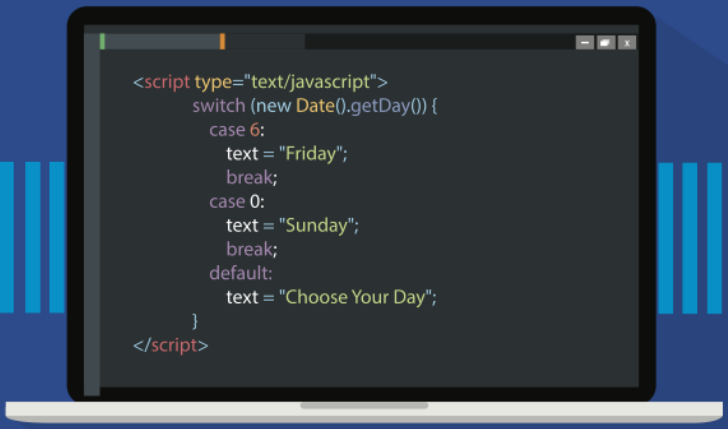
Zip Code Wilmington's Programming in JavaScript

A CRASH COURSE IN CODING



WorkBook

ZIPCODE
<WILMINGTON>
SCHOOL OF CODING

Zip Code Wilmington's Workbook for JavaScript

Kristofer Younger

Version 1.8.4, 2021-02-02

Table of Contents

Colophon

Zip Code Wilmington's Workbook for JavaScript by Kristofer Younger

Cover Design: Janelle Bowman

Published in the U.S.A.

April 2021: First Edition

Preface

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 exercises to assist in learning the most basic fundamentals of 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 completing these exercises, reading the companion *Fundamentals of Javascript* book and/or taking one of ZipCodeWilmington's preparation sessions, someone who has never coded in any language before can use this as a place to start, to study and to get ready to take the ZipCodeWilmington admissions assessment. 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 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.

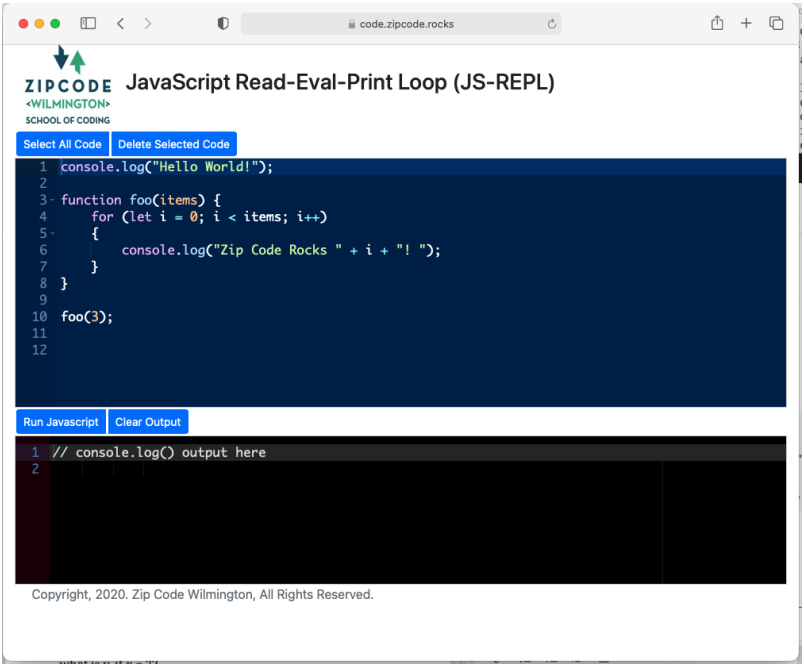
[1] check out his terrific work: https://learncodethehardway.org

Chapter 1. Using the REPL

A "REPL" is a special kind of program. It allows you to type in a Javascript program, run it, and see what happens. Even though it is simple and easy to use, it is a pretty powerful computer programming tool.

The one we've created for your use, *https://code.zipcode.rocks*, let you type in various JavaScript programs and run them showing you the results.

