Zip Code Wilmington's Programming in JavaScript

Kristofer Younger

Version 1.3, 2020-08-6

Table of Contents

Preface

I’d like to thank the instructors of ZiCodeWilmington for help editing this book: Chris Nobles, Roberto DeDeus, Tyrell Hoxter, L. Dolio Durant, and Mikaila Akeredolu. Without them, this book would have remained a "nice to have". 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 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 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 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 the <repl.it> web site for the running of the JavaScript examples you'll find in this book. You run almost each code snippet and *see* what the code does.

Gets you focused 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 an 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?), when it comes right down to it, 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 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".

Well, I'm here to tell you that all the excuses in the world don't stop a real writer from writing. "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.

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

The only way you're going to learn to code, is by doing it. Making mistakes, fixing them, learning from what worked and what didn't.

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. I've seen time and time again, 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 <HackerRank>. 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 programing 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.

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.)

Working on code, everyday, 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 final interviews. And then, well, then you get to learn Java or Python and work yourself to exhaustion doing so. 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? Right? RIGHT?

Okay, let go.

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

Programming with JavaScript

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!". [2]

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://repl.it and create an account for yourself. Once that's done you should create a JavaScript repl, a place where you can type the code from this book to the repl 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);
```

Which, if you typed into your repl and clicked the "Run >"

button, you saw

```
Car's Speed: 55
```

as the program's output.

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

[2] 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."

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).

```
// This comment occupies a line of its own
console.log('Hello');

console.log('World'); // This comment follows the
statement
```

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 concole.log line is showing what you can expect to see in the output. Here it would be "3.5" printed by itself.

We can also added 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 cases.

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

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**.

Expressions

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

```
2 + 2 * 65536

speed > 55.0

regularPrice * (1.0 - salePercentOff)
```

Each of these lines is someting 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.

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 expression in them (like totalGroceries * salesTaxRate), some are just simple **assignment** statements (like totalGroceries = 38.99, where we assign the variable 'totalGroceries' the value 38.99) Don't panic, thse 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.

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.

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 JavScript code. == Variables and Data Types

What are 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 _
- 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.

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
```

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
```

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.

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 capitalized 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.

JavaScript 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.

Arithmetic Operators

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

Basics

Operator	Name	Description
+	Addition	Add two values
-	Subtraction	Subtract one from another
*	Multiplication	Multiply 2 values
/	Division	Divide 2 numbers
%	Modulus	returns the remainder
++	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
```

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

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

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

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 we have, sometimes, 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
```

Order is Important

A strange thing about these operators is the order in which they 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 x 2" part first, giving us 12.
  - then we'd add the 30 to 12 giving us 42 [3]
- But say we don't like multiplication, and want to save it for later
  - we would then do the add first, 2 + 30 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.

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 5 * 3 / 2, we would do the multiply first and then the divide. 6 * 3 / 2 is 9

Add and Subtract rank equally (and go left to right) So if we have 9 - 6 + 5, we do the subtract first and then the 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 multiply is *higher priority* than the addition, _even though the addition is before the multiply in the expression. Let's check it in JavaScript.

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

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?

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 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. == JavaScript 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
```

Trigonometry

The word trigonometry comes from the greek word - 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**

(TODO: INSERT a simple triangle diagram)

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)
```

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

Simple Calculation Programs

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.

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, about how 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.) There are 52 weeks in the year and 7 days in the week. (Sometimes you just gotta say the Very Obvious to make sure you're thinking it all through.) But, we know there are 365 days in a 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?

// print result of computation
console.log(totalCost);
```

What's going to be the answer? [4] Run it in your Repl.it window. Notice how the weeks stuff disappeared wasn't used in this problem. Sometimes that happens, a problem describes some stuff you can ignore.

And let's do one more.

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…) [5]

[4] totalCost will be $243.33

[5] total data used (GB) 11.05

Boolean Expressions

Many times the idea of a boolean variable, something that is either 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 sme 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. So when if the 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. 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**.

Comparison Operators

```
let healthScore = 5;
```

is healthScore less than 7?? (very healthy)

is healthScore greater than or equal to 3?? (maybe barely alive?)

