the control flow to move around within the code. Here we see using both a loop and a conditional IF statement to change the flow of control.

```
int q = 0;
int j = 6;
while (j > 0) {
    q = j * 4 - 20;
    System.out.println(q);
    j--;
    if (q > 0) {
        System.out.println("q is still positive");
    }
}
```

This ability to manipulate the control flow of a program is very important when you start developing logic for your apps and programs. Logic in programs depends heavily on being able to manipulate the control flows through the code. Let's take a look at how each kind of statement allows a programmer to change the flow of control in programs.

Chapter 13. Conditional Statements

We have been seeing programs which consist of a list of statements, one after another, where the "flow of control" goes from one line to the next, top to bottom, and so on to the end of the list of lines. There are more useful ways of breaking up the "control flow" of a program. Java has several conditional statements that let the programmer do things based on conditions in the data.

13.1. If statement

The first conditional statement is the **if** statement.

```
if (something-is-true) {
    doSomething;
}
```

Here are a few simple examples.

```
if (speed > speedLimit) {
    driver.getsATicket();
}

if (x <= -1) {
    System.out.println("Cannot have negative numbers!");
}

if (account.balance >= amountRequested) {
    subtract(account, amountRequested);
    produceCash(amountRequested);
    printReceipt(amountRequested);
}
```

Java also has an **else** part to the **if** statement. When the **if**

condition is False, the else part gets run. Here, if the account doesn't have enough money to fulfill the amountRequested, the else part of the statement gets run, and the customer gets an insufficient funds receipt.

```
if (account.balance >= amountRequested) {
    // let customer have money
} else {
    printReceipt("Sorry, you don't have enough money in your
account!")
}
```

Java can also "nest" if statements, making them very flexible for complicated situations. You can also see here how curly-braces make it clear what statements get executed based on which case or condition is true.

```
int timeOfDay = "Afternoon";

if (timeOfDay === "Morning") {
    System.out.println("Time to eat breakfast");
    eatCereal();
} else if (timeOfDay === "Afternoon") {
    System.out.println("Time to eat lunch");
    haveASandwich();
} else {
    System.out.println("Time to eat dinner");
    makeDinner();
    eatDinner();
    doDishes();
}
```

Notice how this becomes a 3-way choice, depending on the timeOfDay.

Write code to check if a user is old enough to drink.

- if the user's age is under 18. Print out "Cannot party with us"
- Else if the user's age is 18 or over, Print out "Party over here"
- Else print out, "I do not recognize your age"

You should use an if statement for your solution!

Finally, make sure to change the value of the age variable in the repl, to output out different results and test that all three options can happen. What do you have to do to make the `else` clause happen?

```
int userAge = 17;
if (userAge < 18) {
    System.out.println("Cannot party with us");
} else if (userAge >= 18) {
    System.out.println("Party over here");
} else {
    System.out.println("I do not recognize your age");
}
```

If statements are one of the most commonly used statements to express logic in a Java program. It's important to know them well.

Chapter 14. Loops

Loops allow you control over repetitive steps you need to do in your *control flow*. Java has two different kinds of loops we will talk about: **while** loops and **for** loops. Either one can be used interchangeably; but, as you will see there are couple cases where using one over the other makes more sense.

The primary purpose of loops is to avoid having lots of repetitive code.

14.1. While Loop

Loop through a block of code (the body) WHILE a condition is true.

```
while (condition_is_true) {

    // execute the code statements
    // in the loop body

}
```

See the code below. In this case, we start with a simple counter in x = 1. Then, after the loop starts, it checks to see if x < 6, and 1 is less than 6, so the loop body gets executed. We print out 1 and then increment x. Then we go to the top of the loop and check to see if x (now 2) is less than 6. Since that's true so we print out 2 and increment x again. This continues like this for three more times, printing 3, 4, and 5.

Then, x is incremented to 6, and the check is made again, 6 < 6 ... well, no that is false. So we don't execute the loop's body and we fall through to the last System.out.println line, and print out x.

```
int x = 1;
```

```
while (x < 6) {
    System.out.println(x);
    x++;
}

System.out.println("ending at", x); // ? what will print here ?
```

While loops work well in situations where the condition you are testing at the top of the loop is one that may not be related to a simple number.

```
while (player[1].isAlive() === true) {
    player[1].takeTurn();
    game.updateStatus(player[1]);
}
```

This will keep letting player[1] take a turn in the game until the player dies. Another way to do something like this is with an *infinite loop*. (No, infinite loops are not necessarily a bad thing, watch.) We're going to use both **continue** and **break** in this example, and we will describe them better after we're done with loops.

```
Game game = new Game();
Player player = game.newPlayer();

while (true) { // <- notice right here, an infinite loop

    player.takeTurn();
    game.updateScores();
    game.advanceTime();

    if (player.isAlive() == true) {
        continue; // start at top of loop again.
    } else {
        break; // breaks out of loop and ends game.
    }
}
```

```
game.sayToHuman("Game Over!");
```

Here, we are using the continue statement to force the flow of control to the top of the loop. We are also using the break statement to break out of the infinite loop, letting us do other things after the player has 'died'.