Let's try the sample on that is built-in to the REPL page. Use a browser to go to the URL *https://code.zipcode.rocks*. You should see something like:



Take a look at the black section. What does it say?

Notice the two sections of blue buttons, click the one that says **Run Javascript**. (It's in the middle of the page). Now, what does the black section say?

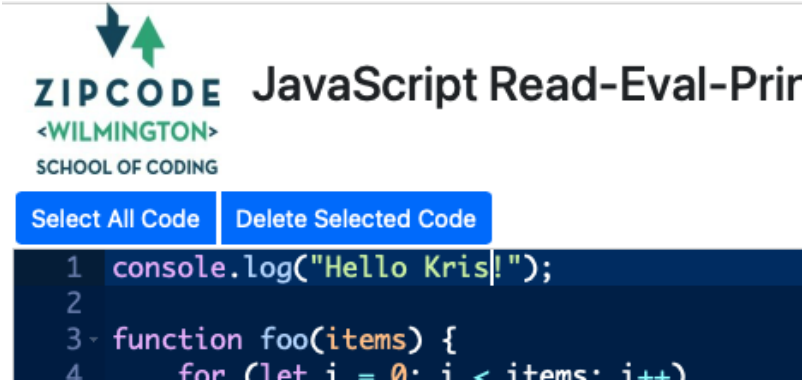
That is how you run JavaScript in this REPL.

Now, what does **REPL** stand for? Well, it means "Read Evaluate Print Loop" (R-E-P-L). A REPL is a program that reads in your code (here it is JavaScript), evaluates what it means, and then prints out any output the program might have. Here, we are using the command *console.log()* to print out a series of output values.

1.1. Editing JavaScript Code

In the REPL window, the blue part, you can change the JavaScript code. It works like a text editor or a word processor.

Change Line 1 in the blue section from "Hello World" to "Hello YourName" (see, I've changed *World* to *Kris*) (and use your real name there, not *YourName*).



Now click the blue **Run Javascript** button, what did the

black section change to??

Pretty cool, huh?

Notice that the source code is several colors. This is because many code editing programs today, like our REPL here, use different colors to point out different parts of the JavaScript program. Here, numbers are in orange, Strings are in green, and special JavaScript *reserved words* are in a pinkish color. This *syntax coloring* is used to indicate extra information to the coder.

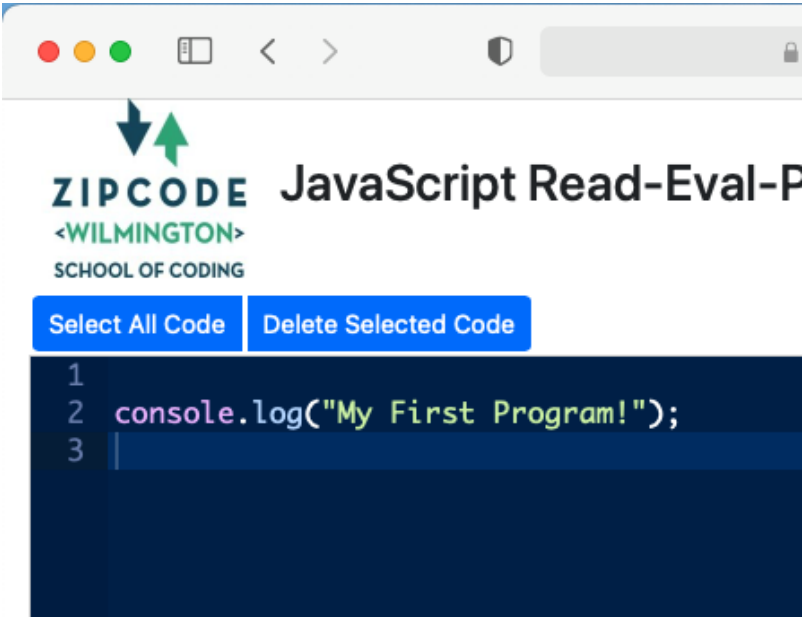
1.2. A First Program

Let's write your first program. First, click the **Select All Code** blue button. Second, click the **Delete Selected Code** blue button. This empties the blue editor of all the code that was there.

