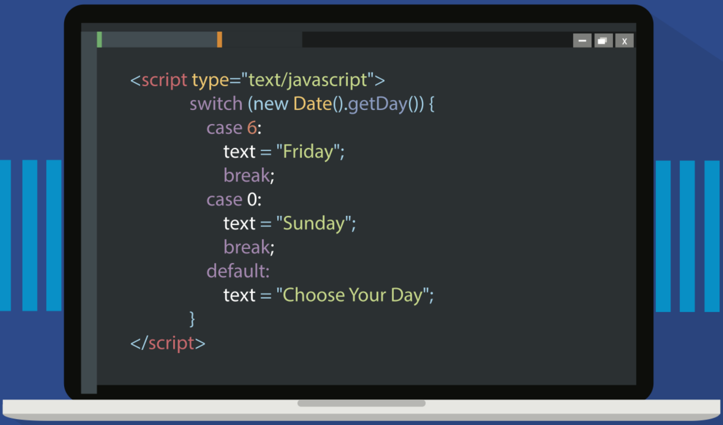
Zip Code Wilmington's Programming in JavaScript

A CRASH COURSE IN CODING



```
<script type="text/javascript">
        switch (new Date().getDay()) {
          case 6:
            text = "Friday";
            break;
          case 0:
            text = "Sunday";
            break;
          default:
            text = "Choose Your Day";
        }
</script>
```

ZIPCODE
<WILMINGTON>
SCHOOL OF CODING

Zip Code Wilmington’s Programming in JavaScript

Kristofer Younger

Version 1.8.8, 2023-09-05

Table of Contents

Colophon

Zip Code Wilmington's Programming in JavaScript by Kristofer Younger

Cover Design: Janelle Bowman

Published in the U.S.A.

September 2023: Third Edition

Preface

I’d like to thank the instructors of ZipCodeWilmington for inspiring this book: 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 JavaScript, the world's most popular programming language. It comes from the preparation sessions we often give prospective Zip Code applicants on how to do well on the Zip Code application coding assessment. By reading this book or taking one of those prep sessions, someone who has never coded in any language before can use this as a place to start, study and get ready to take that assessment. To someone who has spent some time with programming languages, this might be just a breezy intro to JavaScript. 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.

You may be aware that Zip Code doesn't focus on JavaScript *per se*, and that can leave you wondering why the first book we wrote was all about JavaScript. Well…

JavaScript: Easy to Understand

JavaScript 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, methods and concepts; but make no mistake - JavaScript is world-class language capable of amazing things.

Simple to Use

JavaScript also has the advantage that one can learn to use it without a lot of setup on a computer. In fact, we recommend the use of a "REPL" for the running of the JavaScript examples you’ll find in this book. The REPL at https://code.zipcode.rocks is a simple webpage where you can edit JavaScript scripts and run them. You can run almost every code snippet in the book and *see* what the code does.

Focuses on coding

Finally, because in this book all we aim to teach you is "programming in the small", JavaScript is great for that. Many of the examples here are significantly less that 20 lines in length. We want you to get better at looking at small blocks of code to see how they work. These smaller examples and concepts are a core building block as you become proficient in coding.

You’ll learn it eventually

The truth is, in today’s coding world, all of us eventually learn to do things with JavaScript. So, start early, get comfortable with it, and then go on and study other computer languages like Java or Python. JavaScript will always be there, waiting patiently for you to return.

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 console.log()

Let’s start with a really simple program. Perhaps the simplest JavaScript program is:

```
console.log("Hello, World!");
```

This program just prints "Hello, World!". [1]

Logging, in this case, means *outputting* or *printing* the "Hello, World!" somewhere. That somewhere is the *console*, a place JavaScript uses to communicate with a user (in this case, us, the programmers.)

(And if you haven’t done it yet, go to https://code.zipcode.rocks and make a browser bookmark for yourself. Once that’s done, you can use that REPL [2] as a place where you can type in the code from this book and run it to see what each code snippet does.)

We will use `console.log` 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. JavaScript’s is `console.log`)

Here’s your second JavaScript program:

```
let milesPerHour = 55.0;
console.log("Car's Speed: ", milesPerHour);
```

If you typed into your REPL and clicked the "Run" button, you should have seen this as your output:

```
Car’s Speed: 55
```

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

[1] And while you might *not yet* understand this *technical description*, it is a program of one *line* of code, which says "call the 'log' method on the 'console' object, using the string "Hello, World!" as the argument to be logged."

[2] a REPL is short for "read-evaluate-print loop", a special kind of computer program that lets you run code of a given language.

Chapter 2. Comments

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

In your program, you can write stuff that JavaScript 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. Javascript will ignore anything on a line after two forward slashes. "//"

```
// this is a comment. it might describe something
in the code.
console.log('Hello');

console.log('World'); // this is also a comment.
```

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

```
let flourAmount = 3.5;

console.log(flourAmount); // -> 3.5
```

That comment at the end of the console.log 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.

```
let flourAmount = 3.5;

console.log("We need", flourAmount, "cups of
flour."); // -> We need 3.5 cups of flour.
```

See how JavaScript 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. Statements and Expressions

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

3.1. 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 JavaScript 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 JavaScript, *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.

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.

3.2. Statements

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

```
// at the Grocery

salesTaxRate = 0.06;

totalGroceries = 38.99;
salesTax = totalGroceries * salesTaxRate;

chargeToCard = totalGroceries + salesTax;
```

And this is what a JavaScript 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 JavaScript to give you a feel for it. We’ll go thru each of these kinds of things slowly in sections ahead.

3.3. 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, JavaScript 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 JavaScript does, read until you find the ';'. It’s like a period in an English sentence.

3.4. Block Statement

Very often in JavaScript, 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 JavaScript code.

Chapter 4. Variables and Data Types

4.1. Variables

In JavaScript, 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.

There are some rules about variables.

- 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 JavaScript 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 JavaScript:

```
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).

```
Current_Speed
current_speed
CURRENT_SPEED
```

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

4.1.1. Declaring a Variable

```
let x;
```

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

```
console.log(x);  // -> undefined
```

There is *nothing* there.

If we were to declare a variable named 'cats' and assign it the value 3:

```
let cats = 3;
console.log(cats);  // -> 3
```

4.1.2. Assign a value to a variable

```
let age = 19;
let name = "James";
console.log(name, "is", age, "years old"); // ->
James is 19 years old
age = 21;
name = "Gina";
console.log(name, "is", age, "years old"); // ->
Gina is 21 years old
```

4.1.3. Reassigning a value to a variable

```
let x = "five";
console.log(x); // -> five
x = "nineteen";
console.log(x); // -> nineteen
```

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

```
let age = 3;
// have a birthday
age = age + 1;
// have another birthday
age = age + 1;
console.log(age); // -> 5
```

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

Now, one of the weird (to me anyway) things JavaScript can do is change the type of a variable while the program is running. A lot of languages won’t let you do this. But it can be handy in JavaScript. In JavaScript, variables are dynamic (can contain any data) which means a variable can be a string and later be a number.