Operator	Description	Example
==	Equal to	x == 5
===	Equal value and type	x === '5'

Operator	Description	Example
!=	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.
```

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.

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

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

```
(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 impoerant 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. == 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 program. 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.

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, 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. 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!\""
```

Strings can be thought of tightly packed list. Each item, each letter, is numbered. The entire string can be "indexed", meaning I can reach in 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 and 3, 'o' is at index position 4. Computers often start lists at 0, not at one. It just one of those things.

Declaring a string

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

Now we have a string variable named **name** and it's value

is "Wacka Flocka".

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
```

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.

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.
```

SubStrings

The 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. (Although, honestly, they all do exactly the *same thing!*)

*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!)

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")

Well?

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

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

Summary of substring methods

```
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
```

Reverse a String

To Reverse a String

Now let's reverse the string STOP to say POTS

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

Step2 - Use the reverse() method to reverse the newly created array of string characters

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

Print out the reversed 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.

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 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* and 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. If we need to deduct health points from one of the players we can do something like this

```
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?

JavaScript Arrays

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

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

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!
```

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"
```

Add things tos 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.
```

Getting 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

Grabbing 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
```

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.

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.

Conditional Statements

While we have been seeing programs which consist of of a list of statements, one after another, where the "flow of control" goes from one line to the next, 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.

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(ammountRequested);
    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 get run, and the customer gets an insuffcient 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 excuted 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 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"

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 th emost commonly used statements to express logic in a JavaScript program. It's important to know them well. === TODO Switch Statements

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

```
let timeOfDay = "Afternoon";
switch(timeOfDay){
    case "Morning":
        console.log("Time to eat breakfast");
        break;
    case "Afternoon":
        console.log("Time to eat lunch");
        break;
    default:
        console.log("Time to eat dinner");
}
```

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

- 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 ===

You should use a switch statement. Finally, make sure to change the value of the age variable to output out different results.

== 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 prupose of loops is to avoid having lots of repetitive code.

=== 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 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. 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, 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'.

=== 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);
```

=== 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);
}
// print
1
2
3
4
5
```

Notice how there are THREE parts to the 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.
- body console.log() Runs again and again while the condition is true.
- step j++ Executes after the body on each iteration.

==== Break

```
for(let p = 1; p &lt; 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. === 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(command);
}
console.log("Exiting.")
```

Here 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.

=== 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. Here the expression (i % 2) is zero if the number is even, it not the number must be odd. You want to remember this trick, how to find odd or even numbers. It's 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. === 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. == TODO Functions in JavaScript

```
A function is a block of code designed to
perform a particular task. Functions get
executed when invoked
```

Functions let you avoid duplicating code and organize your code.

Functions are invoked when:

- An event occurs (when a user clicks a button)
- It is invoked (called) from JavaScript code

=== JS Function Syntax

```
function | NameOfFunction | (Parameters){
//Logic goes here
}
//----------------------------------------
function myFunction(parameter1, parameter2) {
return parameter1 * parameter2;
}
```

=== Creating and using a function

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

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

=== 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"

```
function greetUser(username) {
return "Hello " + username;
}
console.log(greetUser("Welcome back, Mike
Jones"));
// function returns a string to be printed on
console
```

=== Function Parameters

```
function printReceipt(price, productName, tax)
{
//this method has 3 parameters
}
```

- 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. Print Zip
- If parameter is divisible by 5. Print Coder
- If parameter is divisible by 3 and 5. Print ZipCoder Phew…Finally
- Call the method and pass in 45 as your parameter

=== 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
```

== 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 your JavaScript programs are fairly small, when you are creating solutions to HankerRank type problems. They can be thought of 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 "import" and "export" to see how modules interact with your code.

== TODO Objects

There are eight data types in JavaScript. Seven of them are called "primitive", 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. In JavaScript, objects penetrate almost every aspect of the language. So we must understand them first before going in-depth anywhere else.

An object can be created with figure brackets {…} 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.

We can imagine an object as a cabinet with signed files. Every piece of data is stored in its file by the key.

It’s easy to find a file by its name or add/remove a file.

(more needed)