Now, type onto line 1,

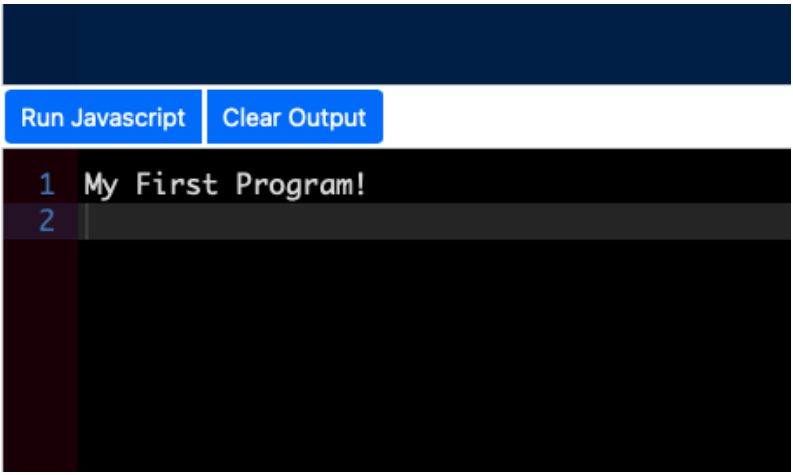
```
console.log("My First Program!");
```

Take care to notice the double quotes (") AND notice the semicolon (;) at the end of the line.



And now click the **Run Javascript** blue button. What does the black output section say now?

Does it look like this?



Congratulations! You've just written a Javascript program.

Maybe you are Zip Code material after all.

1.3. Coding Assessments

To attend a Zip Code Wilmington cohort, Zip Code asks you to complete a JavaScript coding assessment. Usually, this is done on a website like HackerRank. By asking you to solve some pretty simple programming challenges, we get a chance to see how you do when learning something entirely new. The ability to learn new and different things continually is a very important aspect of building a professional career as a coder. As you will see, if you do make coding a professional endeavor, learning is something you will do all the time.

Chapter 2. Algebra Exercises

Let's look at a series of problems that you can solve to get a feel for what we're looking for on the Zip Code admissions assessment. If you're able to reason through these problems, you should just fine on the assessment.

Feel free to use the https://code.zipcode.rocks REPL to work these problems out. (You can also just work them out in your head or on paper)

Computing is often programming a computer to do math. The use of variables, like in algebra, can make the computation quite flexible. By solving these problems, you'll get a feel for the kind of easy algebra problems we'd like you to be able to solve.

As with many kinds of programming tasks, these problems can be solved a large number of ways.

2.1. Y equals ?

Look at each of these formulas, and determine what **y** is based on substituting the given value of **x**.

```
If y = 3x + 5 and x = 2
```

```
x = 2

y = 3x + 5

y = (3 times x) plus 5

y = (3 times 2) plus 5

y = 6 plus 5

y = 11
```

Now, here are a few problems for you to solve.

2.1.1. ?

```
Given the following formula,

y = 6x - 7

what is y, if x = 2?
```

2.1.2. ?

```
Given the following formula,

y = 3x + 8

what is y, if x = 5?
```

Multiply and Divide

When we are computing, we express math in a slightly different way than we do in Algebra class. Take your mind back to Mrs Johnson's 7th grade algebra class. (Weren't all our 7th grade algebra teachers named Mrs Johnson?? No? Are you sure?)

In algebra, we might say **4x** or **3x**. We mean, of course, **4 times x** or **3 times x**.

Well, in coding we usually write `3 * x` or `4 * x` - the * (an asterisk or shift-8 on your keyboard) means **multiply** or **times**.

Division, too is different. In algebra we used the old divide sign ÷ whenever we divided **9 ÷ 4**, but in JavaScript code we use a slash /. So, **5 divided by 2** is written **5 / 2**.

It's okay, you'll get used to * and / for times and divide.

```
Given the following formula,

y = (5 * x) + 100

what is y, if x = 3?
```

For each of these problems, you need to compute the value of **y**. In each case the value of **x** is different, and so is the formula you need to compute.