```
let height = 62.0; // inches maybe?
console.log(height); // -> 62

height = "very tall";
console.log(height); // -> very tall
// yep, first height is a number
// and then it's a string.
```

You can’t see it, but I am slowly shaking my head in disbelief. Some day, maybe I’ll explain why.

4.2. 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.

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

An attempt to re-assign a value to a constant will fail.

4.3. Data Types

JavaScript 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 floats
  - **Integers** are like 0, -4, 5, 6, 1234
  - **Floats** 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?]
- **Null** - no value at all
- **Undefined** - a variable not yet assigned - "let x;"
  - this is a weird type, and not very common.

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 JavaScript 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:

- boolean,
- float,
- integer,
- string
- constant (integer)

Store a value in each.

```
// Here are some samples.

// integer
let x = 0;

// boolean
let playerOneAlive = true;

// float
let currentSpeed = 55.0;

// string
let playerOneName = "Rocco";

// constant integer
```

```
const maxPainScore = 150000;
```

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

Chapter 5. Arithmetic Operators

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

5.1. 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.

```
console.log(2 * 3);
```

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

```
let a = 2;
let b = 3;
// Multiply a times b
let answer = a * b;
console.log(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?

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

```
let a = 9;
let b = 3;

let L1 = b - a;
let L2 = a / b;
let L3 = a + 1;
//or using increment
L3 = a++;
let L4 = a % b;
console.log(L1); // -> -6
console.log(L2); // -> 3
console.log(L3); // -> 10
console.log(L4); // -> 0
L4 = 10 % b;
console.log(L4); // now -> 1
```

5.2. Division and Remainder

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

```
let a = 6 / 3; // -> 2
let a = 12 / 3; // -> 4
let a = 15 / 3; // -> 5
let a = 10 / 4; // -> 2.5
```

But sometimes, we have a need to know what the remainder of a division is. The remainder operator %, despite its appearance, is not related to percents.

The result of a % b is the remainder of the integer division of a by b.

```
console.log( 5 % 2 ); // 1, a remainder of 5
divided by 2
console.log( 8 % 3 ); // 2, a remainder of 8
```

```
divided by 3
```

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

```
let a = 3;
let b = 2;
// Modulus (Remainder)
let answer = a % b;
console.log(answer); // -> 1
```

```
let a = 19;
let b = 4;
// Remainder
let answer = a % b;
console.log(answer); // -> 3
```

```
let a = 4;
let b = 4;
// Remainder

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

5.3. 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.

5.3.1. P.E.M.D.A.S

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

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 JavaScript:

```
let result = 30 + 6 * 2;
console.log(result);
```

This gives us 42.

Now there is another way to force JavaScript to do things "out of order" with parenthesis.

(30 + 6) × 2

What happens now?

```
let result = (30 + 6) * 2;
console.log(result);
```

What’s going to happen? Will the answer be 42 or 72?

5.4. JavaScript Math Object

There is a useful thing in JavaScript called the Math object 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:
>
> ```
> let squareRootTwo = Math.sqrt(2.0);
> console.log(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"

```
let result = Math.pow(6,2) + 5;
console.log(result);
```

What will the answer be? 279936 or 41?

How did JavaScript solve it?

Well, 6^2 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 6. 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)

```
// And in JavaScript:
// a = b3 - 6

let b = 3;
let a = b * 3 - 6;
console.log(a); // -> 3
```

Now you try it.

Solve the equation with JavaScript...

> q = 2j + 20
>
> if j = 5, then q = ?
>
> Take a moment and write down your solution before reading on.

```
let q = 0;
let j = 5;
q = 2 * j + 20;
console.log(q); // -> 30
```

Let’s try another…

> Solve the equation with JavaScript…
>
> x = 5y + y^3 - 7
>
> if y=2, x = ?
>
> and print out x.

My solution is pretty simple.

```
let y = 2;
let x = 5 * y + Math.pow(y, 3) - 7;
console.log(x); // -> 11
```

6.1. *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 JavaScript Height is 20 Base is 10 Formula: A = h * b / 2

```
let height = 20;
let base = 10;
let areaOfaTriangle  =  height * base / 2;
console.log(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!

```
let radius = 7.7;
let area  =  Math.PI * Math.pow(radius, 2);
console.log(area); // -> 186.26502843133886 (wow)
```

Chapter 7. Simple Calculation Programs

7.1. How far can we go in the car?

Let’s create a simple problem to solve with Javascript.

> 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…

```
let totalMilesDriven = 0;

// print result of computation
console.log(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.

```
let totalMilesDriven = 0;
let totalHoursTaken = 0;

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

// print result of computation
console.log(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 JavaScript statements.

```
let totalMilesDriven = 0;
let totalHoursTaken = 0;

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

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

```
// print result of computation
console.log(totalMilesDriven, totalHoursTaken);
```

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

Let’s do another.

7.2. 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

```
let totalCost = 0;
let cansNeeded = 365;
let costPerCan = 2.0 / 3.0;

totalCost = cansNeeded * costPerCan; // right?
```

```
let monthsPerYear = 12;
let costPerMonth = totalCost / monthsPerYear;
// print result of computation
console.log(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.

7.3. 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.

```
let dataUsed = 0.0;

// print result of computation
console.log("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.

```
let dataUsed = 0.0;
let costPerGB = 1.0;
let monthlyRate = 12.95;
let thisBill = 21.20;

// print result of computation
console.log("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.

```
let dataUsed = 0.0;
let costPerGB = 1.0;
let monthlyRate = 12.95;

let thisBill = 21.20;
let totalDataCost = thisBill - monthlyRate;

dataUsed = totalDataCost / costPerGB;

// print result of computation
console.log("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 8. 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 NO 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 JavaScript, 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 AND player[2].isAlive()
=== true AND spaceStation.hasAir() === true
```

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.

8.1. Comparison Operators

```
let 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
<	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()  //
everything is fine, keep flying.
```

If you’re puzzling about the difference between the '===' equals and the '==' equals, until you learn a lot more about JavaScript, just use the triple '===' equals. That goes for the 'not equals' too; always use the '!==' not equals.

Like the ANDs in the examples above or this last boolean expression with the "&&" in it, we have in JavaScript the ability to combine expressions into larger more complex expressions using AND and OR.

8.2. Logical Operators

The **logical operators** are AND and OR, except in JavaScript 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 JavaScript
- And print them "console.log(2 > 1)"

Here is an example:

```
let houseTemp = 67.0;
let thermostatSetting = 70.0;

console.log(houseTemp >= 55.0);
console.log(houseTemp <= thermostatSetting);
console.log(thermostatSetting != 72.0);
console.log(houseTemp > 65.0 && thermostatSetting
== 68.0)
```

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

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

Chapter 9. Strings

Strings are perhaps the most important data type in JavaScript. 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 JavaScript.