14.2. Do..While Loop

There is another kind of loop, a **Do..While** loop. Why? well, sometimes you need a loop to go at least once, no matter what and continue until the condition on the loop becomes false. These are used only occasionally, and only in very specific situations.

```
int x = 0;

do {
    System.out.println(x);
    x++;
}
while (x < 5);
```

14.3. For Loop

The **for** loop is more complex, but it's also the most commonly used loop.

```
for (begin; condition; step) {

    // execute the code statements
    // in the loop body

}
```

Here's one where we go from 1 to 5.

```
for(int j = 1; j < 6;  j++){
    // loop body code
    System.out.println(j);
}
```

This loop will print out:

```
// print
1
2
3
4
5
```

Notice how there are THREE parts to the FOR loop's mechanism.

- begin: j = 1 // Executes once upon entering the loop.
- condition: j < 6 // Checked before every loop iteration. If false, the loop stops.
- loop step: j++ // Executes after the body on each iteration.

and

- body: System.out.println() // Runs again and again while the condition is true.

Let's show you another glimpse of the **break** statement.

```
for(int p = 1; p < 6; p++){
    if(p === 4){
        break;
    }
System.out.println("Loop " + p + " times");
}
```

Jumps out of the loop when p is equal to 4.

- Print from 10 to 1 with a for loop and a while loop (hint use decrement)
- Write a loop that prints 1 - 5 but break out at 3

Stretch Goal: S/he who dares wins!

- Go back to Arrays
- Look at an array of donuts
- Create an array of donuts
- Loop through the array of donuts and print each donut string

You can do it, I know you can!

```
for(int x = 0; x < donuts.length; x++){
    System.out.println(donuts[x]);
}
```

If you had something like this, buy yourself a donut, you deserve it.

14.4. Break Statement

Normally, a loop exits when its condition becomes false. But we can force the exit at any time using the special **break** statement.

```
while (true) {

  String cmd = getCommandFromUser("Enter a command", '');

  if (cmd == "exit") break;

  execute(cmd);
```

```
}
System.out.println("Exiting.")
```

Here, you are asking the user to type in a command. If the command is "exit", then quit the loop and output "Exiting", and end the program. Otherwise, execute the command and go around to the top of the loop and ask for another command.

14.5. Continue Statement

The continue statement doesn't stop the whole loop. Instead, it stops the current iteration and forces the loop to start a new one (if the condition allows).

We can use it if we're done with the current iteration and would like to move on to the next one. This loops prints odd number less than 10.

```
for (let i = 0; i < 10; i++) {

  // if true, skip the remaining part of the body
  // will only be true if the number is even

  if (i % 2 === 0) continue;

  System.out.println(i); // prints 1, then 3, 5, 7, 9
}
```

What's interesting here is the use of the remainder operator (%) to see if a number is odd. The expression (i % 2) is zero if the number is even, if not, the number must be odd. You want to remember this trick of how to find odd or even numbers. It's a common programming problem that you will get asked. The continue statement starts the loop over, not letting the System.out.println() to print out the number when it's even.

Chapter 15. Code Patterns

Any experienced coder would say that the ability to see patterns in code, remember them, and learn from them when creating code is another kind of 'superpower'. The following samples are really simple techniques, but they show some common ways of doing things that you should think about and study. In almost all these examples, there may be some missing variable declarations. Just roll with it. If you think about it, I'm sure you can figure out what "let" variable declarations are needed to run the sample in the REPL page.

15.1. Simple Patterns

If you wanted to find the larger of two values, x and y and assign it to 'max':

```
if (x > y) {
    max = x;
} else {
    max = y;
}
```

Related to it, if we have two variables x and y, and we want the smaller in x, and the larger in y.

```
if (x > y) {
    let t = x;
    x = y;
    y = t;
}
```

Do you see the three statements in the block there? That's called a 'swap'. If you need to swap two values in two variables, you just create a quick temporary variable 't' and use it as a place to make

a copy of the first variable's value.