For this first one, if **x equals 3**, you would first multiply **5** times **3**, and then add 100 to it. So a correct answer is **15** + **100** or **115**.

Each of these below are computed in pretty much the same way.

2.1.3. ?

```
Given the following formula,

y = (4 * x) - 5

what is y, if x = 5?
```

2.1.4. ?

```
Given the following formula,

y = (6 * x) - 7

what is y, if x = 2?
```

2.1.5. ?

```
Given the following formula,

y = (15 / x) * 8

what is y, if x = 5?
```

2.2. Velocities

Velocity is often thought of as **speed**. It's common to drive at 40 or 45 miles per hour. That speed is a **velocity**. **Acceleration**, you may remember, is the change in velocity.

If a car is moving at an initial velocity V, and accelerates at rate of A for a set time period of T, the formula to determine it final velocity S is:

```
S = A*T + V
```

So if we have a **A=2**, **T=3**, and **V=4**, we would compute

```
S = (2 * 3) + 4

S = 6 + 4

S = 10
```

You substitute each variable (V, A and T) into the formula and compute the result.

2.2.1. ?

```
What is S if (V = 10), (A = 4), (T = 3)?
```

2.2.2. ?

```
What is S if (V = 0), (A = 5), (T = 20)?
```

2.2.3. ?

```
What is S if (V = 17), (A = 2), (T = 14)?
```

2.3. F of x or F(x)

Sometimes you will see a formula expressed as a function which takes one parameter, **x**. an *parameter* to a function is it's input value. This is an algebraic (or math) function, not a Javascript function.

```
F(x) = 3x + 4
```

This is just another way of describing a simple computation. The result of the function, F(x) (which is pronounced *F of x*), is the same as **y** was in the problems above.

```
y = 3x + 4
```

Y ended up being the answer to the computation. So is *F of x* is also the answer to the computation. Don't overthink this, **y** and **f of x** are just two ways of saying the same thing.

The other thing to notice is the **3x** portion. In JavaScript, you need to change it to **3 * x** so a correct form of a

JavaScript function would be

```
function f(x) {
    return (3 * x) + 4;
}
```

Write out these three formulas as JavaScript functions. Use the example above to help with your answers.

2.3.1. ?

f(x) = 5x - 7

2.3.2. ?

g(x) = 6x + 3

2.3.3. ?

h(x) = 123x - 68

Parameters and Arguments

As you learn to code, you will be using this thing called a **function**. Functions are the main **building blocks** of a program. They allow the code to be called many times without repetition.

So, a programmer bundles up smaller pieces of their program into functions.

In JavaScript, and many other programming languages (yes, there are many!), we often need to supply a function with data. And that data can often change.

When we supply a function with data, we **pass** some **parameters** to the function.

Say I have a function that does something really simple, it takes a number a divides it by 2:

```
function halve(x) {
    return x / 2;
}
```

In this example, **x** is a **parameter**. It is also called an **argument**. I'll try to use **parameter** throughout this booklet, but in some problems you may see on-line, you may also see **x** in the above function described as an **argument**.

2.4. F of (x, y)

Well, so if **F of x** is the result of a function of one parameter **x**, **F of x and y** is also the result of a function. But in this case, the functions takes two parameters, **x** and **y**.

Given the following function using x and y:

```
F(x, y) = 2x + y -27
```

And then if *F(1, 2) = -23*, well, how did we get that result?

The idea is that you think it through like this:

given F(1, 2), we assign x=1 and y=2 and substitute these two variables to compute the result.

F(1,2) = (2 * x) + y - 27

F(1,2) = (2 * 1) + 2 - 27

F(1,2) = 2 + 2 - 27

F(1,2) = 23

Using that as a **pattern**, work through these problems:

2.4.1. ?

```
f(20, 8) = ?
```

2.4.2. ?

f(6, 25) = ?

2.4.3. ?

f(5, 2) = ?

2.4.4. ?

f(0, 0) = ?

2.5. Complete the answer

For this problem, you need to figure out the correct answer to the last expression.