Pay close attention; this stuff is VERY important.

9.1. 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 JavaScript. 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.

9.2. Declaring a string

```
let name = "Wacka Flocka";
```

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

9.3. String Properties

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

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

```
let str = "Wakanda Forever!";
let answer = str.length;
console.log(answer); // -> 16
```

9.4. Accessing Characters in a String

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

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

9.5. String Concatenation (Joining strings)

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

```
let price = 20;
let dollarSign = "$";
let priceTag = dollarSign + price; // $20
//or
let priceTag = dollarSign.concat(price); // $20
```

```
console.log(priceTag); // -> $20
```

Or perhaps a little more useful example:

```
let name = "Mikaila";
let hoursWorked = 12;

let workReport = "Today, " + name + " worked a
total of " + hoursWorked + " hours."
console.log(workReport);
```

The output would be:

```
Today, Mikaila worked a total of 12 hours.
```

9.6. 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.) There are 3 methods in JavaScript to get a substring: substring, substr and slice. Let's look at each one. They have very slight differences, so you may want to pick one and memorize what your choice does.

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

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

*someString.**slice**(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.

```
let firstName = "Christopher";
```

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

```
let firstName = "Christopher";
console.log(firstName.substring(0,5)); // "Chris"
//or
let a = firstName.slice(0,5); // "Chris"
console.log(a);
//or
let b = firstName.substr(0,5); // "Chris"
console.log(b);
```

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

```
let firstName = "Christopher";
console.log(firstName.substring(4,8)); // "stop"
//or
let a = firstName.slice(4,8); // "stop"
console.log(a);
//or
let b = firstName.substr(4,4); // "stop"
console.log(b);
```

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…

```
let 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?

```
let fName = "Christopher";
console.log(fName.substring(4,8).toUpperCase());
```

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

9.7. 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'.

```
let rapper = "mikaila";

console.log(rapper.substr(0,4));  // mika
console.log(rapper.substr(1,3));  // ika

console.log(rapper.substring(0,4));  // mika
console.log(rapper.substring(1,4));  // ika

console.log(rapper.slice(0,4)); // mika
console.log(rapper.slice(1,4)); // ika
console.log(rapper.slice(1,3)); // ik
```

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

9.8. Reverse a String

Now let’s reverse the string "STOP" to say "POTS".

To Reverse a String

Step 1 - Use the split() to return an array of strings

Step 2 - Use the reverse() method to reverse the newly created array of string characters

Step 3 - Use the join() method to join all elements into a String

Solution

```
var str = "Christopher";
var res = str.substring(4, 8).toUpperCase(); // ->
"STOP"
var spl = res.split("");  // -> ["S", "T", "O",
"P"]
var rev = spl.reverse();  // -> ["P", "O", "T",
"S"]
var result = rev.join("");  // -> "POTS"
console.log(result); // -> POTS
```

Strings are perhaps the most important data type in almost any language. Being able to manipulate them easily and do powerful things with them in JavaScript, makes you a better coder.

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

Chapter 10. 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)

```
let playerZeroHealthScore = 100;
let playerOneHealthScore = 100;
let playerTwoHealthScore = 100;
let playerThreeHealthScore = 100;
let 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 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:

```
let playerHealthScores = [100, 100, 100, 100,
100];
```

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

```
                      //   0    1    2    3    4
let playerHealthScores = [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.) If we need to deduct health points from one of the players, we can do something like this:

```
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.

10.1. Declaring Arrays

Declaring and initializing some arrays in JavaScript:

```
let donuts = ["chocolate", "glazed", "jelly"];

let arrayofLetters = ['c','h','r','i','s'];

let mixedData = ['one', 2, true];  // a string, a
number and a boolean!
```

10.2. 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]*.

```
let donuts = ["chocolate", "glazed", "jelly"];

console.log(donuts[0]);  // "chocolate" (we could
say "donuts sub zero")

console.log(donuts[2]);  // "jelly"
```

10.3. Adding to an Array

We can also add things to the end of the array.

```
let donuts = ["chocolate", "glazed", "jelly"];

donuts[3] = "strawberry"   // notice there is no
element 3 before this,

console.log(donuts);  // but after, there are now
4 things in the array.
```

10.4. Get the size of an Array

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

```
let donuts = ["chocolate", "glazed", "jelly"];

console.log(donuts.length);  // it'll print 3
```

Note: A string is an ARRAY of single characters

10.5. Get the last element of an Array

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

```
let donuts = ["chocolate", "glazed", "jelly"];
```

```
donuts[3] = "strawberry";    // -> ["chocolate",
"glazed", "jelly", "strawberry"]

console.log(donuts[donuts.length - 1]); //
strawberry

donuts[4] = "powdered"    // -> ["chocolate",
"glazed", "jelly", "strawberry", "powdered"]

console.log(donuts[donuts.length - 1]); //
powdered
```

10.6. Append to the Array

We can also use the `.length` to add to the end of the array!

```
let donuts = ["chocolate", "glazed", "jelly"];

donuts[donuts.length] = "strawberry";
console.log(donuts[donuts.length - 1]); //
strawberry

donuts[donuts.length] = "powdered";

console.log(donuts[donuts.length - 1]); //
powdered
```

Chapter 11. 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.

```
let q = 0;
let j = 5;
q = j * 4 - 20;
console.log(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.

```
let q = 0;
let j = 6;
while (j > 0) {
    q = j * 4 - 20;
    console.log(q);
    j--;
    if (q > 0) {
        console.log("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 12. 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. JavaScript has several conditional statements that let the programmer do things based on conditions in the data.

12.1. 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)
    console.log("Cannot have negative numbers!");

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

JavaScript 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!")
}
```

JavaScript 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.

```
let timeOfDay = "Afternoon";

if (timeOfDay === "Morning") {
    console.log("Time to eat breakfast");
    eatCereal();
} else if (timeOfDay === "Afternoon") {
    console.log("Time to eat lunch");
    haveASandwich();
} else {
    console.log("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?

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

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

12.2. Switch Statement

Switch statements are used to perform different actions based on different conditions.

Here we have a long example, where the user types a command, and the program runs the code for a given command. (This is sometimes referred to as a shell or command loop).

Notice how we first get a command from the user, look at each possibility, and do something specific if we find a matching command string. Otherwise, we print an error message.

```
let lastCommand = getCommandFromUser();

switch(lastCommand){
    case "exit":
        console.log("so long!");
        break;
    case "run":
        console.log("running simulation...");
        runSim();
        break;
    case: "rename":
        renameSim();
        break;
    case: "delete":
        if (makeSureDeleteIsOkay()) {
            deleteSim();
        } else {
           console.log("delete cancelled...");
        }
        break;
    case: "new":
        createNewSim();
        break;
    case: "help":
        showHelpToUser();
        break;
    default:
        console.log("command not found: try again
or type help");
}
```

Switch statements can be quite elaborate (and in this case much better than a whole lot of IF statements).