If I needed to make sure a number is always positive (greater than zero), it's easy - this is called taking the "absolute value" of a number.

```
if (n < 0) n = -n;
```

15.2. Loop Patterns

The next few are examples of the handy use of loops to do a bunch of math easily and quickly. Imagine a problem where you have to "add all the numbers from 1 to 100 and print the sum." It might also be expressed as "**sum** all the number from x to y" (where x and y are two integers). Turns out there is a very easy pattern to learn here.

```
int sum = 0;
int n = 100;
for (let i = 1; i < n; i++) {
    sum = sum + i;
}
System.out.println(sum);
```

Now, if you wanted to find the average of a bunch of numbers, that's as easy as taking the sum of the numbers and dividing the sum by the number of numbers (or n).

```
int sum = 0;
int n = 100;
for (let i = 1; i < n; i++) {
    sum = sum + i;
}
int average = sum / n;
System.out.println(average);
```

Pretty easy, yes? And the other common pattern here is doing a **product** of all the numbers from 1 to n. (Let's try 20)

```
int product = 1;
int n = 20;
for (let i = 1; i < n; i++) {
    product = product * i;
}
System.out.println(product);
```

Perhaps you want to print a table of values of some equation.

```
for (let i = 0; i <= n; i++) {
    System.out.println(i + " " + i*i/2);
}
```

15.3. Array Patterns

Arrays are often something that confuses beginning coders. Let's look at some code patterns with arrays that let you see how arrays and loops can work together to get a lot of work pretty easily.

The array we are going to use in all these cases is pretty simple. It's an array of 7 numbers.

```
int a = [ 4, 3, 7, 0, -4, 1, 8];
```

Here how to print out the array, one value per line.

```
for (let i = 0; i < a.length; i++) {
    System.out.println(i + " " + a[i]); // print the index and
value of an array element.
}
```

If we needed to find the **smallest** number in the array, we could do:

```
int min = a[0];
for (let i = 1; i < a.length; i++) {
    if (a[i] < min) min = a[i];
}
System.out.println(min);
```

We should look carefully here. First, notice how I have taken the first element a[0] and made my first 'min' that value. Then, I started at 1 (not 0), to be my first compare. Then we step through the array, looking at each value and if the new value is smaller than the previous one, we update it; otherwise, we just do the next value. [1]

NOW, if you wanted to find the **largest** value in the array, you really only have to change a couple things.

```
int max = a[0];
for (let i = 1; i < a.length; i++) {
    if (a[i] > max) max = a[i];
}
System.out.println(max);
```

Carefully look at the code, comparing to the one above. What's different? Well, for one, we changed the variable from 'min' to 'max'. (But did we need to do that? We could have left it max, but it's cleaner to make the change so people who read it aren't confused.) We also changed the comparison in the 'if' statement from "less than <" to "greater than >" which lets us decide if the new number is larger than the previous largest we found.

In both of these cases, we start with an initial value, then we step through the array, look at each value comparing it to the smallest (or largest) value we have yet found. If we need to update the

'carrying variable', we do; otherwise, we just ignore the value.

What about finding the average of the values in the array? Well, we do it a lot like the average of the series of numbers.

```
int sum = 0;
for (let i = 0; i < a.length; i++) {
    sum = sum + a[i];
}
int average = sum / a.length ; // whoa! lookee there?

System.out.println(average);
```

Yep, the "a.length" is very handy, it has exactly the count of the numbers in the array!

Finally, if we wanted to reverse the values in the array, we could write some code:

```
System.out.println("before:", a);
int n = a.length;
int half = Math.ceil(n / 2);
for (let i = 0; i < half; i++) {
    let t = a[i];
    a[i] = a[n-1-i];
    a[n-i-1] = t;
}
System.out.println("after: ",a);
```

[1] YES, if the array is only one element long, this will fail. But I'm merely trying to show some concepts here. I'd do this differently, if it were to be in some codebase somewhere.

Chapter 16. Return statement

The **return** statement is a very simple one. It just finishes the running of code in the current function and "returns" to the function's caller.

As you have seen, *methods* are used to make code more understandable, cleaner and more organized. Say we have a couple of functions in our program:

```
int minorHit = 3;
int majorHit = 7;

boolean adjustHealth(Player player, int hit) {
    player.health = player.health - hit;

    if (isAlive(player) == false) {
        return playerDead;
    }

    return playerAlive;
}

boolean function isAlive(player, hit) {
    if (player.health >= 20) {
        return true;
    } else { // player has died!
        return false;
    }
}
```

If someplace in our code we were to do something like:

```
// big hit!
continuePlaying = adjustHealth(playerOne, majorHit);

if (continuePlaying == playerDead) endGame();
```

You can see how when we call the function "adjustHealth()" it

returns either playerAlive or playerDead, and we make a decision to end the game if the player has died.

Notice too, you can have multiple return statements in functions, and each one can return a different value if that's what you need.

Appendix A: Additional Java Resources

Here are a series of other resources to go on from this point. Dev.java is a really good one.

Some Java sites for you to explore:

- https://dev.java
    - https://dev.java/learn/getting-started-with-java/
    - https://dev.java/learn/java-language-basics/

If you're looking for more of a professional code tool, use an IDE like vscode: https://code.visualstudio.com (Many people use this these days.)