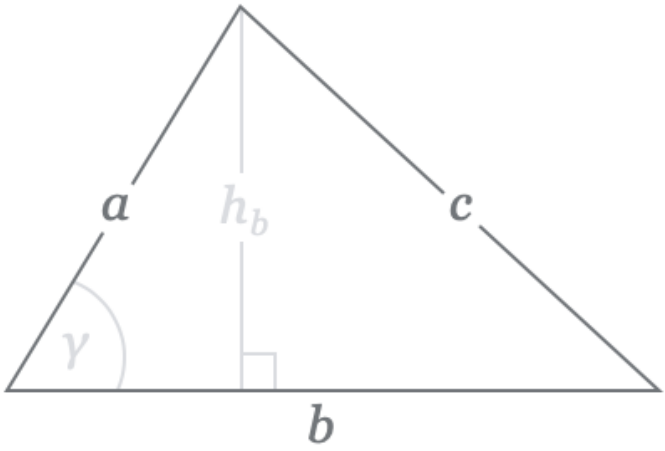
```
dog + fish = 25
dog + bird = 35
bird + fish = 30
```

If the above expressions are true, what is the answer to this?

```
dog + bird + fish = ?
```

Chapter 3. JavaScript Coding

3.1. Area of a Triangle



The area of a triangle is calculated using the following function:

```
Area = (h*b)/2
```

where Area represents the area, h represents the height perpendicular to the length of its base, b.

```
if h = 6 and b = 4,
(6 * 4) is 24
24 / 2 is 12
= 12
```

```
What is Area, if h = 4 and b = 5 ?
What is Area, if h = 2 and b = 6 ?
```

What do you need to do to this function to make it use the triangle equation?

```
function triangleArea(h, b) {
    area = 0;
    return area
}

console.log(triangleArea(4,5));
console.log(triangleArea(2,6));
```

3.2. Javascript Math

If asked to write a function to calculate and return a number divided by 2, you would code

```
function f(x) {
    return x / 2;
}
```

x is the parameter, the number we want to divide by 2. We

just `return x / 2`. So if x=6, the number returned by the function will be **3**.

If asked to produce a function that calculated a more complicated expression, perhaps

```
234 * 76 / 89 * 564
```

You could just write

```
function whoa() {
    return 234 * 76 / 89 * 564;
}
```

Use these as **patterns** to solve the problems below.

Complete the function to calculate multiplication of two input values. 4 x 4 should print 16.

```
function multiply(x, y) {
    return
}
```

Write a simple function that returns the remainder (modulus) of the following equation: 118 % 31

```
function foo() {
    return
}
```

Using Arithmetic Operators, write a function to return the product of 789 x 123.

```
function m() {
    return
}
```

Complete the function to calculate the square of an input value. Remember that the square of a number is the result of multiplying that number times itself.

```
function square(y) {
    return
}
```

3.3. Variables

Variables, as you will remember, allow a program to "name" various data and keep track of it as it changes during the running of a program. You can imagine that a computer game which tracks the scores of each player, would keep these values in variables as the game is played.

Write a function **one** which creates a variable called **t** with the value 1024 and return the variable.

```
function one() {



}
```

Write out a series of JavaScript variables.

```
a variable named foot equal to 12
a variable named fingers equal to 4
a variable named thumb equal to 1
a variable name isPlayerAlive equal to true
a variable named speed equal to 55.0
a variable named playerOne equal to a string "Joe"
a variable named PI equal to 3.14159
a variable named hand equal to 2 times fingers
plus 2 times thumbs
```

3.4. Functions

3.4.1. String

Make a function that returns the string value "Zip Code" from the given function.

```

```

3.4.2. Length of String

Using the JavaScript length method, return the length of the string "ZipCodeRocks" using the given function.

Example 1 input: "hello".length output: 5

Example 2 input: "Delaware" output: 8

3.5. Array

An array is a list of some kind of data. Here is an example of an array of fruits.

```
let fruits = ["Apple", "Orange", "Plum"];

console.log( fruits[0] ); // Apple
console.log( fruits[1] ); // Orange
console.log( fruits[2] ); // Plum
```

Notice how the phrase **fruits[0]** prints out just **Apple** from the array. The list of fruits is **ordered**, and **indexed** from 0 to 2. We can also find out the length of the array by asking for its **length** property.

```
console.log( fruits.length ); // 3
```

Now, use a **for** loop to print out each item of our **fruits** array.