Here’s the exercise from the *if* section. Oftentimes, you can write a switch statement in *if* statements, or a bunch of *if* statements as a *switch* statement. This time, you should use a **switch** statement.

Write code to check if a user is old enough to drink. (Using a switch)

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

Finally, make sure to change the value of the age variable to output different results.

Chapter 13. Loops

Loops allow you control over repetitive steps you need to do in your *control flow*. JavaScript 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.

13.1. 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 console.log line, and print out x.

```
let x = 1;

while (x < 6) {
    console.log(x);
    x++;
}

console.log("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.

```
let 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'.

13.2. 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.

```
let x = 0;

do {
    console.log(x);
    x++;
}
while (x < 5);
```

13.3. 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(let j = 1; j < 6;  j++){
    // loop body code
    console.log(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: console.log() // Runs again and again while the condition is true.

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

```
for(let p = 1; p < 6; p++){
    if(p === 4){
        break;
    }
console.log("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(let x = 0; x < donuts.length; x++){
    console.log(donuts[x]);
}
```

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

13.4. 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) {

  let cmd = +prompt("Enter a command", '');

  if (cmd == "exit") break;

  execute(cmd);
}
```

```
console.log("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.

13.5. 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;

  console.log(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 console.log to print out the number when it’s even.

Chapter 14. 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.

14.1. 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;
```

14.2. 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.

```
let sum = 0;
let n = 100;
for (let i = 1; i < n; i++) {
    sum = sum + i;
}
console.log(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).

```
let sum = 0;
let n = 100;
for (let i = 1; i < n; i++) {
    sum = sum + i;
}
let average = sum / n;
console.log(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)

```
let product = 1;
let n = 20;
for (let i = 1; i < n; i++) {
    product = product * i;
}
console.log(product);
```

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

```
for (let i = 0; i <= n; i++) {
    console.log(i + " " + i*i/2);
}
```

14.3. 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.

```
let 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++) {
    console.log(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:

```
let min = a[0];
for (let i = 1; i < a.length; i++) {
    if (a[i] < min) min = a[i];
}
console.log(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.

```
let max = a[0];
for (let i = 1; i < a.length; i++) {
    if (a[i] > max) max = a[i];
}
console.log(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.

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

```
there?

console.log(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:

```
console.log("before:", a);
let n = a.length;
let 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;
}
console.log("after: ",a);
```

But perhaps the easier way to reverse an array in Javascript is to just call the library function:

```
a = a.reverse();
console.log(a);
```

It can be useful to look at the "longer" way to continue to get a feel for how to do small, useful things with simple logic.

[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 15. Functions

A function is a block of code designed to perform a particular task. A function encloses a set of statements and is a fundamental modular unit of JavaScript. They let you reuse code, and provide a way for you to organize your programs, keeping them easier to understand and easier to modify. It’s often said that the craft of programming is the creation of a set of functions and data structures which implement a solution to some problem or set of requirements.

Functions are objects, like many things in JavaScript. They can be stored in variables, other objects, or even collected into arrays. Functions are very powerful because in JavaScript, they can be passed as arguments to other functions, returned from functions, and can even have methods attached to them. The most important thing that functions can do is get **invoked**.

You create functions easily.

15.1. Function Literal

Two examples of how you write a function literal:

```
let add = function (parameter1, parameter2) {
    return parameter1 + parameter2;
}
```

```
function add(p1, p2) {
    return p1 + p2;
}
```

In both of these cases, you end up with a function named 'add' that takes two parameters and adds them, returning the result. (Yes, you could just write (p1 + p2) and everyone would understand. It’s just a very simple example.) Here is another example:

15.2. Creating a Function

```
function greetUser(username) {
    console.log( "Hello " + username);
    return;
}

//calling/Invoking the function
greetUser("Mike Jones"); // "Hello Mike Jones"
```

15.3. Invoking Functions

Functions are meant to be *invoked*. So you can have one function call another function which calls a third function and so on; it’s very common.

Imagine we have a program that gets an airplane ready for flight. We can imagine a whole series of functions we’d have to write. Things like 'loadPassengers', 'loadBaggage', 'loadFuel', 'checkTirePressures', and so on. Then we might have a 'higher level' functions which brings all these pieces together:

```
function prepFlight(airplane) {
    loadBaggage(airplane);
    loadPassengers(airplane);

    loadFuel(airplane);
    performPreflightChecklist(airplane.copilot);
    askTowerToDepart(airplane.pilot);
    departGate(airplane);

    taxiToRunway(airplane, mainRunway90);
    takeoff(airplane.pilot, airplane);

    return;
}
```

You can see how functions let you perform different things in a particular order, and while you might have no idea *how* 'loadBaggage' does what it does, you can see how the program preps the airplane object for flight and makes sure everything important is done. Each of the functions from 'loadBaggage' to 'takeoff' is invoked and returns so the next one can be invoked. This is the power of functions - the ability to take some code and put it a function so that it much easier to understand.

15.4. Anonymous Functions

A common pattern used in JavaScript revolves around an **anonymous** function. It’s a function without a name, and one that probably only gets called *once*. Something like:

```
function () {
    let x = 5;
    let y = 4;
    while (x < 100) {
        x = x + y;
    }
    console.log("x is", x);
} () ;
```

Look at that last line. What’s going on here? Well, first we are creating a function object *without a name* (so it’s *anonymous*). Then on the last line, we are *invoking* the function once (see those two parentheses? '()'?), so all the code inside gets run. (x get printed out).

You see this often in JavaScript. It’s a way to put some code into an argument to another function call.

```
let milliSecInASecond = 1000;

setTimeout(function () {
    console.log('We waited for 5 seconds, to print
this')
}, 5 * milliSecInASecond);
```

See how we pass an anonymous function to 'setTimeout' as one argument, and a timer value as the second argument? The idea here is that the anonymous function gets called at 5 seconds in the future. (Like scheduling an alarm to do something in 5 seconds).

15.5. Function Return

Once JavaScript reaches a return statement, the function will stop executing. Functions often compute a return value. The return value is "returned" back to the "caller". You can have many returns in a functions, depending on how the flow of control is changed.

```
function greetUser(username) {
    return "Hello " + username;
}

let result = greetUser("Welcome back, Mike
Jones");

console.log(result); // will print "Hello Welcome
back, Mike Jones"
```

Or like this:

```
function determineWinner(home, visitor) {
    if (home.score > visitor.score) {
        return "Home Team Wins! Let's have a
Parade!"
    } else if (home.score < visitor.score) {
        return "Visitors Win! (oh Well)"
    }
    return "It's a Tie!"
}
```

Notice how in this case, we check to see the scoring results with two conditions (which are, what? yes, *boolean* *expressions*). If neither condition is true, the third one must be the case. But if either condition is true, then we return right away, and the function is done.

Again, to be clear, we might use this function like this:

```
let home = {
    name = "Fightin Cats"
    score = 0;
}

let visitor = {
    name = "Wild Horses"
    score = 0;
}

playGame(home, visitor); // a lot of work done in
this function(!)

// game is done
let result = determineWinner(home, visitor);

// and then print the result..
console.log(result);
```

15.6. Function Parameters

Functions can also take parameters to be used within a function.

```
function addThreeNumbers(a, b, c) {
    return (a + b + c);
}
```

```
function determineWinner(home, visitor) {
    if (home.score > visitor.score) {
        return "Home Team Wins! Let's have a
Parade!";
    } else if (home.score < visitor.score) {
        return "Visitors Win! (oh Well)";
    }
    return "It's a Tie!";
}

function makeNegative(number) {
    if (number > 0) {
        return -(number);
    }
    // already negative, it's less than 0
    return number;
}
```

Remember how we had the expression ot see if a number was even? (x % 2 === 0) Now, here’s a way to decide is number was divisible cleanly by another, it’s a standard arithmetic expression:

```
(number % divisor === 0)
```

So to see if a number is even, we could use '(number % 2 == 0)':

```
console.log((8 % 2 === 0)); // true
console.log((7 % 2 === 0)); // false
console.log((4 % 2 === 0)); // true
```

And we can use the same technique to see if a number is evenly divisible by 3 or 5.

Try to write a function that will perform the following requirements:

- Create a function called zipCoder
- Your function takes one parameter of type number
- Your function checks and does the following
- If parameter is divisible by 3 and 5 (15). Print ZipCoder
- If parameter is divisible by 3. Print Zip
- If parameter is divisible by 5. Print Coder Phew…Finally
- Call the method and pass in 45 as your parameter

OKAY! Write it yourself!

Do it.

Just write it yourself.

C’mon, write your own version first.

No, really.

Wait.

Do you want to be a ZipCoder, or just a Copy-Paste Stylist?

Well, here's one solution:

```
// Function ZipCoder

function zipCoder(aNumber) {
if (aNumber % 15 == 0) console.log("ZipCoder");
else if (aNumber % 3 == 0) console.log("Zip");
else if (aNumber % 5 == 0) console.log("Coder");
}

zipCoder(45); // -> ZipCoder
```

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

```
let minorHit = 3;
let majorHit = 7;

function adjustHealth(player, hit) {
    player.health = player.health - hit;

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

    return playerAlive;
}

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.

Chapter 17. Modules

In JavaScript, modules allow for code to loaded into a program only if it is needed. Modules are one of the advanced topics in JavaScript that we won’t spend too much time on, but here are the basics.

Most of your JavaScript programs are fairly small when you are creating solutions to HackerRank type problems. They can be seen as a simple script you store in a file on your computer. Usually, when they are stored on your computer, they are ".js" files.

```
console.log("Hello, World!");
```

helloworld.js

If your program is much larger, it might be split into different files to keep it all more organized or readable. All those files might be kept in a folder all together, as a project. But again, this is beyond what you need to know to do HackerRank JavaScript problems.

Modules are also used to import code others have written that you wish to take advantage of. There are millions of chunks of JavaScript you can find and use in your code. A lot of it is used by many, many people, and it’s important to know where the code you use comes from. It can be dangerous to use someone else’s code that isn’t trustworthy.

See https://javascript.info/modules-intro for more on modules! Look for information on "import" and "export" to see how modules interact with your code.

Chapter 18. Objects

There are only a few data types in JavaScript. All but one of them are called “primitive” data types, because their values contain only a single thing (be it a string or a number or whatever).

In contrast, **objects** are used to store **keyed** collections of various data and more complex entities. *Objects* add tremendous power to Javascript and penetrate almost every aspect of the language. Here are a few different objects, all these are actually objects in JavaScript, which makes the language rather elegant:

- arrays
- functions
- regular expressions
- objects [1]

Objects are used to collect and organize data - and that data can be variable values, functions and other things. Objects can also contain other objects(!), in kind of a "nesting" way. This allows for large data structures to be built using a very simple and elegant mechanism.

An object (in this case an *object literal*) can be created with curly braces {…} with an optional list of properties. A property is a “key: value” pair, where key is a string (also called a “property name”), and value can be anything. [2]

```
let katniss = {
    firstname: "Katniss",
    lastname: "Everdene",
    homedistrict: 12,
    skills: ["foraging", "wildlife", "hunting",
"survival"]
}
```

Notice how you use commas for all but the last one in the list?

18.1. Object Creation

We can imagine an object as a container where everything has a key, and we can get data from the key, add data with a key to an object, and delete a key (with its data value) from the object. Each pair (of key and value) can be referred to as a **property**. Properties can be almost any JavaScript data type at all. Let's create a sample player object for a game:

```
let player1 = {
    name: "Rocco",
    species: "warrior",
    super_power: flight,
    lives: 4,
    weapons: {
        sword: "Excalibur",
        bow: "Legolas",
        arrows: 12
    }
};
```

A property has a key (also known as "name" or "identifier") before the colon ":" and a value to the right of it.

Here 'name' is a *key* and "Rocco" is the *value*. This 'player1' object has 5 key/value pairs (and you can add as many as you like, and can delete those you don’t need). One of those key/value pairs is a nested object, the 'weapons' key has another object with 3 key/value pairs inside of it. (Notice the commas ',' after all but the last item in each object.)

18.2. Property Retrieval

Values can be retrieved from inside an object by using a string expression in a `[ ]` (bracket) pair. You can also use "dot notation".

```
player1[name] // "Rocco"
player.name   // "Rocco"
player1.lives // 4
```

You get 'undefined' if you try to retrieve a value whose key doesn’t exist.

player1.nickname // undefined

18.3. Property Update

If we were to use an assignment statement, we can change the properties inside of the object.

```
player1.name = "Gina";
player1[species] = "Mage";
player1.weapons.sword = "Anduril";  // notice how
to reach into nested objects!
```

```
console.log(player1.weapons.sword);  // Anduril
```

18.4. Object Reference

This next little bit will stretch your mind a bit.

If we have an object:

```
let airplane512 = {
    type: "airbusA330",
    topSpeed: 0.86, // Mach
    passengerLimit: 253,
    passengersOnboard: 0
}
```

We have built a plane object.

```
let flight77 = airplane512;  // I'm assigning a
plane to a flight

flight77.destination = "Tokyo"; //create two new
properties in the object
flight77.origin = "Seattle";

// now watch carefully...
flight77.passengersOnboard = 120;

// and if I print out some of the properties...
console.log(airplane512.passengersOnboard);  //
120
console.log(flight77.type); // airbusA330
console.log(airplane512.type); // airbusA330
```

This is because both flight77 and airplane512 _refer to the **same** object. so when you make changes through the flight77 name, the properties in the object change, and you can get and/or set properties through the airplane512 name as well. This proves very powerful in many cases.