First, need to build an empty function.

```
function printFruits() {
    let fruits = ["Apple", "Orange", "Plum"];
}
```

Second, we need to put in a loop that step through each string in the array.

```
function printFruits() {
    let fruits = ["Apple", "Orange", "Plum"];
    for (let i = 0; i < fruits.length; i++) {

    }
}
```

And finally, we print out each item in the array.

```
function printFruits() {
    let fruits = ["Apple", "Orange", "Plum"];
    for (let i = 0; i < fruits.length; i++) {
        console.log( fruits[i] );
    }
}
```

Before you try to solve this next problem, spend some time reviewing

Swap Two Elements

Complete the function below to swap two elements in an array and return the result. Your function should take three parameters: An array and two integers. The integers are the indexes of the two elements in the array you should swap.

Example 1:

Input: [7,4,9,3,6,2], 4, 2

Output: [7,4,6,3,9,2],

3.6. First and Last

This problem is a little harder. We need to take a string and copy out the first an last letters, and then return them reversed in order and with a space in between them.

First and Last

Given a string, create a new string made up of its first and last letters, reversed and separated by a space.

Example Given the word 'bat', return 't b'. Given the word 'motor', return 'r m'.

Function Description Complete the function lastLetters in the editor below.

lastLetters has the following parameter(s): string word: a string to process

Returns: string: a string of two space-separated characters

Constraint 2 ≤ length of word ≤ 100

3.7. Sum a List of Numbers

Below, we will be using a list of numbers. Imagine we have an array of numbers like this:

```
let numbers = [3, 8, 5, 7, 9, 2, 13];
```

Sum an Array

Calculate the sum of an array of integers.

Example numbers = [3, 8, 5, 7, 9, 2, 13]

The sum is 3 + 8 + 5 + 7 + 9 + 2 + 13 = 47.

Function Description Complete the function arraySum in the editor below.

arraySum has the following parameter(s): int numbers[n]: an array of integers Returns int: integer sum of the numbers array

This problem expects us to make a function that returns the sum of all the numbers in the array.

How are we going to do this?

Well, the first thing to solve this problem is to setup the function we will be writing.

```
function arraySum(aList) {

}
```

Okay, now we know we have to figure out the result of all the addition. Let's set up a variable called **runningSum** and set it to zero, and at the end of the function, return it as the result of the function. This will not give a working program yet, but it is meant to show how you can step through writing the function bit by bit.

```
function arraySum(aList) {
    let runningSum = 0;

    return runningSum;
}
```

This step by step fashion is very useful to get some of the fundamental pieces of the solution out of the way.

Now, we need to figure out how to *step through the array* and adding each number we find in there to the **runningSum** variable. To do that we are going use a **loop**. This **for** loop takes the **aList** parameter and *for each* item in **aList**, assigns it to **n**. We then add **n** to our **runningSum**. At the end of the loop, we return the value of **runningSum**.

```
function arraySum(aList) {
    let runningSum = 0;

    for (let n of aList) {
        runningSum = runningSum + n;
    }
    return runningSum;
}
```

And when we test it, it should give us the right answer. (Use the REPL to see)

```
function arraySum(aList) {
    let runningSum = 0;

    for (let n of aList) {
        runningSum = runningSum + n;
    }

    return runningSum;
}

let numbers = [3, 8, 5, 7, 9, 2, 13];

console.log( arraySum(numbers) ); // answer should
be 47.
```

Now, you change the loop used to the **for** loop with an index, and test it out.

```
function arraySum(aList) {
    let runningSum = 0;

    // use a loop which has an index *i* and uses
*aList[i]* to get each number.

    return runningSum;
}

let numbers = [3, 8, 5, 7, 9, 2, 13];

console.log( arraySum(numbers) ); // answer should
be 47.
```

Appendix A: 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.)