18.5. Delete Properties

Properties can be deleted as well.

```
delete flight77.destination;
delete flight77.origin;
```

18.6. Object Functions

Now to make things even more powerful, JavaScript objects can have *functions* attached to them. Let's build a special counter object, something we might use in a program or app somewhere.

As you may be aware, in American football there are four common changes to a team's score during a game:

```
let scoreCounter = {
    options: {"fieldgoal": 3, "touchdown": 6,
    "point_after_touchdown": 1, "safety": 2},

    score: 0,

    touchdown: function () {
        this.score += this.options["touchdown"];
    },
    fieldgoal: function () {
        this.score += this.options["fieldgoal"];
    },
    pat: function () {
        this.score +=
this.options["point_after_touchdown"];
    },
    safety: function () {
        this.score += this.options["safety"];
    },
    get_score: function () {
        return this.score;
    }
}
```

We can use that object, with its function methods like this:

```
scoreCounter.touchdown(); // add 6
scoreCounter.pat(); // add 1
scoreCounter.fieldgoal(); // add 3

console.log(scoreCounter.get_score()); // ??
```

There are two data properties (options and score), and 5(!) functions. These functions are called **methods** (functions which are attached to an object), and get invoked when you make the *method call* (or *invoke* the method on the *object*).

Notice the **this** variable. *This* is the special variable used to refer to the *object itself*. (which is a rather advanced topic for this book, so, we'll leave it right there. When you get a chance, read about *this* in a deeper JavaScript resource.)

18.7. Follow Ons

We have tried to give you some of the very basic parts of JavaScript, in order for you to be able to do well on the Zip Code Wilmington assessment. (Or for you to get a very basic understanding of coding in JavaScript and whether or not you enjoy learning this sort of thing.)

There are a number of very powerful things we have left out of this discussion about JavaScript objects. We have not covered the ideas of **prototypes** and the **prototype chain** here, which are not really needed for the assessment you may be taking.

We also have not discussed an extremely powerful concept, **closures**. But rest assured, there is much much more for you to learn about Objects in JavaScript.

Master what we've written about here and then forge ahead into more complicated and powerful capabilities.

There is a lot more to learn about JavaScript. But you made it this far, so perhaps you have what it takes to learn the 21st century super-power of coding.

[1] Now, unlike a lot of languages, JavaScript has no notion of *classes*. It uses a different model of *prototypes*.

[2] In some languages, a listing of key/value pairs is called a dictionary, an associative array or a hashtable.

Appendix A: Advanced Ideas

We’re going to look at a few "modern" ways of handling a collection of data. Frequently, you have a list, or an array, of data that needs to be gone through to print it out, transform it in some way, or to summarize it (such as a total or an average). As you have seen in the code patterns section, there are common loops used for such things, a simple pattern that you can memorize.

There are other methods of doing these things, and we’re going to discuss a few of them here. These ideas are based primarily on methods made popular by Hadoop and other "big data" applications and tools. And what’s good for "big" data is often good for "small" data as well.

Each of these sections is an example of a more "elegant" way of expressing coding logic. By studying each one and comparing it to the ways we’ve discussed before using loops and conditional statements, we’re expanding your understanding, making you see how these techniques can be used to create more extensible and elegant code.

Let’s use this array for the following examples.

```
const groceries = [
  {
    name: 'Breakfast Cereal',
    price: 5.50,
  },
  {
    name: 'Rice',
    price: 14.99,
  },
  {
    name: 'Oranges',
    price: 6.49,
  },
  {
    name: 'Crackers',
    price: 4.79,
  },
  {
    name: 'Potatoes',
    price: 3.99,
  },
];
```

A common grocery list, we have this as a list (or array) of objects (what's known as a key/value data structure).

A.1. Simplifying Loops

Now, if you wanted to print out each item's name in the grocery list to the console, you could do something like this:

```
let idx = 0;
while (idx < groceries.length) {
  console.log(groceries[idx].name);
  idx = idx + 1;
}
```

This is a very common code pattern in JavaScript. It's also fraught with possible errors. It relies on the `idx` variable. If we forget or mess up the increment step at the end of the loop, we could be in trouble. Rather, how about this:

```
groceries.forEach((item) => {
  console.log(item.name);
});
```

`forEach` is a *higher-order function* that takes in another function as an argument and executes the provided function once for each element in the array. It is meant to simplify your code. By using forEach, we remove the extraneous code for tracking and accessing the array using an index, and focus on our logic: printing out the name of each grocery item.

A higher-order function is a function that does at least one of the following:

- takes one or more functions as arguments (i.e. procedural parameters),
- returns a function as its result.

Let’s look at a few other uses of higher-order functions. Say we wanted to get a list of just the prices of our grocery list. If we use a loop, we have a very recognizable pattern.

```
let index = 0;
const prices = [];
while (index < groceries.length) {
  prices.push(groceries[index].price);
  index = index + 1;
}
```

But if we use a different higher-order function, `map`:

```
let prices = groceries.map((item) => {
  return item.price;
});
```

The value of prices would be: `[5.5,14.99,6.49,4.79,3.99]`. And if we were to want something a little more useful than just producing a list, we would do this when using a loop:

```
let index = 0;
let total = 0;
while (index < groceries.length) {
  total = total + groceries[index].price;
  index = index + 1;
}
```

Now we are tracking *two* pieces of information, the index and the total. With the sum of all the prices ending up in `total`.

But look how much simpler our code can be if we use `reduce`, another higher-order function.

```
let total = groceries.reduce((sum, item) => {
  return sum += item.price;
}, 0);
```

The result, if we print `total` is the awkward number `35.760000000000005`, and why that is, well, later we'll discuss it. But, `35.76` should suffice.

If we wanted to pull out all the even numbers from a list of numbers, we'd need a function like this to decide if a number if even. (Remember the trick using modulus?)

```
function even (value) {
    return value % 2 === 0;
}

console.log(even(3), even(4), even(126));

// giving us 'false true true'
```

If we remove the name `even` from the definition, and use a higher-order function named `filter`, we have something like this.

```
numList = [1,2,3,4,5,6,7,8];
numList.filter(
    function (value) {return value % 2 === 0}
);
```

`filter` is a function which filters out the elements in an array that don't pass a test. You can visualize it this way:

```
function filter(array, test) {
  let passed = [];
  for (let element of array) {
    if (test(element)) {
      passed.push(element);
    }
  }
  return passed;
}

function even (value) {
    return value % 2 === 0;
}

let dataList = [1,2,3,4];

console.log(
    filter(dataList, even)
); // produces [2,4]
```

See what I've done? First, I've shown you how to express `filter` with both a loop and an `if` statement, expressing the function in more verbose code to give you the idea of what's going on. Second, I've then used it to show it in action.

I've defined two functions, `filter` and `even`. Then created a short array/list called `dataList`. Finally, I've printed the result of calling `filter(dataList,even)`. Wait, what? I passed the function's name, `even` as an argument to another function. Well, sure, why not? Functions in JavaScript are called `first class objects`, just like a variable or an object or a value.

And it turns out JavaScript already has a function called `filter`, a higher order function. And I can reduce it even more to something like this:

```
let result = [1,2,3,4].filter(
    function (value) {return value % 2 === 0}
);
```

```
// or, If I have 'dataList' defined as [1,2,3,4]
let result = dataList.filter(
    function (value) {return value % 2 === 0}
);
```

And as you will see further down, we can even reduce that to a simpler form, called a `lambda`.

In these four cases, we see how we can use a different form of computing, a *functional* form, to simplify our code by removing loops and their trappings and replacing them with higher-order functions, letting us hand some of our logic to the language itself. And making our code more elegant in the process.

A.2. Simplifying Conditionals

We can use the same ideas with conditionals. Conditionals can get thick and complicated without too much effort. Say we need to keep track of and perform different discounts for various purposes. Sounds like an `if` statement! With `else` statements too!

But `else` statements, for instance, have a habit of complicating code.

Every time you add an else statement, you increase the complexity of your code two-fold. Conditional constructs like if-else and switch statements are foundational blocks in the world of programming. But they can also get in the way when you want to write clean, extensible code.

Let’s create a function that computes a discount for a price amount based on sone discount code. We might, happily, build something like this:

```
const discount = (amount, code) => {
  if (code == 'TWENTYOFF') {
      return amount * 0.80;
  } else if (code == 'QUARTEROFF') {
      return amount * 0.75;
  } else if (code == 'HALFOFF') {
      return amount * 0.50;
  } else { // no discount
      return amount;
  }
}; // whew! that a lot of braces.

let netprice = discount(200.00, 'HALFOFF'); //
would be 100.
```

But think about adding another discount, we'd have to add another `if`, more braces, and make sure we nest it in there carefully, otherwise we break the whole, rickety, mess.

I know! Let's use a `switch` statement, and simplify! Well…

Switch statements too, have a way of expanding on you, getting long, and sometimes complex, requiring care to maintain and/or extend. Say you wanted to add some more discounts to the following switch statement?

```
const discount = (amount, code) => {
  switch (code) {
    case 'TWENTYOFF':
      return amount * 0.80;
    case 'QUARTEROFF':
      return amount * 0.75;
    case 'HALFOFF':
      return amount * 0.50;
  }
};

let netprice = discount(200.00, 'HALFOFF'); //
would be 100.
```

We have to add two lines of code for each `case`. And if you make a mistake, you break the whole contraption.

But consider this idea: use a combination of a simple data structure and a small piece of code (called an arrow function (or "lambda")).

```
const DISCOUNT_MULTIPLIER = {
  'TWENTYOFF': 0.80,
  'QUARTEROFF': 0.75,
  'HALFOFF': 0.50,
};

const discount = (amount, code) => { // look at
that arrow '=>'?
  return amount * DISCOUNT_MULTIPLIER[code];
};
```

Whoa! How easy is it to add another 1, 3 or 7 discount cases? Just one line each. This re-factor effectively decouples the data we use from the core calculation logic, which makes it much easier to modify either independently. No `ifs`, `elses` or `switches`, just an object holding data, a simple lambda (arrow) function which does simple math.

A.3. Lambdas (or Arrow Functions)

One of the ways we do a lot of this kind of simplification within code is by replacing more complex logic with simpler forms.

In JavaScript, we have function expressions which give us an anonymous function (a function without a name). Here we are creating an anonymous function and assigning it to a variable.

```
var anon = function (a, b) { return a + b };

// this is the same as
function anon (a, b) {
    return a + b;
}
```

It’s really just a different form of the same thing.

But we also have `lambdas` or `arrow functions` with a more flexible syntax that has some bonus features and gotchas. We could write the above example as:

```
var anon = function (a, b) { return a + b }; //
from above

var anon = (a, b) => a + b; // Sweet!
```

```
// or we could
var anon = (a, b) => { return a + b };
// if we only have one parameter we can loose the
parentheses
var anon = a => a + a;
// and without parameters
var () => {} // this does nothing. So who cares?

// this looks pretty nice when you change
something like:
[1,2,3,4].filter(
    function (value) {return value % 2 === 0}
);
// to:
[1,2,3,4].filter(value => value % 2 === 0);
```

See how much easier it is to read the last line in the example over the previous filter using the anonymous function? Lambdas are a powerful way to express small functions, and use them in a variety of ways. They are often paired with higher-order functions, as they simplify the code quite a bit.

A.4. Polymorphism and K.I.S.S.

Remember "keep it simple, stupid"? Yeah, we suffer from over-complicating things in coding as well. Another way to replace conditionals is by using a key feature of object-oriented programming languages: polymorphism. Let’s show some code which helps bill a customer.

```
// list of customers we want to 'checkout'
```

```
const customers = [
  {
    name: 'sam',
    amount: 75.00,
    paymentMethod: 'credit-card',
  },
  {
    name: 'frodo',
    amount: 50.00,
    paymentMethod: 'debit-card',
  },
  {
    name: 'galadriel',
    amount: 25.00,
    paymentMethod: 'cash',
  },
];
```

I'm going to gloss over the code needed to do each of the three kinds of payment. But show you how I might have to account for all three inside a `checkout` function.

```
const checkout = (amount, paymentMethod) => {
  switch (paymentMethod) {
    case 'credit-card':
      // Complex code to charge ${amount} to the
credit card.
      break;
    case 'debit-card':
      // Complex code to charge ${amount} to the
debit card.
      break;
    case 'cash':
      // Complex code to put ${amount} into the
cash drawer.
      break;
  }
};
```

Now, I'd like to take the list of customers, and checkout each one. (Notice how I'm using the higher-order function here, not a `for` loop.)

```
customers.forEach(({ amount, paymentMethod }) => {
  checkout(amount, paymentMethod);
});
```

But if I use `polymorphism`, I can make each customer's checkout method wired directly to the data list. And look how I have broken the large function up, into three simpler things.

```
class CreditCardCheckout {
  static charge(amount) {
    // Complex code to charge ${amount} to the
credit card.
  }
}
class DebitCardCheckout {
  static charge(amount) {
    // Complex code to charge ${amount} to the
debit card.
  }
}
class CashCheckout {
  static charge(amount) {
    // Complex code to put ${amount} into the cash
drawer.
  }
}
const customers = [
  {
    name: 'sam',
    amount: 75.00,
    paymentMethod: CreditCardCheckout,
  },
  {
    name: 'frodo',
    amount: 50.00,
    paymentMethod: DebitCardCheckout,
  },
  {
    name: 'galadriel',
    amount: 25.00,
    paymentMethod: CashCheckout,
  },
];
customers.forEach(({ amount, paymentMethod}) => {
  paymentMethod.charge(amount);
});
```

I am using a `class` in this example, well, three of them actually. One for each payment method. I can put the complex code within each class, and if I set them all up to have a `charge` method (a method being the term we use to talk about a function wired to a class), I know I just need to call `charge` on each customer, and the classes will all figure out which piece of code to use. This is an example of polymorphism, "many forms, same name".

Another example, commonly used in explaining polymorphism, is a series of geometric shapes, like Square, Triangle and Circle. Each of those shapes has a different way of computing the `area` of itself. A Square's `area()` is `(side * side)`, right? But a Circle's `area()` is `(Math.PI * (radius * radius))`. Two different ways of calculating the area of a shape, depending on the kind of shape we're working with. Each of these shapes would have it's own class, each with a different definition of how to find the area of the shape. That's polymorphism in a nutshell.

Each of these techniques are currently considered "advanced" JavaScript, even though in many cases they are simpler and less error-prone than more "traditional" loops and conditionals.

Be sure to consider how each of them are largely the same in functionality but simpler in expressing the logic of your program. Remember to make your code more elegant by adding more simplicity.

Appendix B: Mars Lander

This is some code to show you how you might write a simple mars lander simulation in JavaScript. It's taken from history, way back in the 1970's - this idea was passed around as some of the very first open source programs.

Meant as an example of a longer program (159 lines) to get you thinking, it's really not very complicated. The general idea is you have a series of "burns" in a list, and the game (or simulator, if you will) steps through the list applying each burn. If you run out of altitude (or height) while you're going too fast, you will crash.

The tricky bit would be for you to figure out what `burnArray` would be used to safely land at a vehicle speed 1 or 2. That could be hard.

```
// Mars Lander Source Code.

const Gravity = 100
/* The rate in which the spaceship descents in
free fall (in ten seconds) */

const version = "1.2"; /* The Version of the
program */

// various end-of-game messages.
const dead = "\nThere were no survivors.\n\n";
const crashed = "\nThe Spaceship crashed. Good
luck getting back home.\n\n";
const success = "\nYou made it! Good job!\n\n";
const emptyfuel = "\nThere is no fuel left. You're
floating around like Wheatley.\n\n";
```

```
function randomheight() {
    let max = 20000;
    let min = 10000;
    let r = Math.floor(Math.random() * (max -
min)) + min;
    return (r % 15000 + 4000)
}

function gameHeader() {
    s = "";
    s = s + "\nMars Lander - Version " + version +
"\n";
    s = s + "This is a computer simulation of an
Apollo mars landing capsule.\n";
    s = s + "The on-board computer has failed so
you have to land the capsule manually.\n";
    s = s + "Set burn rate of retro rockets to any
value between 0 (free fall) and 200\n";
    s = s + "(maximum burn) kilo per second. Set
burn rate every 10 seconds.\n"; /* That's why we
have to go with 10 second-steps. */
    s = s + "You must land at a speed of 2 or 1.
Good Luck!\n\n";
    return s;
}

function getHeader() {
    s = "";
    s = s + "\nTime\t";
    s = s + "Speed\t\t";
    s = s + "Fuel\t\t";
    s = s + "Height\t\t";
    s = s + "Burn\n";
    s = s + "----\t";
    s = s + "-----\t\t";
    s = s + "----\t\t";
    s = s + "------\t\t";
    s = s + "----\n";
    return s;
}


function computeDeltaV(vehicle) {
    return (vehicle.Speed + Gravity -
vehicle.Burn)
}

function checkStatus(vehicle) {
    s = "";
    if (vehicle.Height <= 0) {
        if (vehicle.Speed > 10) {
            s = dead;
        }
        if (vehicle.Speed < 10 && vehicle.Speed >
3) {
            s = crashed;
        }

        if (vehicle.Speed < 3) {
            s = success;
        }
    } else {
        if (vehicle.Height > 0) {
            s = emptyfuel;
        }
    }
    return s
}
```

```
function adjustForBurn(vehicle) {
    // save previousHeight
    vehicle.PrevHeight = vehicle.Height;
    // compute new velocity
    vehicle.Speed = computeDeltaV(vehicle);
    // compute new height of vehicle
    vehicle.Height = vehicle.Height -
vehicle.Speed;
    // subtract fuel used from tank
    vehicle.Fuel = vehicle.Fuel - vehicle.Burn;
}

function stillFlying() {
    return (vehicle.Height > 0);
}

function outOfFuel(vehicle) {
    return (vehicle.Fuel <= 0);
}

function getStatus(vehicle) {
    // create a string with the vehicle status on
it.
    let s = "";
    s = vehicle.Tensec + "0 \t\t" + vehicle.Speed
+ " \t\t" + vehicle.Fuel + " \t\t" +
        vehicle.Height;
    return s
}

function printString(string) {
    // print long strings with new lines the them.
    let a = string.split(/\r?\n/);
    for (i = 0; i < a.length; i++) {
        console.log(a[i]);
    }
}

// this is initial vehicle setup
var vehicle = {
    Height: 8000,
    Speed: 1000,
    Fuel: 12000,
    Tensec: 0,
    Burn: 0,
    PrevHeight: 8000,
    Step: 1,
}

// main game loop
function runGame(burns) {
    let status = ""

    /* Set initial vehicle parameters */
    let h = randomheight()
    vehicle.Height = h;
    vehicle.PrevHeight = h;

    burnIdx = 0;

    printString(gameHeader());
    printString(getHeader());

    while (stillFlying() === true) {

        status = getStatus(vehicle);

        vehicle.Burn = burns[burnIdx];
        printString(status + "\t\t" +
vehicle.Burn);
```

```
        adjustForBurn(vehicle);

        if (outOfFuel(vehicle) === true) {
            break;
        }
        vehicle.Tensec++;
        burnIdx++;


    }
    status = checkStatus(vehicle);
    printString(status);
}

// these are the series of burns made each 10 secs
by the lander.
// change them to see if you can get the lander to
make a soft landing.
// burns are between 0 and 200. This burn array
usually crashes.
const burnArray = [100, 100, 200, 200, 100, 100,
0, 0, 200, 100, 100, 0, 0, 0, 0];

runGame(burnArray);
```

Appendix C: Additional JavaScript Resources

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

Some JavaScript sites for you to explore:

- https://javascript.info
  - https://javascript.info/first-steps
- https://eloquentjavascript.net
- http://jsforcats.com
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide

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.)